ComponentOne

# Reports for .NET Designer Edition

*Corporate Headquarters*
**ComponentOne LLC**
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 • USA

**Internet:** info@ComponentOne.com

**Web site:** http://www.componentone.com

**Sales**

E-mail: sales@componentone.com

Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

### Trademarks

The ComponentOne product name is a trademark and ComponentOne is a registered trademark of ComponentOne LLC. All other trademarks used herein are the properties of their respective owners.

### Warranty

ComponentOne warrants that the original CD (or diskettes) are free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective CD (or disk) to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for a defective CD (or disk) by sending it and a check for $25 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original CD (or disks) set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. We are not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

### Copying and Distribution

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.

This manual was produced using ComponentOne Doc-To-Help™.

# Table of Contents

# ComponentOne Reports for .NET Designer Edition Overview

**ComponentOne Reports™ for .NET Designer Edition** includes **ComponentOne Reports for WinForms** (which now includes the **ComponentOne Reports for .NET** and the **ComponentOne Preview for .NET** products) and the WYSIWYG **ComponentOne Report Designer** application. C1ReportDesigner is a Windows Forms control that provides the ability to design and edit report definitions that can be used with ComponentOne reporting controls.

> **Note:** See the ComponentOne Reports for WinForms Help for more information on using that product.

The C1ReportDesigner control displays reports in design mode, and allows users to drag, copy, and resize report fields and sections. The control also provides an unlimited undo/redo stack and a selection mechanism designed for use with the **PropertyGrid** control that ships with Visual Studio.

You can use the C1ReportDesigner control to incorporate some report design features into your applications, or you can write your own full-fledged report designer application. We include full source code for the C1ReportDesigner application that ships with **Reports for .NET Designer Edition** and uses the C1ReportDesigner control extensively.

Writing your own customized report designer is useful in many situations, for example:

- You may want to integrate the designer tightly into your application, rather than running a separate application. (For example, see the report designer in Microsoft Access).
- You may want to customize the data sources available to the user, or the types of fields that can be added to the report. (For example, you may want to use custom data source objects defined by your application).
- You may want to provide a menu of stock report definitions that makes sense in the scope of your application and allows users to customize some aspects of each stock report. (For example, see the printing options in Microsoft Outlook).
- You may want to write a better, more powerful report designer application than the one you use now, which makes it easier to do things that are important to you or to your co-workers. (For example, add groups of fields to the report).

## What's New in C1ReportDesigner

There were no new features added to C1ReportDesigner in this release. For new features in **ComponentOne Reports for WinForms**, see What's New in Reports for WinForms.

> **Tip:** A version history containing a list of new features, improvements, fixes, and changes for each product is available on the ComponentOne Web site at http://helpcentral.componentone.com/VersionHistory.aspx.

## Installing Reports for .NET Designer Edition

The following sections provide helpful information on installing **ComponentOne Reports for .NET Designer Edition**.

## Reports for .NET Designer Edition Setup Files

The **ComponentOne Reports for .NET Designer Edition** installation program will create the following directory: **C:\Program Files\ComponentOne\Reports Designer Edition**. This directory contains the following subdirectories:

| | |
|---|---|
| **bin** | Contains copies of all ComponentOne binaries (DLLs, EXEs). |
| **C1Report** | Contains files for **Reports for .NET Designer Edition**. |
| **H2Help** | Contains Microsoft Help 2.0 integrated documentation for all Studio components. |
| **HelpViewer** | Contains Microsoft Help Viewer Visual Studio 2010 integrated documentation for all Studio components. |

### Samples

Samples for the product are installed in the **ComponentOne Samples** folder by default. The path of the **ComponentOne Samples** directory is slightly different on Windows XP and Windows 7/Vista machines:

**Windows XP path:** C:\Documents and Settings\<username>\My Documents\ComponentOne Samples

**Windows 7/Vista path:** C:\Users\<username>\Documents\ComponentOne Samples

The **ComponentOne Samples** folder contains the following subdirectories:

| | |
|---|---|
| **Common** | Contains support and data files that are used by many of the demo programs. |
| **C1Report\C1Report** | Contains reporting samples and tutorials for **Reports for .NET Designer Edition**. |
| **C1Report\C1Preview** | Contains previewing samples and tutorials for **Reports for .NET Designer Edition**. |

Samples can be accessed from the **ComponentOne Sample Explorer**. To view samples, on your desktop, click the **Start** button and then click **ComponentOne | Reports Designer Edition | Samples | C1Report Samples** or **C1Preview Samples**.

## Reports for .NET Designer Edition Components

**ComponentOne Reports for .NET Designer Edition** is developed and published by ComponentOne LLC. You may use it to develop applications in conjunction with Microsoft Visual Studio or any other programming environment that enables the user to use and integrate the control(s).

The following table lists the tools comprising **ComponentOne Reports for .NET Designer Edition**, including: their components that can be added to the Visual Studio Toolbox, their namespaces, the assemblies to which they belong, and the ComponentOne redistributable files.

| ComponentOne Tool | Component | Namespace | *Redistributable File/Assembly |
|---|---|---|---|
| Reports for WinForms | C1Report | C1.C1Report | C1.C1Report.2.dll |
| | C1PrintDocument | C1.Preview | C1.Win.C1Report.2.dll |

| | | | |
|---|---|---|---|
| | C1PreviewPane<br>C1PreviewTextSearchPanel<br>C1PreviewThumbnailView<br>C1PrintPreviewControl<br>C1PrintPreviewDialog<br>C1PreviewOutlineView | C1.Win.C1Report | |
| Report Designer Control | C1ReportDesigner | C1.Win.C1ReportDesigner | C1.Win.C1ReportDesigner.2.dll |

The **C1ReportDesigner** and **C1ReportsScheduler** stand-alone applications are also included with **Reports for .NET Designer Edition**. Note that if you purchased **ComponentOne Reports for .NET Designer Edition** you may also redistribute the **C1Report Designer** application (so that your end users can create their own reports).

Site licenses are available for groups of multiple developers. Please contact Sales@ComponentOne.com for details.

* You may distribute, free of royalties, the redistributable files with any such application you develop to the extent that they are used separately on a single CPU on the client/workstation side of the network.

## System Requirements

System requirements include the following:

| | |
|---|---|
| **Operating Systems:** | Windows 2000 |
| | Windows Server® 2003 |
| | Windows Server 2008 |
| | Windows XP SP2 |
| | Windows Vista™ |
| | Windows 7 |
| **Environments:** | .NET Framework 2.0 or later |
| | C# .NET |
| | Visual Basic® .NET |
| **Disc Drive:** | CD or DVD-ROM drive if installing from CD |

## Installing Demonstration Versions

If you wish to try **Reports for .NET Designer Edition** and do not have a serial number, follow the steps through the installation wizard and use the default serial number.

The only difference between unregistered (demonstration) and registered (purchased) versions of our products is that registered versions will stamp every application you compile so a ComponentOne banner will not appear when your users run the applications.

## Uninstalling Reports for .NET Designer Edition

To uninstall **Reports for .NET Designer Edition**:

1. Open **Control Panel** and select **Add or Remove Programs** (**Programs and Features** in Windows 7/Vista).
2. Select **ComponentOne Reports for .NET Designer Edition** and click the **Remove** button.

3. Click **Yes** to remove the program.

# End-User License Agreement

All of the ComponentOne licensing information, including the ComponentOne end-user license agreements, frequently asked licensing questions, and the ComponentOne licensing model, is available online at http://www.componentone.com/SuperPages/Licensing/.

# Licensing FAQs

This section describes the main technical aspects of licensing. It may help the user to understand and resolve licensing problems he may experience when using ComponentOne WinForms and ASP.NET products.

## What is Licensing?

Licensing is a mechanism used to protect intellectual property by ensuring that users are authorized to use software products.

Licensing is not only used to prevent illegal distribution of software products. Many software vendors, including ComponentOne, use licensing to allow potential users to test products before they decide to purchase them.

Without licensing, this type of distribution would not be practical for the vendor or convenient for the user. Vendors would either have to distribute evaluation software with limited functionality, or shift the burden of managing software licenses to customers, who could easily forget that the software being used is an evaluation version and has not been purchased.

## How does Licensing Work?

ComponentOne uses a licensing model based on the standard set by Microsoft, which works with all types of components.

> **Note:** The **Compact Framework** components use a slightly different mechanism for run-time licensing than the other ComponentOne components due to platform differences.

When a user decides to purchase a product, he receives an installation program and a Serial Number. During the installation process, the user is prompted for the serial number that is saved on the system. (Users can also enter the serial number by clicking the **License** button on the **About Box** of any ComponentOne product, if available, or by rerunning the installation and entering the serial number in the licensing dialog box.)

When a licensed component is added to a form or Web page, Visual Studio obtains version and licensing information from the newly created component. When queried by Visual Studio, the component looks for licensing information stored in the system and generates a run-time license and version information, which Visual Studio saves in the following two files:

- An assembly resource file which contains the actual run-time license
- A "licenses.licx" file that contains the licensed component strong name and version information

These files are automatically added to the project.

In WinForms and ASP.NET 1.x applications, the run-time license is stored as an embedded resource in the assembly hosting the component or control by Visual Studio. In ASP.NET 2.x applications, the run-time license may also be stored as an embedded resource in the App_Licenses.dll assembly, which is used to store all run-time licenses for all components directly hosted by WebForms in the application. Thus, the App_licenses.dll must always be deployed with the application.

The licenses.licx file is a simple text file that contains strong names and version information for each of the licensed components used in the application. Whenever Visual Studio is called upon to rebuild the application resources, this file is read and used as a list of components to query for run-time licenses to be embedded in the appropriate assembly resource. Note that editing or adding an appropriate line to this file can force Visual Studio to add run-time licenses of other controls as well.

Note that the licenses.licx file is usually not shown in the Solution Explorer; it appears if you press the **Show All Files** button in the Solution Explorer's Toolbox, or from Visual Studio's main menu, select **Show All Files** on the **Project** menu.

Later, when the component is created at run time, it obtains the run-time license from the appropriate assembly resource that was created at design time and can decide whether to simply accept the run-time license, to throw an exception and fail altogether, or to display some information reminding the user that the software has not been licensed.

All ComponentOne products are designed to display licensing information if the product is not licensed. None will throw licensing exceptions and prevent applications from running.

## Common Scenarios

The following topics describe some of the licensing scenarios you may encounter.

### *Creating components at design time*

This is the most common scenario and also the simplest: the user adds one or more controls to the form, the licensing information is stored in the licenses.licx file, and the component works.

Note that the mechanism is exactly the same for Windows Forms and Web Forms (ASP.NET) projects.

### *Creating components at run time*

This is also a fairly common scenario. You do not need an instance of the component on the form, but would like to create one or more instances at run time.

In this case, the project will not contain a licenses.licx file (or the file will not contain an appropriate run-time license for the component) and therefore licensing will fail.

To fix this problem, add an instance of the component to a form in the project. This will create the licenses.licx file and things will then work as expected. (The component can be removed from the form after the licenses.licx file has been created).

Adding an instance of the component to a form, then removing that component, is just a simple way of adding a line with the component strong name to the licenses.licx file. If desired, you can do this manually using notepad or Visual Studio itself by opening the file and adding the text. When Visual Studio recreates the application resources, the component will be queried and its run-time license added to the appropriate assembly resource.

### *Inheriting from licensed components*

If a component that inherits from a licensed component is created, the licensing information to be stored in the form is still needed. This can be done in two ways:

- Add a LicenseProvider attribute to the component.

  This will mark the derived component class as licensed. When the component is added to a form, Visual Studio will create and manage the licenses.licx file, and the base class will handle the licensing process as usual. No additional work is needed. For example:

  ```
  [LicenseProvider(typeof(LicenseProvider))]
  class MyGrid: C1.Win.C1FlexGrid.C1FlexGrid
  {
    // ...
  }
  ```

- Add an instance of the base component to the form.

This will embed the licensing information into the licenses.licx file as in the previous scenario, and the base component will find it and use it. As before, the extra instance can be deleted after the licenses.licx file has been created.

Please note, that C1 licensing will not accept a run-time license for a derived control if the run-time license is embedded in the same assembly as the derived class definition, and the assembly is a DLL. This restriction is necessary to prevent a derived control class assembly from being used in other applications without a design-time license. If you create such an assembly, you will need to take one of the actions previously described create a component at run time.

### *Using licensed components in console applications*

When building console applications, there are no forms to add components to, and therefore Visual Studio won't create a licenses.licx file.

In these cases, create a temporary Windows Forms application and add all the desired licensed components to a form. Then close the Windows Forms application and copy the licenses.licx file into the console application project.

Make sure the licenses.licx file is configured as an embedded resource. To do this, right-click the licenses.licx file in the Solution Explorer window and select **Properties**. In the Properties window, set the **Build Action** property to **Embedded Resource**.

### *Using licensed components in Visual C++ applications*

There is an issue in VC++ 2003 where the licenses.licx is ignored during the build process; therefore, the licensing information is not included in VC++ applications.

To fix this problem, extra steps must be taken to compile the licensing resources and link them to the project. Note the following:

1. Build the C++ project as usual. This should create an EXE file and also a licenses.licx file with licensing information in it.

2. Copy the licenses.licx file from the application directory to the target folder (**Debug** or **Release**).

3. Copy the C1Lc.exe utility and the licensed DLLs to the target folder. (Don't use the standard lc.exe, it has bugs.)

4. Use C1Lc.exe to compile the licenses.licx file. The command line should look like this:
   ```
   c1lc /target:MyApp.exe /complist:licenses.licx /i:C1.Win.C1FlexGrid.dll
   ```

5. Link the licenses into the project. To do this, go back to Visual Studio, right-click the project, select **Properties**, and go to the **Linker/Command Line** option. Enter the following:
   ```
   /ASSEMBLYRESOURCE:Debug\MyApp.exe.licenses
   ```

6. Rebuild the executable to include the licensing information in the application.

### *Using licensed components with automated testing products*

Automated testing products that load assemblies dynamically may cause them to display license dialog boxes. This is the expected behavior since the test application typically does not contain the necessary licensing information, and there is no easy way to add it.

This can be avoided by adding the string "C1CheckForDesignLicenseAtRuntime" to the AssemblyConfiguration attribute of the assembly that contains or derives from ComponentOne controls. This attribute value directs the ComponentOne controls to use design-time licenses at run time.

For example:

```
   #if AUTOMATED_TESTING
[AssemblyConfiguration("C1CheckForDesignLicenseAtRuntime")]
   #endif
```

```
public class MyDerivedControl : C1LicensedControl
{
      // ...
}
```

Note that the AssemblyConfiguration string may contain additional text before or after the given string, so the AssemblyConfiguration attribute can be used for other purposes as well. For example:

```
[AssemblyConfiguration("C1CheckForDesignLicenseAtRuntime,BetaVersion")]
```

THIS METHOD SHOULD ONLY BE USED UNDER THE SCENARIO DESCRIBED. It requires a design-time license to be installed on the testing machine. Distributing or installing the license on other computers is a violation of the EULA.

## Troubleshooting

We try very hard to make the licensing mechanism as unobtrusive as possible, but problems may occur for a number of reasons.

Below is a description of the most common problems and their solutions.

### *I have a licensed version of a ComponentOne product but I still get the splash screen when I run my project.*

If this happens, there may be a problem with the licenses.licx file in the project. It either doesn't exist, contains wrong information, or is not configured correctly.

First, try a full rebuild (**Rebuild All** from the Visual Studio **Build** menu). This will usually rebuild the correct licensing resources.

**If that fails follow these steps:**

1. Open the project and go to the Solution Explorer window.
2. Click the **Show All Files** button on the top of the window.
3. Find the licenses.licx file and open it. If prompted, continue to open the file.
4. Change the version number of each component to the appropriate value. If the component does not appear in the file, obtain the appropriate data from another licenses.licx file or follow the alternate procedure following.
5. Save the file, and then close the licenses.licx tab.
6. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

**Alternatively, follow these steps:**

1. Open the project and go to the Solution Explorer window.
2. Click the **Show All Files** button on the top of the window.
3. Find the **licenses.licx** file and delete it.
4. Close the project and reopen it.
5. Open the main form and add an instance of each licensed control.
6. Check the Solution Explorer window, there should be a licenses.licx file there.
7. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

**For ASP.NET 2.x applications, follow these steps:**

1. Open the project and go to the Solution Explorer window.
2. Find the **licenses.licx** file and right-click it.

3. Select the **Rebuild Licenses** option (this will rebuild the App_Licenses.licx file).
4. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

### *I have a licensed version of a ComponentOne product on my Web server but the components still behave as unlicensed.*

There is no need to install any licenses on machines used as servers and not used for development.

The components must be licensed on the development machine, therefore the licensing information will be saved into the executable (.exe or .dll) when the project is built. After that, the application can be deployed on any machine, including Web servers.

For ASP.NET 2.x applications, be sure that the App_Licenses.dll assembly created during development of the application is deployed to the bin application bin directory on the Web server.

If your ASP.NET application uses WinForms user controls with constituent licensed controls, the run-time license is embedded in the WinForms user control assembly. In this case, you must be sure to rebuild and update the user control whenever the licensed embedded controls are updated.

### *I downloaded a new build of a component that I have purchased, and now I'm getting the splash screen when I build my projects.*

Make sure that the serial number is still valid. If you licensed the component over a year ago, your subscription may have expired. In this case, you have two options:

**Option 1 – Renew your subscription to get a new serial number.**

If you choose this option, you will receive a new serial number that you can use to license the new components (from the installation utility or directly from the **About Box**).

The new subscription will entitle you to a full year of upgrades and to download the latest maintenance builds directly from http://prerelease.componentone.com/.

**Option 2 – Continue to use the components you have.**

Subscriptions expire, products do not. You can continue to use the components you received or downloaded while your subscription was valid.

## Technical Support

ComponentOne offers various support options. For a complete list and a description of each, visit the ComponentOne Web site at http://www.componentone.com/SuperProducts/SupportServices/.

Some methods for obtaining technical support include:

- Online Resources
  ComponentOne provides customers with a comprehensive set of technical resources in the form of FAQs, samples and videos, Version Release History, searchable Knowledge base, searchable Online Help and more. We recommend this as the first place to look for answers to your technical questions.

- **Online Support via our Incident Submission Form**
  This online support service provides you with direct access to our Technical Support staff via an online incident submission form. When you submit an incident, you'll immediately receive a response via e-mail confirming that you've successfully created an incident. This email will provide you with an Issue Reference ID and will provide you with a set of possible answers to your question from our Knowledgebase. You will receive a response from one of the ComponentOne staff members via e-mail in 2 business days or less.

- **Peer-to-Peer Product Forums**
  ComponentOne peer-to-peer product forums are available to exchange information, tips, and techniques regarding ComponentOne products. ComponentOne sponsors these areas as a forum for users to share information. While ComponentOne does not provide direct support in the forums and newsgroups, we

periodically monitor them to ensure accuracy of information and provide comments when appropriate. Please note that a ComponentOne User Account is required to participate in the ComponentOne Product Forums.

- **Installation Issues**
  Registered users can obtain help with problems installing ComponentOne products. Contact technical support by using the online incident submission form or by phone (412.681.4738). Please note that this does not include issues related to distributing a product to end-users in an application.

- **Documentation**
  Microsoft integrated ComponentOne documentation can be installed with each of our products, and documentation is also available online. If you have suggestions on how we can improve our documentation, please email the Documentation team. Please note that e-mail sent to the Documentation team is for documentation feedback only. Technical Support and Sales issues should be sent directly to their respective departments.

**Note:** You must create a ComponentOne Account and register your product with a valid serial number to obtain support using some of the above methods.

# About This Documentation

### Acknowledgements

*Microsoft, Visual Studio, Visual Basic, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.*

### ComponentOne

If you have any suggestions or ideas for new features or controls, please call us or write:

*Corporate Headquarters*

**ComponentOne LLC**
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 • USA
412.681.4343
412.681.4384 (Fax)

http://www.componentone.com/

### ComponentOne Doc-To-Help

This documentation was produced using ComponentOne Doc-To-Help® Enterprise.

# Namespaces

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

The general namespace for ComponentOne Windows products is **C1.Win**. The namespace for the C1ReportDesigner component is **C1.Win.C1ReportDesigner**. The following code fragment shows how to declare a C1ReportDesigner component using the fully qualified name for this class:

- Visual Basic
  ```
  Dim c1rpt As C1.Win.C1ReportDesigner.C1ReportDesigner
  ```

- C#
  ```
  C1.Win.C1ReportDesigner.C1ReportDesigner c1rpt;
  ```

Namespaces address a problem sometimes known as *namespace pollution*, in which the developer of a class library is hampered by the use of similar names in another library. These conflicts with existing components are sometimes called *name collisions*.

For example, if you create a new class named ClipboardHandler, you can use it inside your project without qualification. However, the C1ReportDesigner assembly also implements a class called ClipboardHandler. So, if you want to use the C1ReportDesigner class in the same project, you must use a fully qualified reference to make the reference unique. If the reference is not unique, Visual Studio .NET produces an error stating that the name is ambiguous. The following code snippet demonstrates how to declare these objects:

- Visual Basic

```
' Define a new ClipboardHandler object (custom ClipboardHandler class)
Dim MyClipboard as ClipboardHandler

' Define a new C1ReportDesigner.ClipboardHandler object
Dim ReportDesignerClip as _
      C1.Win.C1ReportDesigner.ClipboardHandler
```

- C#

```
// Define a new ClipboardHandler object (custom ClipboardHandler class)
MyClipboard ClipboardHandler;

// Define a new C1ReportDesigner.ClipboardHandler object
ReportDesignerClip C1.Win.C1ReportDesigner.ClipboardHandler;
```

Fully qualified names are object references that are prefixed with the name of the namespace where the object is defined. You can use objects defined in other projects if you create a reference to the class (by choosing **Add Reference** from the **Project** menu) and then use the fully qualified name for the object in your code.

Fully qualified names prevent naming conflicts because the compiler can always determine which object is being used. However, the names themselves can get long and cumbersome. To get around this, you can use the **Imports** statement (**using** in C#) to define an alias – an abbreviated name you can use in place of a fully qualified name. For example, the following code snippet creates aliases for two fully qualified names, and uses these aliases to define two objects:

- Visual Basic

```
Imports C1ClipboardHandler = C1.Win.C1ReportDesigner.ClipboardHandler
Imports MyClipboard = MyProject.ClipboardHandler

Dim c1 As C1ClipboardHandler
Dim c2 As MyClipboard
```

- C#

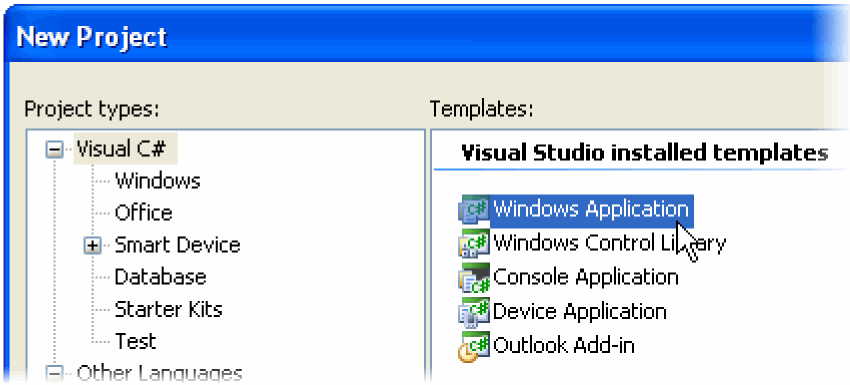
```
using C1ClipboardHandler = C1.Win.C1ReportDesigner.ClipboardHandler;
using MyClipboard = MyProject.ClipboardHandler;

C1ClipboardHandler c1;
MyClipboard c2;
```

If you use the **Imports** statement without an alias, you can use all the names in that namespace without qualification provided they are unique to the project.

## Creating a .NET Project

To create a new .NET project, complete the following steps:

1. From the **File** menu in Microsoft Visual Studio, select **New Project**. The **New Project** dialog box opens.
2. Under **Project Types**, choose either **Visual Basic** or **Visual C#**, and select **Windows Application** from the list of **Templates** in the right pane.



3. Enter a name for your application in the **Name** field and click **OK**.

   A new Microsoft Visual Studio .NET project is created and a new Form1 is displayed in the Designer view.

4. Double-click the desired C1ReportDesigner components from the Toolbox to add them to Form1. For information on adding a component to the Toolbox, see Adding the C1ReportDesigner Component to a Project.

## Adding the C1ReportDesigner Component to a Project

### Manually Adding C1ReportDesigner to the Toolbox

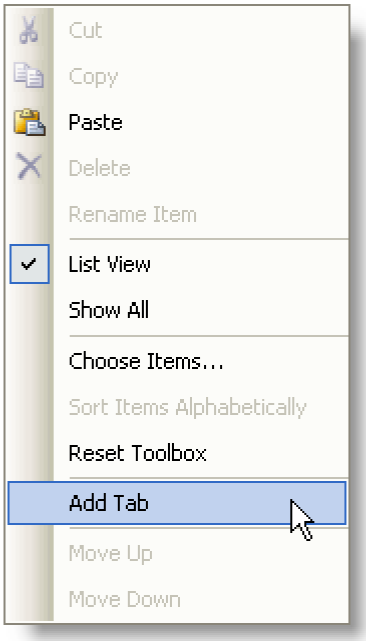
When you install **ComponentOne Reports for .NET Designer Edition**, the following **Reports for .NET Designer Edition** component will appear in the Visual Studio Toolbox customization dialog box:

- C1ReportDesigner

To manually add the C1ReportDesigner control to the Visual Studio Toolbox:

1. Open the Visual Studio IDE (Microsoft Development Environment). Make sure the Toolbox is visible (select **Toolbox** in the **View** menu if necessary) and right-click it to open the context menu.

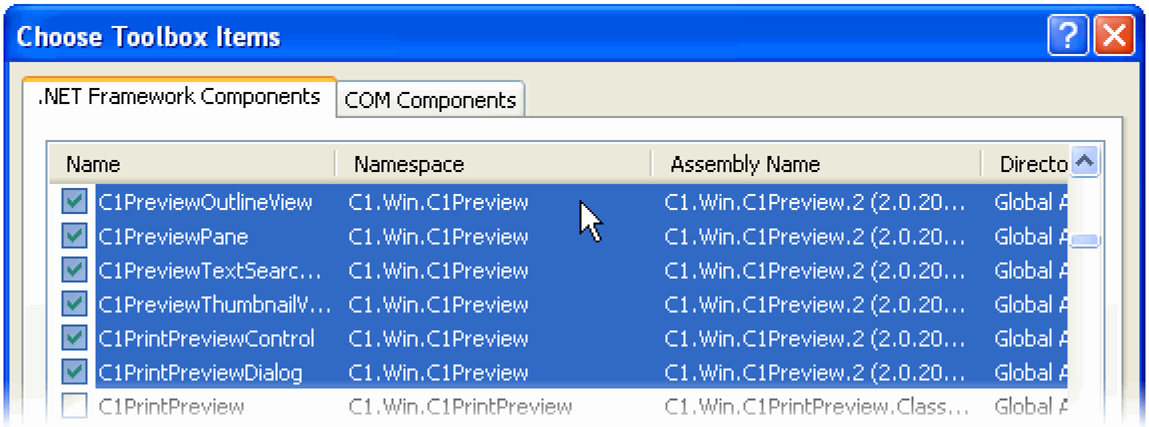
2. To make the C1ReportDesigner component appear on its own tab in the Toolbox, select **Add Tab** from the context menu and type in the tab name, **C1ReportDesigner**, for example.



3. Right-click the tab where the component is to appear and select **Choose Items** from the context menu.

   The **Choose Toolbox Items** dialog box opens.

4. In the dialog box, select the **.NET Framework Components** tab. Sort the list by Namespace (click the Namespace column header) and check the check boxes for all components belonging to namespace C1.Win.C1ReportDesigner. Note that there may be more than one component for each namespace.



| Name | Namespace | Assembly Name | Directo |
|---|---|---|---|
| ☑ C1PreviewOutlineView | C1.Win.C1Preview | C1.Win.C1Preview.2 (2.0.20... | Global A |
| ☑ C1PreviewPane | C1.Win.C1Preview | C1.Win.C1Preview.2 (2.0.20... | Global A |
| ☑ C1PreviewTextSearc... | C1.Win.C1Preview | C1.Win.C1Preview.2 (2.0.20... | Global A |
| ☑ C1PreviewThumbnailV... | C1.Win.C1Preview | C1.Win.C1Preview.2 (2.0.20... | Global A |
| ☑ C1PrintPreviewControl | C1.Win.C1Preview | C1.Win.C1Preview.2 (2.0.20... | Global A |
| ☑ C1PrintPreviewDialog | C1.Win.C1Preview | C1.Win.C1Preview.2 (2.0.20... | Global A |
| □ C1PrintPreview | C1.Win.C1PrintPreview | C1.Win.C1PrintPreview.Class... | Global A |

### Adding Reports for .NET Designer Edition to the Form

To add **Reports for .NET Designer Edition** to a form:

1. Add the C1ReportDesigner control to the Visual Studio Toolbox.
2. Double-click the control or drag it onto your form.

### Adding a Reference to the Assembly

To add a reference to the **Reports for .NET Designer Edition** assembly:

1. Select the **Add Reference** option from the **Project** menu of your project.

2. Select the **ComponentOne C1ReportDesigner** assembly from the list on the **.NET** tab or browse to find the C1.Win.C1ReportDesigner.2.dll file and click **OK**.

3. Double-click the form caption area to open the code window. At the top of the file, add the following **Imports** statements (**using** in C#):

```
Imports C1.Win.C1ReportDesigner
```

> **Note:** This makes the objects defined in the **C1ReportDesigner** assembly visible to the project. See Namespaces for more information.

## Migrating a C1ReportDesigner Project to Visual Studio 2005

To migrate a project using ComponentOne components to Visual Studio 2005, there are two main steps that must be performed. First, you must convert your project to Visual Studio 2005, which includes removing any references to a previous assembly and adding a reference to the new assembly. Secondly, the .licx file, or licensing file, must be updated in order for the project to run correctly.

**To convert the project:**

1. Open Visual Studio 2005 and select **File | Open | Project/Solution** .
2. Locate the **.sln** file for the project that you wish to convert to Visual Studio 2005. Select it and click **Open**. The **Visual Studio Conversion Wizard** appears.
3. Click **Next**.
4. Select **Yes, create a backup before converting** to create a backup of your current project and click **Next**.
5. Click **Finish** to convert your project to Visual Studio 2005. The **Conversion Complete** window appears.
6. Click **Show the conversion log when the wizard is closed** if you want to view the conversion log.
7. Click **Close**. The project opens. Now you must remove references to any of the previous ComponentOne .dlls and add references to the new ones.
8. Go to the Solution Explorer (**View | Solution Explorer**) and click the **Show All Files** button.

   > **Note:** The **Show All Files** button does not appear in the Solution Explorer toolbar if the Solution project node is selected.

9. Expand the **References** node, right-click C1.Common and select **Remove**. Also remove C1.Win.C1Report the same way.
10. Right-click the **References** node and select **Add Reference**.
11. Locate and select **C1.Win.C1Report.2.dll**. Click **OK** to add it to the project.

**To update the .licx file:**

1. In the Solution Explorer, right-click the **licenses.licx** file and select **Delete**.
2. Click **OK** to permanently delete **licenses.licx**. The project must be rebuilt to create a new, updated version of the .licx file.
3. Click the **Start Debugging** button to compile and run the project. The new .licx file may not be visible in the Solution Explorer.
4. Select **File, Close** to close the form and then double-click the **Form.vb** or **Form.cs** file in the Solution Explorer to reopen it. The new **licenses.licx** file appears in the list of files.

   The migration process is complete.

# Key Features

The C1ReportDesigner control includes several key features, such as:

- **Closely Integrate the Designer Into Your Application**

  Writing your own customized report designer allows you to integrate the designer tightly into your application, rather than running a separate application. You can customize the data sources available to the end-user, or the types of fields that can be added to the report. For example, you may want to use custom data source objects defined by your application.

- **Supply Custom Stock Report Definitions to End-users**

  With your own customized report designer you can provide a menu of stock report definitions that make sense in the scope of your application. It allows end-users to customize some aspects of each stock report, similar to the printing options in Microsoft Outlook.

- **Royalty-free Run-time Distribution**

  Developers can now enjoy royalty-free run-time distribution of the **C1ReportDesigner** application for flexible application deployment to an unlimited number of clients. Developers can easily create, customize, and deploy powerful and integrated reporting solutions including end-user report designers, and distribute them without royalty-fee restrictions.

- **Full Source Code for the C1ReportDesigner Application**

  The designer edition ships with full source code for the **C1ReportDesigner** application included in **ComponentOne Reports™ for .WinForms Designer Edition**. Full source code allows developers to customize the designer application or even integrate it within their own applications.

- **Easy-to-use C1ReportDesigner Component**

  The **C1ReportDesigner** component displays reports in design mode, and allows users to drag, copy, and resize report fields and sections. The component also provides an unlimited undo/redo stack and a selection mechanism designed for use with the **PropertyGrid** control that ships with Microsoft Visual Studio .NET.

- **Access to ComponentOne's Powerful Reporting Object Model**

  **ComponentOne Reports for WinForms** adds powerful, flexible database reporting to your applications. Now you can quickly and easily create Microsoft Access-style database reports for your Visual Studio .NET applications. **ComponentOne Reports for WinForms** includes the **C1Report** component, which generates Access-style database reports, and its companion **C1ReportDesigner** application. **C1ReportDesigner** is a stand-alone application that enables you to create, edit, load, and save report definition files that can be read directly by the **C1Report** component. **ComponentOne Reports for WinForms** also include the **C1ReportsScheduler** application to help with report scheduling.

- **Access to ComponentOne's Powerful Previewing Object Model**

  **ComponentOne Reports for WinForms'** rich object model also adds robust previewing, formatting, printing, and exporting to your applications. **Reports for WinForms** includes two main previewing components: **C1PrintDocument** (a powerful document generator) and **C1PrintPreviewControl** (a full-featured print preview component). Together, they handle all of your previewing, formatting, printing, and exporting needs.

# Using the C1ReportDesigner Control

To use the C1ReportDesigner control, simply add it to a form, add a **C1Report** component that will contain the report you want to edit, and set the Report property in the designer control.

When you run the project, you will see the report definition in design mode. You will be able to select, move, and resize the report fields and sections. Any changes made through the designer will be reflected in the report definition stored in the **C1Report** component. You can save the report at any time using the **C1Report.Save** method, or preview it using the **C1Report.Document** property and a **Preview** control.

To build a complete designer, you will have to add other user interface elements:

- A **PropertyGrid** control attached to the designer selection, so the user can change field and section properties.
- A **DataSource** selection mechanism so the user can edit and change the report data source.
- A **Group Editor** dialog box if you want to allow the user to create, remove, and edit report groups.
- A **Wizard** to create new reports.
- The usual file and editing commands so users can load and save reports, use the clipboard, and access the undo/redo mechanism built into the C1ReportDesigner control.

Most of these elements are optional and may be omitted depending on your needs. The Report Designer application source code implements all of these, and you can use the source code as a basis for your implementation.

### About this section

This section describes how to implement a simple report designer using the C1ReportDesigner control. The purpose of the sample designer is to illustrate how the C1ReportDesigner control integrates with a designer application. It supports loading and saving files with multiple reports, editing and previewing reports, adding and removing reports from the file, and report editing with undo/redo and clipboard support.

Most designer applications based on the C1ReportDesigner control will have features similar to the ones described here. If you follow these steps, you will become familiar with all the basic features of the C1ReportDesigner control.

The sample designer does not provide some advanced capabilities such as import/export, data source selection/editing, and group editing. All these features are supported by the full version of the **C1ReportDesigner** application, and you can refer to the source code for details on how to implement them.

The following sections describe the step-by-step implementation of the sample designer.

**Sample Report Available**

For the complete project, see the **SimpleDesigner** sample, which is available for download from the ComponentOne HelpCentral Sample page.

## Step 1 of 9: Create and Populate the Main Form

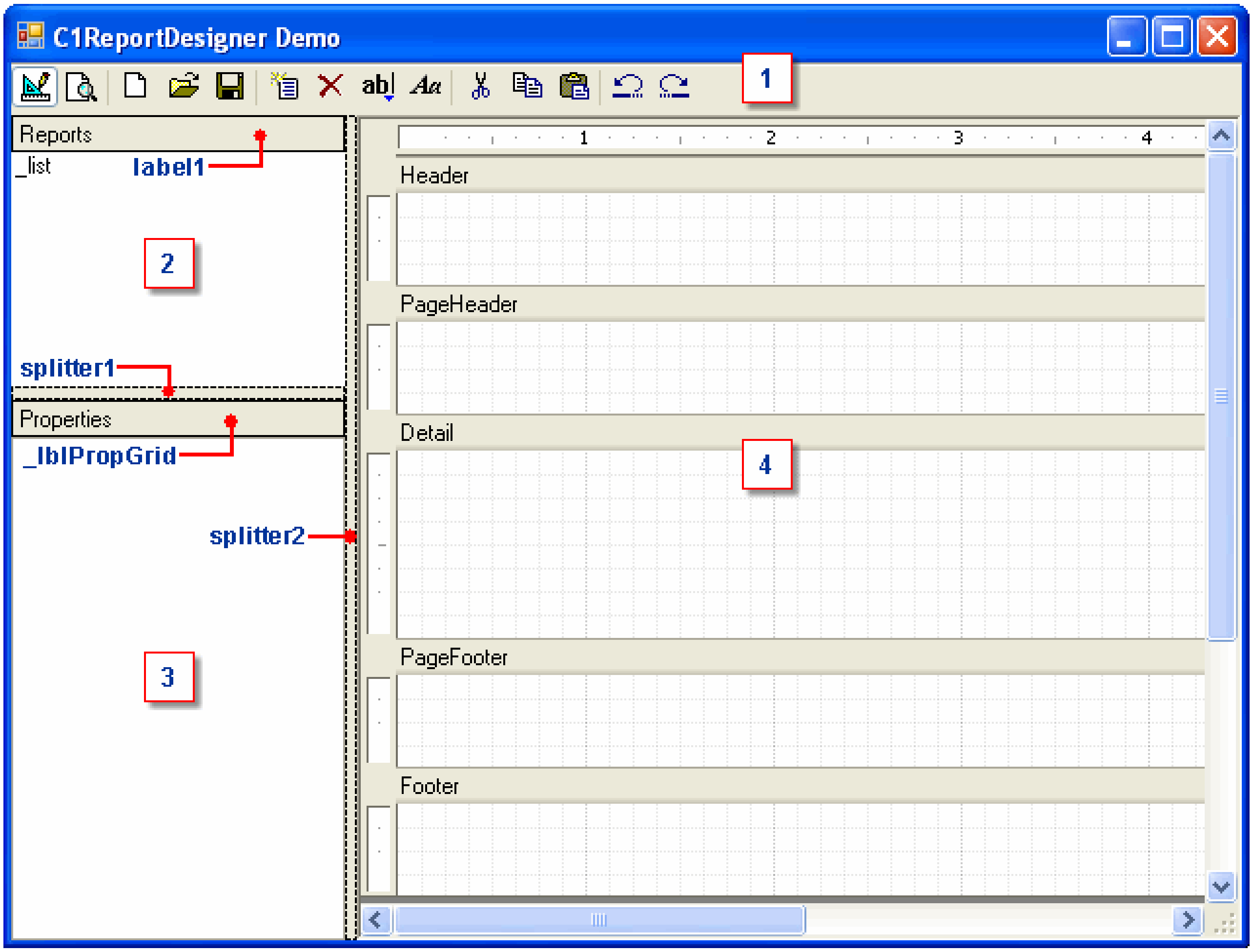
The sample designer consists of a single form with the following main components:

| Control | Control Name | Description |
|---|---|---|
| **ListBox** | **_list** | **ListBox** control with a list of reports currently loaded. |
| **C1PrintPreview** | **_c1ppv** | **C1PrintPreview** control for previewing the reports. |
| **C1ReportDesigner** | **_c1rd** | **C1ReportDesigner** control for designing and editing reports. |
| **PropertyGrid** | **_ppg** | **PropertyGrid** control for editing properties of objects selected in the designer. |
| **ToolBar** | **_tb** | **ToolBar** control with buttons for each command. |
| **C1Report** | **_c1r** | **C1Report** component used for rendering reports into the **_c1ppv** |

| | | |
|---|---|---|
| | | control. |

For steps on adding the C1ReportDesigner control to the form, see [Adding the C1ReportDesigner Component to a Project](#).

The form contains a few other controls such as labels and splitters, which are used to improve the layout. Notice that the controls are numbered and the labels and splitters are named in the image below. Here's what the form should look like:



Refer to the labels on the form to locate the following controls:

- 1 The **ToolBar** control **_tb** appears along the top of the form.
- 2 The report list **_list** appears to the left above the property grid **_ppg**.
- 3 The property grid **_ppg**.
- 4 On the right, the C1ReportDesigner control **_c1rd** fills the client area of the form.

- The preview control **_c1ppv** is invisible in design mode. In preview mode, it becomes visible and hides the report designer.

In this sample designer, the toolbar contains 18 items (14 buttons and 4 separators). If you are creating the project from scratch, don't worry about the images at this point. Just add the items to the **_tb** control (which can easily be done using the **ToolBarButton Collection Editor**) and set their names as each command is implemented.

## Step 2 of 9: Add Class Variables and Constants

In this step, add the following code to your simple designer project to add class variables and constants:

- Visual Basic

```
' fields
Private _fileName As String  ' name of the current file
Private _dirty As Boolean    ' current file has changed

' title to display in the form caption
Dim _appName As String = "C1ReportDesigner Demo"
```

- C#

```
// fields
private string  _fileName;     // name of the current file
private bool    _dirty;        // current file has changed

// title to display in the form caption
private const string _appName = "C1ReportDesigner Demo";
```

## Step 3 of 9: Add Code to Update the User Interface

The simple designer has buttons that may be enabled or disabled, depending on whether the clipboard and undo buffer are empty, whether a file is loaded, and so on. All this functionality is implemented in a single method, called **UpdateUI**.

**UpdateUI** is called often to make sure the UI reflects the state of the application. The first call should be made in response to the **Form_Load** event, to initialize the toolbar and form caption. After pasting the following code into the project, remember to set the names of the buttons in the toolbar control to match the ones used in the **UpdateUI** routine.

Add the following code to update the user interface:

- Visual Basic

```
' update UI on startup to show form title and disable clipboard and
' undo/redo buttons
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    UpdateUI()
End Sub
Private  Sub UpdateUI()
    ' update caption
    _fileName = _appName
    If _fileName.Length > 0 Then
    _fileName = String.Format("{0} - [{1}]", _appName, _fileName)
        If _dirty Then _fileName = _fileName + " *"
End If

    ' push/release design/preview mode buttons
    Dim design As Boolean =  _c1rd.Visible AndAlso (Not
IsNothing(_c1rd.Report))
```

```
    _btnDesign.Pushed = design
    _btnPreview.Pushed = Not design

    ' enable/disable buttons
    _btnCut.Enabled = design AndAlso _c1rd.ClipboardHandler.CanCut
    _btnCopy.Enabled = design AndAlso _c1rd.ClipboardHandler.CanCut
    _btnPaste.Enabled = design AndAlso _c1rd.ClipboardHandler.CanPaste
    _btnUndo.Enabled = design AndAlso _c1rd.UndoStack.CanUndo
    _btnRedo.Enabled = design AndAlso _c1rd.UndoStack.CanRedo

    Dim reportSelected As Boolean =  design AndAlso Not
(IsNothing(_list.SelectedItem))
    _btnAddReport.Enabled = _c1rd.Visible
    _btnDelReport.Enabled = reportSelected
    _btnAddField.Enabled  = reportSelected
    _btnAddLabel.Enabled  = reportSelected
End Sub
```

- C#

```
// update UI on startup to show form title and disable clipboard and
// undo/redo buttons
private void Form1_Load(object sender, System.EventArgs e)
{
    UpdateUI();
}
private void UpdateUI()
{
    // update caption
    Text = (_fileName != null && _fileName.Length > 0)
    ? string.Format("{0} - [{1}] {2}", _appName, _fileName, _dirty? "*":
"")
    : _appName;

    // push/release design/preview mode buttons
    bool design = _c1rd.Visible && _c1rd.Report != null;
    _btnDesign.Pushed  = design;
    _btnPreview.Pushed = !design;

    // enable/disable buttons
    _btnCut.Enabled   = design && _c1rd.ClipboardHandler.CanCut;
    _btnCopy.Enabled  = design && _c1rd.ClipboardHandler.CanCut;
    _btnPaste.Enabled = design && _c1rd.ClipboardHandler.CanPaste;
    _btnUndo.Enabled  = design && _c1rd.UndoStack.CanUndo;
    _btnRedo.Enabled  = design && _c1rd.UndoStack.CanRedo;

    bool reportSelected = design && _list.SelectedItem != null;
    _btnAddReport.Enabled = _c1rd.Visible;
    _btnDelReport.Enabled = reportSelected;
    _btnAddField.Enabled  = reportSelected;
    _btnAddLabel.Enabled  = reportSelected;
}
```

Notice how **UpdateUI** uses the CanCut, CanPaste, CanUndo, and CanRedo properties to enable and disable toolbar buttons.

## Step 4 of 9: Add Code to Handle the Toolbar Commands

To handle the clicks on the toolbar buttons and dispatch them to the appropriate handlers, use the following code:

- Visual Basic

```
' handle clicks on toolbar buttons
Private Sub _tb_ButtonClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ToolBarButtonClickEventArgs) Handles _tb.ButtonClick

    ' design/preview mode
    If e.Button.Equals(_btnDesign) Then
      SetDesignMode(True)
    End If
    If e.Button.Equals(_btnPreview) Then
      SetDesignMode(False)

    ' file commands
    If e.Button.Equals(_btnNew) Then
      NewFile()
    If e.Button.Equals(_btnOpen) Then
      OpenFile()
    If e.Button.Equals(_btnSave) Then
      SaveFile()

    ' allow user to undo clipboard operations
    If e.Button.Equals(_btnCut) Or e.Button.Equals(_btnPaste) Then
        _c1rd.UndoStack.SaveState()
    End If

    ' clipboard
    If e.Button.Equals(_btnCut) Then
      _c1rd.ClipboardHandler.Cut()
    If e.Button.Equals(_btnCopy) Then
      _c1rd.ClipboardHandler.Copy()
    If e.Button.Equals(_btnPaste) Then
            _c1rd.ClipboardHandler.Paste()

    ' undo/redo
    If e.Button.Equals(_btnUndo) Then
      _c1rd.UndoStack.Undo()
    If e.Button.Equals(_btnRedo) Then
      _c1rd.UndoStack.Redo()

    ' add/remove reports
    If e.Button.Equals(_btnAddReport) Then
      NewReport()
    If e.Button.Equals(_btnDelReport) Then
      DeleteReport()

    ' add fields
    ' (just set create info and wait for CreateField event from designer)
    If e.Button.Equals(_btnAddField) Then
      _c1rd.CreateFieldInfo = e.Button
    End If
    If e.Button.Equals(_btnAddLabel) Then
      _c1rd.CreateFieldInfo = e.Button
    End If
End Sub
```

- C#

```
// handle clicks on toolbar buttons
private void _tb_ButtonClick(object sender,
    System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
    // design/preview mode
    if (e.Button == _btnDesign)     SetDesignMode(true);
    if (e.Button == _btnPreview)    SetDesignMode(false);

    // file commands
    if (e.Button == _btnNew)        NewFile();
    if (e.Button == _btnOpen)       OpenFile();
    if (e.Button == _btnSave)       SaveFile();

    // allow user to undo clipboard operations
    if (e.Button == _btnCut || e.Button == _btnPaste)
        _c1rd.UndoStack.SaveState();

    // clipboard
    if (e.Button == _btnCut)       _c1rd.ClipboardHandler.Cut();
    if (e.Button == _btnCopy)      _c1rd.ClipboardHandler.Copy();
    if (e.Button == _btnPaste)     _c1rd.ClipboardHandler.Paste();

    // undo/redo
    if (e.Button == _btnUndo)      _c1rd.UndoStack.Undo();
    if (e.Button == _btnRedo)      _c1rd.UndoStack.Redo();

    // add/remove reports
    if (e.Button == _btnAddReport)   NewReport();
    if (e.Button == _btnDelReport)   DeleteReport();

    // add fields
    // (just set create info and wait for CreateField event from designer)
    if (e.Button == _btnAddField)    _c1rd.CreateFieldInfo = e.Button;
    if (e.Button == _btnAddLabel)    _c1rd.CreateFieldInfo = e.Button;
}
```

This routine dispatches about half of the commands to specialized handlers. These will be described later. The other half (clipboard, undo/redo) is handled directly by the C1ReportDesigner control.

Note that before calling the Cut and Paste methods, the code calls the SaveState method to save the current state of the report. This allows the user to undo and redo clipboard operations. In general, your code should always call SaveState before making changes to the report.

## Step 5 of 9: Implement the SetDesignMode Method

The simple designer has two modes: report design and preview. When the user selects a new report or clicks the **Design** button on the toolbar, the application shows the designer control. When the user clicks the **Preview** button, the application renders the current report into the preview control and shows the result.

Add the following code to implement the **SetDesignMode** method:

- Visual Basic

```
Private Sub SetDesignMode(ByVal design As Boolean)
    ' show/hide preview/design panes
    _c1rd.Visible  = design
    _c1ppv.Visible = Not design

    ' no properties in preview mode
    If Not design Then
        _lblPropGrid.Text = "Properties"
        _ppg.SelectedObject = Nothing
    End If

    ' attach copy of the report to preview control
    ' (so changes caused by script aren't saved)
    If Not design Then
        _c1ppv.Document = Nothing
        _c1r.CopyFrom(_c1rd.Report)
        Cursor = Cursors.WaitCursor
        _c1r.Render()
        If _c1r.PageImages.Count > 0 Then
            _c1ppv.Document = _c1r.Document
        End If
        Cursor = Cursors.Default
    End If

    ' done, update UI
    UpdateUI()
End Sub
```

- C#

```
private void SetDesignMode( bool design)
{
    // show/hide preview/design panes
    _c1rd.Visible  = design;
    _c1ppv.Visible = !design;

    // no properties in preview mode
    if (!design )
    {
        _lblPropGrid.Text = "Properties";
        _ppg.SelectedObject = null;
    }

    // attach copy of the report to preview control
    // (so changes caused by script aren't saved)
    if (!design )
    {
        _c1ppv.Document = null;
        _c1r.CopyFrom(_c1rd.Report);
        Cursor = Cursors.WaitCursor;
        _c1r.Render();
        if (_c1r.PageImages.Count > 0 )
            _c1ppv.Document = _c1r.Document;
        Cursor = Cursors.Default;
    }

    // done, update UI
    UpdateUI();
}
```

Switching to design mode is easy, all you have to do is show the designer and hide the preview control. Switching to preview mode is a little more involved because it also requires rendering the report.

Note that the report is copied to a separate **C1Report** control before being rendered. This is necessary because reports may contain script code that changes the report definition (field colors, visibility, and so on), and we don't want those changes applied to the report definition.

## Step 6 of 9: Implement the File Support Methods

The simple designer has three commands that support files: **New**, **Open**, and **Save**. **NewFile** clears the class variables, report list, preview and designer controls, and then updates the UI.

Add the following code to implement the **NewFile** method:

- Visual Basic

```
Private Sub NewFile()
    _fileName = ""
    _dirty = False
    _list.Items.Clear()
    _c1ppv.Document = Nothing
    _c1rd.Report = Nothing
    UpdateUI()
End Sub
```

- C#

```
private void NewFile()
{
    _fileName = "";
    _dirty = false;
    _list.Items.Clear();
    _c1ppv.Document = null;
    _c1rd.Report = null;
    UpdateUI();
}
```

**OpenFile** prompts the user for a report definition file to open, then uses the **C1Report** component to retrieve a list of report names in the selected file. Each report is loaded into a new **C1Report** component, which is added to the report list (**_list** control).

Instead of adding the **C1Report** components directly to the list box, the code uses a **ReportHolder** wrapper class. The only function of the **ReportHolder** class is to override the **ToString** method so the list box shows the report names.

Add the following code to implement the **OpenFile** method:

- Visual Basic

```
Public Sub OpenFile()
    ' get name of file to open
    Dim dlg As New OpenFileDialog
    dlg.FileName = "*.xml"
    dlg.Title = "Open Report Definition File"
    If dlg.ShowDialog() <> DialogResult.OK Then
        Return
    End If

    ' check selected file
    Try
        reports = _c1r.GetReportInfo(dlg.FileName)
    Catch
        If IsNothing(reports) OrElse reports.Length = 0 Then
            MessageBox.Show("Invalid (or empty) report definition file")
            Return
        End If
    End Try
    ' clear list
    NewFile()

    ' load new file
    Cursor = Cursors.WaitCursor
    _fileName = dlg.FileName
    Dim reportName As String
    For Each reportName In reports
        Dim rpt As New C1Report()
        rpt.Load(_fileName, reportName)
        _list.Items.Add(New ReportHolder(rpt))
    Next
    Cursor = Cursors.Default

  ' select first report
    _list.SelectedIndex = 0
End Sub

' ReportHolder
' Helper class used to store reports in listboxes. The main thing
' it does is override the ToString() method to render the report name.
Public Class ReportHolder
        Public  Sub New(ByVal report As C1Report)
        Me.Report = report
    End Sub
    Public Overrides Function ToString() As String
        Dim text As String = Me.Report.ReportName
        If text.Length = 0 Then text = "Unnamed Report"
        Return text
    End Function
    Public ReadOnly Report As C1Report
End Class
```

- C#

```
public void OpenFile()
{
    // get name of file to open
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.FileName = "*.xml";
    dlg.Title = "Open Report Definition File";
    if (dlg.ShowDialog() != DialogResult.OK)
        return;

    // check selected file
    string[] reports = null;
    try
    {
        reports = _c1r.GetReportInfo(dlg.FileName);
    }
    catch {}
    if (reports == null || reports.Length == 0)
    {
        MessageBox.Show("Invalid (or empty) report definition file");
        return;
    }
    // clear list
    NewFile();

    // load new file
    Cursor = Cursors.WaitCursor;
    _fileName = dlg.FileName;
    foreach (string reportName in reports)
    {
        C1Report rpt = new C1Report();
        rpt.Load(_fileName, reportName);
        _list.Items.Add(new ReportHolder(rpt));
    }
    Cursor = Cursors.Default;

    // select first report
    _list.SelectedIndex = 0;
}
// ReportHolder
// Helper class used to store reports in listboxes. The main thing
// it does is override the ToString() method to render the report name.
public class ReportHolder
{
    public readonly C1Report Report;
    public ReportHolder(C1Report report)
    {
        Report = report;
    }
    override public string ToString()
    {
        string s = Report.ReportName;
        return (s != null && s.Length > 0)? s: "Unnamed Report";
    }
}
```

Finally, the **SaveFile** method prompts the user for a file name and uses an XmlWriter to save each report into the new file using C1Report.**Save** method. Add the following code to implement the **SaveFile** method:

- Visual Basic

```
Public Sub SaveFile()
   ' get name of file to save
   Dim dlg As New SaveFileDialog()
   dlg.FileName = _fileName
   dlg.Title = "Save Report Definition File"
   If dlg.ShowDialog() <> Windows.Forms.DialogResult.OK Then Return

   ' save file
   Dim w As New XmlTextWriter(dlg.FileName, System.Text.Encoding.Default)
   w.Formatting = Formatting.Indented
   w.Indentation = 2
   w.WriteStartDocument()

   ' write all reports to it
   Cursor = Cursors.WaitCursor
   w.WriteStartElement("Reports")
   Dim rh As ReportHolder
   For Each rh In _list.Items
      rh.Report.Save(w) 'rh.Report.ReportName
   Next
   w.WriteEndElement()
   Cursor = Cursors.Default

   ' close the file
   w.Close()

   ' and be done
   _fileName = dlg.FileName
   _dirty = False
   UpdateUI()
End Sub
```

- C#

```
public void SaveFile()
{
    // get name of file to save
    SaveFileDialog dlg = new SaveFileDialog();
    dlg.FileName = _fileName;
    dlg.Title = "Save Report Definition File";
    if (dlg.ShowDialog() != DialogResult.OK)
        return;

    // save file
    XmlTextWriter w = new XmlTextWriter(dlg.FileName,
System.Text.Encoding.Default);
    w.Formatting = Formatting.Indented;
    w.Indentation = 2;
    w.WriteStartDocument();

    // write all reports to it
    Cursor = Cursors.WaitCursor;
    w.WriteStartElement("Reports");
    foreach (ReportHolder rh in _list.Items)
        rh.Report.Save(w); //rh.Report.ReportName;
    w.WriteEndElement();
    Cursor = Cursors.Default;

    // close the file
    w.Close();

    // and be done
    _fileName = dlg.FileName;
    _dirty = false;
    UpdateUI();
}
```

## Step 7 of 9: Hook Up the Controls

The next step is to add the event handlers that hook up all the controls together.

Here is the handler for the **SelectedIndexChanged** event of the **_list** control. Add the following code so that when the user selects a new report from the list, the code displays it in design mode:

- Visual Basic

```
' a new report was selected: switch to design mode and show it
Private Sub _list_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles _list.SelectedIndexChanged
```

```
   ' switch to design mode
   SetDesignMode(True)

   ' attach selected report to designer and preview controls
   _c1rd.Report = Nothing
   _c1ppv.Document = Nothing
   If _list.SelectedIndex > -1 Then
      _c1rd.Report = _list.SelectedItem.Report
   End If
End Sub
```

- C#

```
// a new report was selected: switch to design mode and show it
private void _list_SelectedIndexChanged(object sender, System.EventArgs e)
{
    // switch to design mode
    SetDesignMode(true);

    // attach selected report to designer and preview controls
    _c1rd.Report = null;
    _c1ppv.Document = null;
    if (_list.SelectedItem != null)
        _c1rd.Report = ((ReportHolder)_list.SelectedItem).Report;
}
```

The designer uses a property grid control (**_ppg**) to expose the properties of the report elements selected in the designer. This is done by setting the **SelectedObject** property of the property grid control; in response the control fires a SelectionChanged event.

When the user selects a report section or a field in the designer control, it fires the SelectionChanged event. The event handler inspects the new selection and assigns it to the property grid control. This is a powerful mechanism. The selection can be a single report field, a group of fields, a section, or the whole report.

Add the following code to implement the SelectionChanged event:

- Visual Basic

```
' the selection changed, need to update property grid and show the
' properties of the selected object
Private Sub _c1rd_SelectionChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles _c1rd.SelectionChanged
   Dim sel As Object() = _c1rd.SelectedFields
   If (sel.Length > 0) Then
      _lblPropGrid.Text = "Field Properties"
      _ppg.SelectedObjects = sel
   ElseIf Not IsNothing(_c1rd.SelectedSection) Then
         _lblPropGrid.Text = "Section Properties"
         _ppg.SelectedObject = _c1rd.SelectedSection
      ElseIf Not IsNothing(_c1rd.Report) Then
            _lblPropGrid.Text = "Report Properties"
            _ppg.SelectedObject = _c1rd.Report
         ' nothing selected
         Else
            _lblPropGrid.Text = "Properties"
            _ppg.SelectedObject = Nothing
         End If
   ' done
   UpdateUI()
End Sub
```

- C#

```
// the selection changed, need to update property grid and show the
// properties of the selected object
private void _c1rd_SelectionChanged(object sender, System.EventArgs e)
{
    object[] sel = _c1rd.SelectedFields;
    if (sel.Length > 0)
    {
        _lblPropGrid.Text = "Field Properties";
        _ppg.SelectedObjects = sel;
    }
    else if (_c1rd.SelectedSection != null)
    {
        _lblPropGrid.Text = "Section Properties";
        _ppg.SelectedObject = _c1rd.SelectedSection;
    }
    else if (_c1rd.Report != null)
    {
        _lblPropGrid.Text = "Report Properties";
        _ppg.SelectedObject = _c1rd.Report;
    }
    else // nothing selected
    {
        _lblPropGrid.Text = "Properties";
        _ppg.SelectedObject = null;
    }

    // done
    UpdateUI();
}
```

The property grid (**_ppg**) displays the properties of the object selected in the designer (**_c1rd**). When the user changes the properties of an object using the grid, the designer needs to be notified so it can update the display. Conversely, when the user edits an object using the designer, the grid needs to be notified and update its display.

Add the following code to implement the handlers for the **PropertyValueChanged** event of the **_ppg** control and the ValuesChanged event of the **_c1rd** control:

- Visual Basic

```
' when a value changes in the property window, refresh the designer to
show the changes
Private Sub _ppg_PropertyValueChanged(ByVal s As Object, ByVal e As
System.Windows.Forms.PropertyValueChangedEventArgs) Handles
_ppg.PropertyValueChanged
   _c1rd.Refresh()
   _dirty = True
   UpdateUI()
End Sub

' when properties of the selected objects change in the designer,
' update the property window to show the changes
Private Sub _c1rd_ValuesChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles _c1rd.ValuesChanged
   _c1rd.Refresh()
   _dirty = True
   UpdateUI()
End Sub
```

- C#

```
// when a value changes in the property window, refresh the designer
// to show the changes
private void _ppg_PropertyValueChanged(object s,
      Systems.Windows.Forms.PropertyValueChangedEventArgs e)
{
    _c1rd.Refresh();
    _dirty = true;
    UpdateUI();
}
// when properties of the selected objects change in the designer,
// update the property window to show the changes
private void _c1rd_ValuesChanged(object sender, System.EventArgs e)
{
    _ppg.Refresh();
    _dirty = true;
    UpdateUI();
}
```

## Step 8 of 9: Add Code to Create and Remove Reports

To remove reports from the list, use the **DeleteReport** method. The **DeleteReport** method simply removes the selected item from the report list, clears the Report property of the designer control, then makes a new selection if the list is not empty.

Add the following code to remove reports using the **DeleteReport** method:

- Visual Basic

```
' remove current report from the list
Private Sub DeleteReport()
   ' a report must be selected
   Dim index As Integer = _list.SelectedIndex
   If (index < 0) Then Return

   ' remove report from the designer and from the list
   _c1rd.Report = Nothing
   _list.Items.RemoveAt(index)

   ' select another report if we can
   If (index > _list.Items.Count – 1) Then
      index = _list.Items.Count - 1
      If (index > - 1) Then
          _list.SelectedIndex = index
      End If
   End If
   ' done
   _dirty = True
   UpdateUI()
End Sub
```

- C#

```
// remove current report from the list
private void DeleteReport()
{
      // a report must be selected
      int index = _list.SelectedIndex;
      if (index < 0) return;
      // remove report from the designer and from the list
      _c1rd.Report = null;
      _list.Items.RemoveAt(index);

      // select another report if we can
      if (index > _list.Items.Count-1)
          index = _list.Items.Count-1;
      if (index > -1)
          _list.SelectedIndex = index;

      // done
      _dirty = true;
      UpdateUI();
}
```

The **AddReport** method is a little more complex. In the full-fledged report designer, this command invokes a wizard that allows the user to select a data source, grouping options, layout, and style. When implementing your designer, you can use the wizard code as-is or customize it to suit your needs.

Rather than just creating a blank new report, the simple designer prompts the user for an MDB file, selects the first table it can find, then the first five fields, and creates a report based on that.

Add the following code to create reports using the **AddReport** method:

- Visual Basic

```
Private Sub NewReport()
   ' select a data source (just mdb files in this sample)
   Dim dlg As New OpenFileDialog()
   dlg.FileName = "*.mdb"
   dlg.Title = "Select report data source"
   If dlg.ShowDialog() <> Windows.Forms.DialogResult.OK Then Return

   ' select first table from data source
   Dim connString As String =
String.Format("Provider=Microsoft.Jet.OLEDB.4.0;Data Source={0}",
dlg.FileName)

   Dim tableName As String = GetFirstTable(connString)

   If tableName.Length = 0 Then
      MessageBox.Show("Failed to retrieve data from the selected source.")
      Return
   End If

   ' create new report
   Dim rpt As New C1Report()
   rpt.ReportName = tableName

   ' set data source
   rpt.DataSource.ConnectionString = connString
   rpt.DataSource.RecordSource = tableName

   ' add a title field
   Dim s As Section = rpt.Sections(SectionTypeEnum.Header)
   s.Visible = True
   s.Height = 600
   Dim f As Field = s.Fields.Add("TitleField", tableName, 0, 0, 4000, 600)
   f.Font.Bold = True
   f.Font.Size = 24
   f.ForeColor = Color.Navy

   ' add up to 5 calculated fields
   Dim fieldNames As String() = rpt.DataSource.GetDBFieldList(True)
   Dim cnt As Integer = Math.Min(5, fieldNames.Length)

   ' add a page header
   s = rpt.Sections(SectionTypeEnum.PageHeader)
   s.Visible = True
   s.Height = 400
   Dim rc As New Rectangle(0, 0, 1000, s.Height)

   Dim i As Integer
   For i = 0 To cnt - 1
      f = s.Fields.Add("TitleField", fieldNames(i), rc)
      f.Font.Bold = True
      rc.Offset(rc.Width, 0)
   Next

   ' add detail section
   s = rpt.Sections(SectionTypeEnum.Detail)
   s.Visible = True
   s.Height = 300
   rc = New Rectangle(0, 0, 1000, s.Height)
   For i = 0 To cnt - 1
      f = s.Fields.Add("TitleField", fieldNames(i), rc)
      f.Calculated = True
      rc.Offset(rc.Width, 0)
   Next

   ' add new report to the list and select it
   _list.Items.Add(New ReportHolder(rpt))
   _list.SelectedIndex = _list.Items.Count - 1

   ' done
   _dirty = True
   UpdateUI()
End Sub
```

- C#

```
private void NewReport()
{
      // select a data source (just mdb files in this sample)
      OpenFileDialog dlg = new OpenFileDialog();
      dlg.FileName = "*.mdb";
      dlg.Title = "Select report data source";
      if (dlg.ShowDialog() != DialogResult.OK) return;

      // select first table from data source
      string connString =
            string.Format(@"Provider=Microsoft.Jet.OLEDB.4.0;Data
Source={0};",
            dlg.FileName);
      string tableName = GetFirstTable(connString);
      if (tableName == null || tableName.Length == 0)
      {
            MessageBox.Show("Failed to retrieve data from the selected 
source.");
            return;
      }

      // create new report
      C1Report rpt = new C1Report();
      rpt.ReportName = tableName;

      // set data source
      rpt.DataSource.ConnectionString = connString;
      rpt.DataSource.RecordSource = tableName;

      // add a title field
      Section s = rpt.Sections[SectionTypeEnum.Header];
      s.Visible = true;
      s.Height = 600;
      Field f = s.Fields.Add("TitleField", tableName, 0, 0, 4000, 600);
      f.Font.Bold = true;
      f.Font.Size = 24;
      f.ForeColor = Color.Navy;

      // add up to 5 calculated fields
      string[] fieldNames = rpt.DataSource.GetDBFieldList(true);
      int cnt = Math.Min(5, fieldNames.Length);

      // add a page header
      s = rpt.Sections[SectionTypeEnum.PageHeader];
      s.Visible = true;
      s.Height = 400;
      Rectangle rc = new Rectangle(0, 0, 1000, (int)s.Height);
      for (int i = 0; i < cnt; i++)
      {
            f = s.Fields.Add("TitleField", fieldNames[i], rc);
            f.Font.Bold = true;
            rc.Offset(rc.Width, 0);
      }

      // add detail section
      s = rpt.Sections[SectionTypeEnum.Detail];
      s.Visible = true;
      s.Height = 300;
      rc = new Rectangle(0, 0, 1000, (int)s.Height);
      for (int i = 0; i < cnt; i++)
      {
            f = s.Fields.Add("TitleField", fieldNames[i], rc);
            f.Calculated = true;
            rc.Offset(rc.Width, 0);
      }

      // add new report to the list and select it
      _list.Items.Add(new ReportHolder(rpt));
      _list.SelectedIndex = _list.Items.Count-1;

      // done
      _dirty = true;
      UpdateUI();
}
```

The following code uses a helper function **GetFirstTable** that opens a connection, retrieves the db schema, and returns the name of the first table it finds. Add the following code:

- Visual Basic

```
Private Function GetFirstTable(connString As String) As String
   Dim conn As New OleDbConnection(connString)
   Try
      ' get schema
      conn.Open()
      Dim dt As DataTable =
conn.GetOleDbSchemaTable(OleDbSchemaGuid.Tables, Nothing)
      Dim dr As DataRow
      For Each dr In dt.Rows
         ' check the table type
         Dim type As String = dr("TABLE_TYPE").ToString().ToUpper()
         If (type <> "TABLE" AndAlso type <> "VIEW" AndAlso type <> "LINK"
Then
             'skip this one
         Else
             ' get the table name
             tableName = dr("TABLE_NAME").ToString()
             Exit For
         End If
      Next

      ' done
      conn.Close()
   Catch
   End Try
   ' return the first table we found
   Return tableName
End Function
```

- C#

```
private string GetFirstTable(string connString)
{
    string tableName = null;
    OleDbConnection conn = new OleDbConnection(connString);
    try
    {
        // get schema
        conn.Open();

        DataTable dt = conn.GetOleDbSchemaTable(OleDbSchemaGuid.Tables,
null);
        foreach (DataRow dr in dt.Rows)
        {
            // check the table type
            string type = dr["TABLE_TYPE"].ToString().ToUpper();
            if (type != "TABLE" && type != "VIEW" && type != "LINK")
                continue;

            // get the table name
            tableName = dr["TABLE_NAME"].ToString();
                break;
        }

        // done
        conn.Close();
    }
    catch {}

    // return the first table we found
    return tableName;
}
```

## Step 9 of 9: Add Code to Create Fields

The simple designer is almost done; it is only missing the code used to create new fields in the report.

If you look at the code we are using for the toolbar event handler, you will see that it set the CreateFieldInfo property on the designer and says to wait for the CreateField event from the designer.

Add the following code to create new fields in the report:

- Visual Basic

```
' handle clicks on toolbar buttons
Private Sub _tb_ButtonClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ToolBarButtonClickEventArgs) Handles _tb.ButtonClick
   ' …
   ' add fields
   ' (just set create info and wait for CreateField event from designer)
   If e.Button.Equals(_btnAddField) Then
      _c1rd.CreateFieldInfo = e.Button
   If e.Button.Equals(_btnAddLabel) Then
      _c1rd.CreateFieldInfo = e.Button
End Sub
```

- C#

```
// handle clicks on toolbar buttons
private void _tb_ButtonClick(object sender,
System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
    // …

    // add fields
    // (just set create info and wait for CreateField event from designer)
    if (e.Button == _btnAddField)    _c1rd.CreateFieldInfo = e.Button;
    if (e.Button == _btnAddLabel)    _c1rd.CreateFieldInfo = e.Button;
}
```

The CreateFieldInfo property can be set to any non-null object to indicate to the designer that you want to create a new field. The designer doesn't know what type of field you want to create or how you want to initialize it, so it tracks the mouse and allows the user to draw the field outline on a section. It then fires the CreateField event passing the information you need to create the field yourself.

Add the following code to handle the CreateField event:

- Visual Basic

```
Dim _ctr As Integer

Private Sub _c1rd_CreateField(ByVal sender As Object, ByVal e As
C1.Win.C1ReportDesigner.CreateFieldEventArgs) Handles _c1rd.CreateField
   ' save undo info
   _c1rd.UndoStack.SaveState()

   ' add label field
   _ctr = _ctr + 1
   Dim fieldName As String = String.Format("NewField{0}", _ctr)
   Dim fieldText As String = fieldName
   Dim f As Field = e.Section.Fields.Add(fieldName, fieldText,
e.FieldBounds)

   ' if this is a calculated field,
   ' change the Text and Calculated properties
   If e.CreateFieldInfo.Equals(_btnAddField) Then
      Dim fieldNames As String() =
_c1rd.Report.DataSource.GetDBFieldList(True)
      If (fieldNames.Length > 0) Then
         f.Text = fieldNames(0)
         f.Calculated = True
      End If
   End If
End Sub
```

- C#

```
int _ctr = 0;
private void _c1rd_CreateField(object sender,
        C1.Win.C1ReportDesigner.CreateFieldEventArgs e)
{
    // save undo info
    _c1rd.UndoStack.SaveState();

    // add label field
    string fieldName = string.Format("NewField{0}", ++_ctr);
    string fieldText = fieldName;
    Field f = e.Section.Fields.Add(fieldName, fieldText, e.FieldBounds);

    // if this is a calculated field,
    // change the Text and Calculated properties
    if (e.CreateFieldInfo == _btnAddField)
    {
        string[] fieldNames =
_c1rd.Report.DataSource.GetDBFieldList(true);
        if (fieldNames.Length > 0)
        {
            f.Text = fieldNames[0];
            f.Calculated = true;
        }
    }
}
```

Note how the code starts by calling the SaveState method on the designer, so the user can undo the field creation. After that, the field is created, and the CreateFieldInfo parameter is used to customize the new field and make it behave as a label or as a calculated field.

That concludes the simple designer application: an introduction to the operation of the C1ReportDesigner control.

# Reports for .NET Designer Edition Samples

Please be advised that this ComponentOne software tool is accompanied by various sample projects and/or demos, which may make use of other development tools included with the ComponentOne Studios. Samples can be accessed from the **ComponentOne Sample Explorer**. To view samples, on your desktop, click the **Start** button and then click **ComponentOne | Reports Designer Edition | Samples | C1Report Samples** or **C1Preview Samples**.

**Visual Basic and C# Samples**

| Sample | Description |
|---|---|
| SimpleDesigner | Uses the C1ReportDesigner control to implement a simple report designer. |

Also see the **Reports for WinForms** documentation for additional reporting and previewing samples.

# ComponentOne Reports for WinForms

The following topics detail how to use **ComponentOne Reports for WinForms**.

# ComponentOne Reports for WinForms

**ComponentOne Reports™ for WinForms** provides all the tools you need to meet your reporting, printing, previewing, and exporting needs. Add Microsoft-Access style database reporting. Create complex hierarchical documents with automatic word index, TOC generation, data binding, and more. Export, print, or preview your reports and documents. This edition of **Reports for WinForms** combines two previous products: **Reports for .NET** and **Preview for .NET**. The full functionality of the older **Reports for .NET** product is preserved, but the assembly name and namespace have changed.

### Reporting

Generate Microsoft Access-style reports for your Visual Studio .NET applications quickly and easily with **ComponentOne Reports for WinForms**.

- The C1Report component, which generates data-based banded reports. Render reports directly to a printer or preview control, or export to various portable formats (including XLS, PDF, HTML, text, and images). The component exposes a rich object model for creating, customizing, loading, and saving report definitions. See Working with C1Report for more information.

- The C1RdlReport component, a component that represents an RDL (Report Definition Language) report defined using the 2008 version of the RDL specification. The C1RdlReport component is similar to the C1Report component with the addition of RDL support. See Working with C1RdlReport for more information.

- The **C1ReportDesigner** designer, a stand-alone application used to create report definitions without writing code. The designer allows you to quickly create and edit report definitions, or to import existing Microsoft Access and Crystal report definitions. The designer mimics the Microsoft Access interface, so, if you currently use Microsoft Access, you will quickly adapt to using **C1ReportDesigner**. See Working with C1ReportDesigner for more information.

- The **C1ReportsScheduler** application, a stand-alone application used to schedule report creation to run in the background. Using the **C1ReportsScheduler** application, you can choose what reports to export or print, formats to export the report to, and the schedule and frequency for creating reports. See Working with C1ReportsScheduler for more information.

### Printing and Previewing

No matter how simple or complex your printing requirements, **Reports for WinForms** can help you add printing and previewing capabilities to your project quickly and easily.

- The C1PrintDocument component provides a rich object model which allows you to create arbitrarily complex documents in code. The object model specifically targets paginated documents, providing a rich set of features facilitating automatic and intelligent pagination of complex structured documents. Documents can be completely created in code, or bound to a database via a powerful and flexible data binding model. C1PrintDocument can also import and generate report definitions. See Working with C1PrintDocument for more information.

- The C1MultiDocument component is designed to allow creating, persisting, and exporting large documents that cannot be handled by a single C1PrintDocument object due to memory limitations. C1MultiDocument supports links between contained documents, common TOC, common page numeration, and total page count.

- The **Reports for WinForms** visual preview components provide a powerful, flexible and easy to use set of tools that let you quickly add document viewing capabilities to your application. The integrated components (the C1PrintPreviewControl control and the C1PrintPreviewDialog dialog box) make adding a professional-looking preview to your applications a snap, while the set of specialized controls (C1PreviewPane, C1PreviewThumbnailView, C1PreviewOutlineView, C1PreviewTextSearchPanel) allow you to fine-tune your preview as much as you need.

## What's New in Reports for WinForms

This documentation was last revised for 2011 v2. Enhancements and changes were made to **ComponentOne Reports for WinForms** in this release.

### New Features

The following improvements were made to the **C1MultiDocument** control in the 2011 v2 release of **ComponentOne Reports for WinForms**:

- **New C1MultiDocument Features**

  The following improvements were made to the **C1MultiDocument** control in the 2011 v2 release of **ComponentOne Reports for WinForms**:

  - In addition to **C1PrintDocument**, now **C1Report** and **C1RdlReport** objects can be added to a **C1MultiDocument**. If any reports within a multi-document require parameters, they are requested prior to generating the multi-document.
  - Outline support was added to C1MultiDocument. A collection of outline nodes specific to the multi-document may be specified via the Outlines property. The resulting outline (such as for the preview) is built as a combination of outline nodes in that collection and outline nodes in the contained documents. See C1MultiDocument Outlines for details.
  - The ItemLoaded event was added to C1MultiDocument. This event occurs when the report or document represented by an item is loaded into memory (deserialized) prior to generation, but after processing of any parameters. This event allows programmatically adjusting the properties of the report or document before it is generated. For instance, the data source of a C1Report may be assigned here.

### Improvements and Changes

The following improvements were made in the 2011 v2 release of **ComponentOne Reports for WinForms**:

- New Japanese translations were added.
- The controls provides better diagnostics when trying to open a file in unknown/unsupported format.
- In **C1PrintDocument**'s tags input form, now an up/down spanner is used instead of a drop-down calendar for DateTime input if the format is time.
- The UpcE barcode format is now supported
- A default name ("C1Document") is assigned to a print job when a document without a user specified name is printed. This avoids problems when printing to Adobe Acrobat X and some other virtual printers (output file is not created)..
- Improved performance when serializing styles.
- .NET 4.0 build only: added explicit references to designer assemblies. This fixes issues when multiple versions of C1Report are installed on the same system.
- Properties defined in user types derived from C1PrintDocument, RenderObject or other C1 types are now excluded from C1D/X serialization by default. If you have your own types derived from C1 types, and are sure that you need those properties to be serialized, mark them with either **XmlElementAttribute** or **XmlAttributeAttribute**. (If an exception occurs during serialization, that exception now contains a Log property that might help identify the problem.)
- Removed run time-only **C1PrintDocument.DocumentInto.UserData** from design time property editor.
- When deserializing very large C1PrintDocument objects, use a temporary disk file instead of memory. This helps avoid out of memory exceptions in certain scenarios, such as when previewing a large C1MultiDocument.

### New Members

New members were added to **Reports for Winforms** in the 2011 v2 release.

| **Class** | **Member** | **Description** |
|---|---|---|
| C1MultiDocument | ClearGeneratedPages method | This method allows clearing generated pages and other data without deleting the content of a document or report, so that it can be regenerated. |
| C1MultiDocument | IsGenerating property | Indicates whether the document is currently being generated. |
| C1MultiDocument | ItemAdded event | Occurs when a C1MultiDocumentItem has been added to the current multi-document's Items collection. |
| C1MultiDocument | ItemAdding event | Occurs when a C1MultiDocumentItem is about to be added to the current multi-document's Items collection |
| C1MultiDocument | ItemGenerating event | Occurs when the generation of a document or report associated with a C1MultiDocumentItem is about to start. |
| C1MultiDocument | ItemLoaded event | Occurs when the report or document represented by an item is loaded into memory (deserialized) prior to generation, but after processing of any parameters. This event allows programmatically adjusting the properties of the report or document before it is generated. For instance, the data source of a **C1Report** may be assigned here. |
| C1MultiDocument | ItemRemoved event | Occurs when a C1MultiDocumentItem has been removed from the current multi-document's Items collection. |
| C1MultiDocument | ItemRemoving event | Occurs when a C1MultiDocumentItem is about to be removed from the current multi-document's Items collection. |
| C1MultiDocument | ItemsClear event | Occurs when the current multi-document's Items collection has been cleared. |
| C1MultiDocument | ItemsClearing event | Occurs when the current multi-document's Items collection is about to be cleared. |
| C1MultiDocument | MakeOutlines method | Builds the outline tree for the current multi-document, merging the multi-document's own outlines (Outlines) and outline trees of documents and reports contained within the multi-document. |
| C1MultiDocument | Outlines property | Gets the OutlineNodeCollection containing outline nodes specified for the multi-document.<br>Note that this collection does not include outlines of documents and reports contained within this multi-document. Use the MakeOutlines method to build the complete outline tree for the multi-document. |
| C1MultiDocument | IsGenerating property | Indicates whether the document is currently being generated. |
| C1MultiDocument | UserData property | Gets or sets arbitrary data associated with the current document. |
| C1MultiDocumentItem | OutlineNode property | Gets or sets the outline node representing the current item in Outlines. |

| | | |
|---|---|---|
| C1MultiDocumentItem | Outlines property | Gets the collection of outlines defined on the document or report represented by the current multi-document item. |
| C1MultiDocumentItemCollection | Remove method | Removes an item from the current collection. |
| C1PrintDocument | ClearGeneratedPages method | This method allows clearing generated pages and other data without deleting the content of a document or report, so that it can be regenerated. |
| C1PrintDocument | HasEditableTags method | Checks whether the document contains any tags that can be entered or changed in the tags input dialog box. Note that this method does NOT check the value of the ShowTagsInputDialog property. |
| C1Report | ClearGeneratedPages method | This method allows clearing generated pages and other data without deleting the content of a document or report, so that it can be regenerated. |
| C1RdlReport | ClearGeneratedPages method | This method allows clearing generated pages and other data without deleting the content of a document or report, so that it can be regenerated. |
| C1RdlReport | EmfType property | Gets or sets the type of metafiles created by the current document. |

**Tip:** A version history containing a list of new features, improvements, fixes, and changes for each product is available on the ComponentOne Web site at http://helpcentral.componentone.com/VersionHistory.aspx.

## Revision History

The revision history details recent enhancements to **Reports for WinForms**.

### *What's New in 2011 v1*

Several enhancements and changes were made to **ComponentOne Reports for WinForms** in the 2011 v1 release.

#### New Features

The following improvements were made in the 2011 v1 release of **ComponentOne Reports for WinForms**:

- **New C1MultiDocument Component**

  The C1MultiDocument component was added in this release. This component is designed to allow creating, persisting and exporting large documents that cannot be handled by a single C1PrintDocument object due to memory limitations. See Working with C1MultiDocument for more information.

#### Improvements and Changes

The following improvements were made in the 2011 v1 release of **ComponentOne Reports for WinForms**:

- **Obsolete Property**

  The **C1Report.PageImages** property has been marked as obsolete as using the GetPageCount, GetPageImage, and GetPageImages methods is preferable.

- **RTF Support**

  Added support for the latest RTF implementation from Microsoft if that is available on the system: rtf window class: RICHEDIT60W, provided by riched20.dll shipped with MS Office 2007.

### New Members

New members were added to **Reports for Winforms** in the 2011 v1 release.

| Class | Member | Description |
|---|---|---|
| C1PrintDocument | PageNumberingChange property | Gets or sets the **PageNumberingChange** object applied to the first page of the document. |
| C1PrintOptions | MsPrintDocumentPrintEmfType property | Gets or sets the type of metafiles used when printing standard .NET **PrintDocument** documents. The default is **EmfType.EmfPlusDual**.<br>If your printer needs to scan the output for data (for example, a fax number; a case in point is Tobit Faxware) and fails, try setting this to **EmfType.EmfOnly**. Note that this property ONLY affects printing of standard .NET **PrintDocument** documents, and has no bearing on other supported document types. |
| C1Report | GetPageCount method | Gets the total number of pages in the generated report. Returns the total page count. |
| C1Report | GetPageImage method | Gets a metafile representing the specified page. pageIndex: 0-based page index. Returns the metafile representing the page. |
| C1Report | GetPageImages method | Enumerates page images (represented by metafiles) of the generated report. Using this method in a "foreach" loop allows iterating over all pages of a large report without exhausting system resources. This approach is preferable to **PageImages** property that creates images for all pages and can consume a large amount of resources. Returns: An IEnumerable that allows iterating over the page images. The **C1Report.PageImages** property has been marked as obsolete as using the new methods described above is preferable. |

## *What's New in 2010 v3*

Several enhancements and changes were made to **ComponentOne Reports for WinForms** in the 2010 v3 release.

### New Features

The following improvements were made in the 2010 v3 release of **ComponentOne Reports for WinForms**:

- **New C1RdlReport Component**

  The C1RdlReport component was added in this release. This component represents an RDL (Report Definition Language) report defined using the 2008 version of the RDL specification. The C1RdlReport component is similar to the C1Report component with the addition of RDL support. See Working with C1RdlReport for more information.

  Note that RDL import in **C1PrintDocument** (provided by the ImportRdl and FromRdl methods) is now obsolete. The **C1RdlReport** control should be used instead.

### Improvements and Changes

The following improvements were made in the 2010 v3 release of **ComponentOne Reports for WinForms**:

- **Azure Support**

  Support for the Azure platform was added.

- **Loading Speed**

  The loading speed for **C1Report** XML report definition files containing multiple reports with subreports was improved.

- **Pre-Built CustomFields**

  For .NET 4.0 version only: a pre-built version of C1.C1Report.CustomFields.4.dll is now included in the release. It no longer directly references a specific version of the C1Chart assembly (it is used via reflection), so C1.Win.C1Chart.4.dll can be updated without recompiling CustomFields. See the CustomFields sample for details.

- **Size of Generated Reports**

  The size of reports generated at design time when a report is loaded into the preview control were limited. The following are the control limits:

  - **C1Report**: maximum: 4 pages
  - **C1RdlReport**: maximum: 12 records

### New Members

New members were added to **Reports for Winforms** in the 2010 v3 release.

| Class | Member | Description |
|---|---|---|
| C1PageSettings | PrinterResolutionX property | This property allows setting the printer resolution when printing. (Some printers require resolution to be set to print correctly.) |
| C1PageSettings | PrinterResolutionY property | This property allows setting the printer resolution when printing. (Some printers require resolution to be set to print correctly.) |
| C1Report | ColorizeHyperlinks property | Gets or sets a value indicating whether hyperlinks in the report are colorized automatically. The default is False (which is compatible with older versions of C1Report). |

## *What's New in 2010 v2*

Several enhancements and changes were made to **ComponentOne Reports for WinForms** in the 2010 v2 release.

### New Features, Improvements, and Changes

The following enhancements were made in the 2010 v2 release of **ComponentOne Reports for WinForms**:

- **C1ReportDesigner Enhancements**

  The following enhancements were made to the **C1ReportDesigner** application installed with **Reports for WinForms**.

  - **C1ReportDesigner** no longer requires Administrator rights when running under Vista/Windows7.
  - The state of SQL editor dialog box is now saved between sessions.

- The name of the current report is now shown in the status bar.
- **C1ReportDesigner** now automatically adds brackets to fields that contain dots, spaces, and so on to make names valid identifiers.

- **.Net 4.0 Support and Limitations**

  Support for the .NET 4.0 client profile was added to .NET 4.0 builds of **Reports for WinForms**. HtmlTableFilter is **not** supported in the .NET 4.0 build. This is a temporary limitation that will be removed in a future build.

- **New Localize Option**

  A new **Localize** option was added to the C1Report control's design-time context menu. See Localization for more information.

- **New Localizable String**

  The **C1Report.ScriptEditor** string was added; the string specifies the caption bar text of the dialog box displayed by C1Report to edit parameter values.

### New Members

New members were added to **Reports for Winforms** in the 2010 v2 release.

| Class | Member | Description |
|---|---|---|
| C1PageSettings | PaperSourceRawKind property | Prevents custom papersource information loseses during conversions between **C1PageSettings** and **PageSettings**. |

## Reports and Preview .NET Versions

The **ComponentOne Reports for WinForms** product has evolved through several versions. The current version (#6 in the table below) is a combination of the .NET 2.0 **Preview for .NET** and **Reports for .NET** products. The following table describes the available .NET versions of ComponentOne reporting and previewing products. Note that the list has been numbered to differentiate between versions (this product, **Reports for WinForms**, is #6 below):

| # | Name | .NET Framework | Assemblies | Controls |
|---|---|---|---|---|
| 1 | Preview for .NET | .NET 1.x | C1.C1PrintDocument.dll<br>C1.Win.C1PrintPreview.dll | C1PrintDocument<br>C1PrintPreview |
| 2 | Reports for .NET | .NET 1.x | C1 C1.Win.C1Report.dll | C1Report |
| 3 | Preview Classic for .NET | .NET 2.0 | C1.C1PrintDocument.Classic.2.dll<br>C1.Win.C1PrintPreview.Classic.2.dll | C1PrintDocument<br>C1PrintPreview |
| 4 | Reports for .NET | .NET 2.0 | C1.Win.C1Report.2.dll | C1Report |
| 5 | Preview for .NET | .NET 2.0 | C1.C1Preview.2.dll<br>C1.Win.C1Preview.2.dll | C1PrintDocument<br>C1PreviewPane |

| | | | | |
|---|---|---|---|---|
| | | | | C1PrintPreviewControl<br>C1PrintPreviewDialog<br>C1PreviewThumbnailView<br>C1PreviewOutlineView<br>C1PreviewTextSearchPanel |
| 6 | Reports for WinForms | .NET 2.0 | C1.C1Report.2.dll<br>C1.Win.C1Report.2.dll | C1Report<br>C1PrintDocument<br>C1PreviewPane<br>C1PrintPreviewControl<br>C1PrintPreviewDialog<br>C1PreviewThumbnailView<br>C1PreviewOutlineView<br>C1PreviewTextSearchPanel |
| 7 | Reports for WPF | .NET 3.0 | C1.WPF.C1Report.dll<br>C1.WPF.C1Report.Design.dll<br>C1.WPF.C1Report.VisualStudio.Design.dll | C1DocumentViewer |

### Version Compatibility

While the products above provide reporting and previewing functionality and may include similar components, they are not all backwards compatible. Some considerations for upgrading versions are discussed below:

- **Preview Classic for .NET (.NET 2.0, #3 in the above table)**

  This is the "classic" version of ComponentOne's preview controls. While the assembly names are different from #1, these are 100% backwards compatible, so upgrading from the .NET 1.x product (#1 above) does not require any changes except for changing the references in your project and licenses.licx files. Preview Classic for .NET is no longer actively developed and is in maintenance mode.

- **Reports for .NET (.NET 2.0, #4 above)**

  This is the old .NET 2.0 version of ComponentOne's reporting controls. These reports can be shown by all versions of preview controls (#1, #3 or #5 above), by assigning the **C1Report.Document** property to the **Document** property of a preview control.

- **Preview for .NET (.NET 2.0, #5 above)**

  This is the newer previewing product (new compared to the classic version). This product has different code and object model from the previous versions (#1or #3 above). Automatic conversion from #1 or #3 to this product is **not** supported; in particular the **Convert2Report.exe** utility can **not** convert those older projects. Converting from #1 or #3 to this preview requires rewriting user code, always. The scope of changes differs and may be trivial, but code **must** be updated by hand.

- **Reports for WinForms (.NET 2.0, #6 above)**

  The current .NET 2.0 combined reporting and previewing product. This includes both the "new" preview (#5 above) and reports (#4 above). Unlike in previous versions, to preview a **C1Report**, it itself (rather than its **Document** property) should be assigned to the **Document** property of the preview control. This build is backwards code compatible with #4 and #5, but assembly references and namespaces must be updated. The changes are always trivial and can be made manually or by using the **Convert2Report.exe** utility which can be downloaded from ComponentOne HelpCentral.

- **Reports for WPF (.NET 3.0, #7 above)**

The WPF version of ComponentOne's reporting and preview controls which includes **Preview for .NET 2.0** (#5) and **Reports for .NET 2.0** (#4). This product can be used in .NET 3.0 and 3.5 applications.

## Migrating a Reports for WinForms Project to Visual Studio 2005

To migrate a project using ComponentOne components to Visual Studio 2005, there are two main steps that must be performed. First, you must convert your project to Visual Studio 2005, which includes removing any references to a previous assembly and adding a reference to the new assembly. Secondly, the .licx file, or licensing file, must be updated in order for the project to run correctly.

**To convert the project:**

1. Open Visual Studio 2005 and select **File | Open | Project/Solution**.
2. Locate the **.sln** file for the project that you wish to convert to Visual Studio 2005. Select it and click **Open**. The **Visual Studio Conversion Wizard** appears.
3. Click **Next**.
4. Select **Yes, create a backup before converting** to create a backup of your current project and click **Next**.
5. Click **Finish** to convert your project to Visual Studio 2005. The **Conversion Complete** window appears.
6. Click **Show the conversion log when the wizard is closed** if you want to view the conversion log.
7. Click **Close**. The project opens. Now you must remove references to any of the previous ComponentOne .dlls and add references to the new ones.
8. Go to the Solution Explorer (**View | Solution Explorer**) and click the **Show All Files** button.

   > **Note:** The **Show All Files** button does not appear in the Solution Explorer toolbar if the Solution project node is selected.

9. Expand the **References** node, right-click **C1.Common** and select **Remove**. Also remove C1.Win.C1Report the same way.
10. Right-click the **References** node and select **Add Reference**.
11. Locate and select **C1.Win.C1Report.2.dll**. Click **OK** to add it to the project.

**To update the .licx file:**

1. In the Solution Explorer, right-click the **licenses.licx** file and select **Delete**.
2. Click **OK** to permanently delete **licenses.licx**. The project must be rebuilt to create a new, updated version of the .licx file.
3. Click the **Start Debugging** button to compile and run the project. The new .licx file may not be visible in the Solution Explorer.
4. Select **File, Close** to close the form and then double-click the **Form.vb** or **Form.cs** file in the Solution Explorer to reopen it. The new **licenses.licx** file appears in the list of files.

   The migration process is complete.

## Converting a Preview for WinForms Project to Reports for WinForms

In the 2008 v3 release of the ComponentOne studios, the **Preview for WinForms** and **Reports for WinForms** products were merged into one product: **Reports for WinForms**. The new **Reports for WinForms** contains the **C1Report** component as well as all components and controls that were previously provided by the **C1Preview** assemblies, but, because the assemblies now have different names, projects that used **Preview for WinForms** must have references changed to the new assemblies.

You may want to use this utility if:

- You have existing C# or VB projects that use **C1PrintDocument** or any of the previewing WinForms controls previously provided by the **C1.Win.C1Preview.2** assembly (**C1PrintPreviewControl**, **C1PrintPreviewPane**, and so on).

  OR

- You have an existing C# or VB project that used the old **C1Report** control (build 2.5 or earlier) provided by a single **C1.Win.C1Report.2** assembly.

Specifically, this utility performs the following:

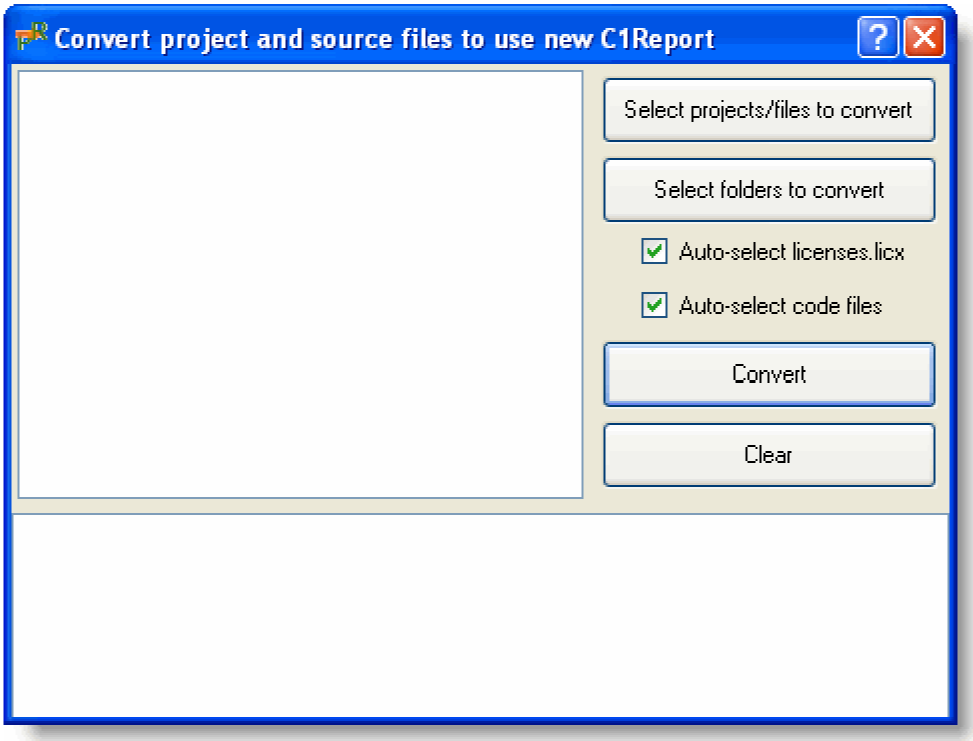
- In C# or VB projects, references are changed from **C1.C1Preview.2** and **C1.Win.C1Preview.2** to the **C1.C1Report.2** and **C1.Win.C1Report.2** assemblies.
- In licenses.licx files, references for **C1PrintDocument** and preview controls are updated to reference the corresponding report assemblies.
- Also in licenses.licx files, references for **C1Report** residing in the **C1.Win.C1Report.2** assembly are updated to point to the **C1.C1Report.2** assembly where it now resides.
- In C# or VB source code files, all mentions of the **C1.Win.C1Report** namespace are replaced with **C1.C1Report**.

The **Convert2Report** utility can convert multiple projects and/or individual files. You can download the **Convert2Report.exe** utility from ComponentOne HelpCentral. Using this utility, select and convert your files.

To convert a project, complete the following steps:

1. Download and open the **Convert2Report.exe** file, located on ComponentOne HelpCentral.

   The application window will look like the following:


Convert project and source files to use new C1Report
Select projects/files to convert
Select folders to convert
☑ Auto-select licenses.licx
☑ Auto-select code files
Convert
Clear

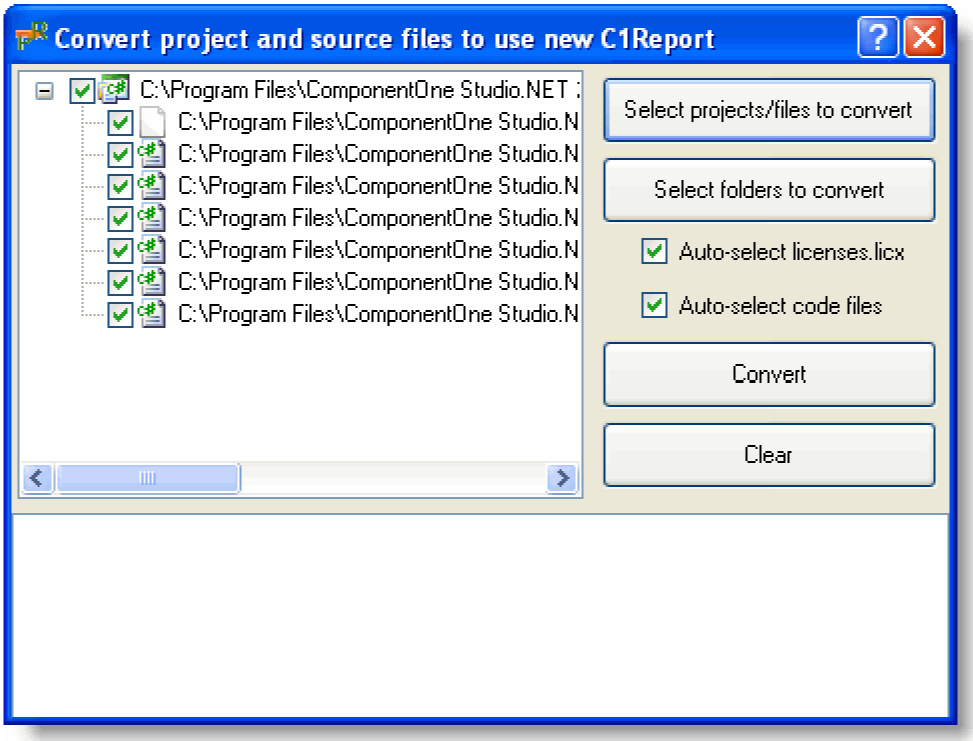
2. Keep the **Auto-select licenses.licx** and **Auto-select code files** check boxes to auto select all included files when a project file is selected for upgrade.

3. Click either the **Select projects/files to convert** or **Select folders to convert** button. The **Open** or **Browse For Folder** dialog box will open.

4. In the **Open** or **Browse For Folder** dialog box, locate either the files or folder you wish to convert and select **OK**.

   If you need to select additional folders or files, you can click the **Select projects/files to convert** or **Select folders to convert** button again to choose more files. If you need to clear your selection and choose files again, press the **Clear** button.

   You dialog box will look similar to the following:

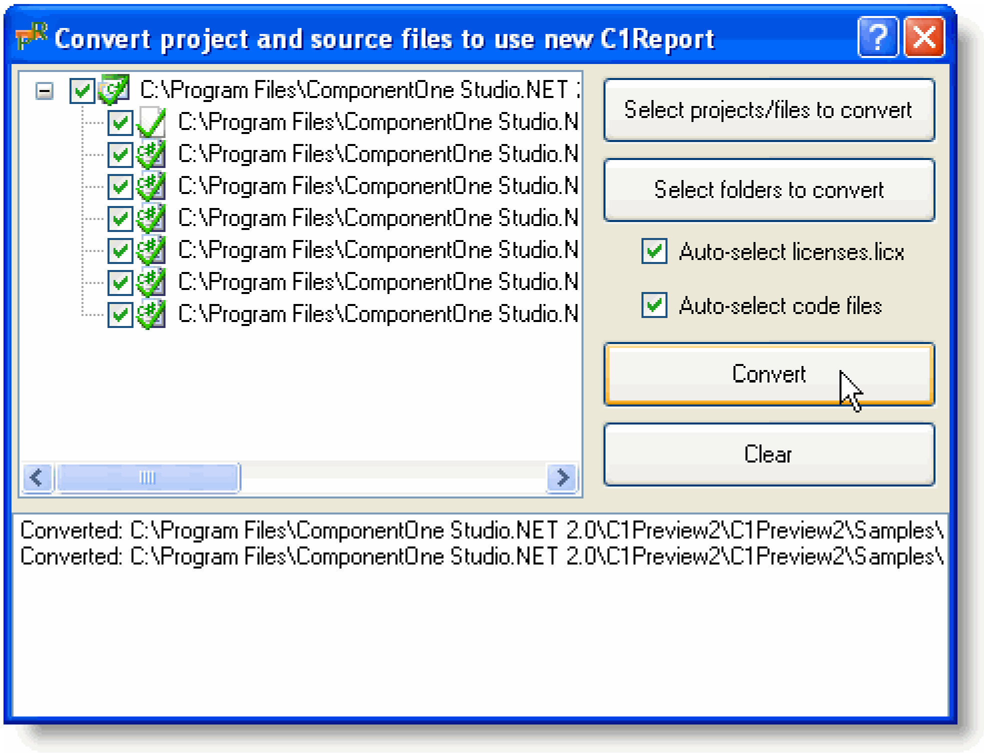


Convert project and source files to use new C1Report
C:\Program Files\ComponentOne Studio.NET :
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
Select projects/files to convert
Select folders to convert
☑ Auto-select licenses.licx
☑ Auto-select code files
Convert
Clear

5. Confirm the files you are converting in the left pane. You can check or uncheck files to change what will be converted.

6. Click **Convert** to convert your **Preview for WinForms** project to using **Reports for WinForms**.

   Files that have been successfully converted will display a green check mark in the left pane.



Convert project and source files to use new C1Report
C:\Program Files\ComponentOne Studio.NET :
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
C:\Program Files\ComponentOne Studio.N
Select projects/files to convert
Select folders to convert
☑ Auto-select licenses.licx
☑ Auto-select code files
Convert
Clear

Converted: C:\Program Files\ComponentOne Studio.NET 2.0\C1Preview2\C1Preview2\Samples\
Converted: C:\Program Files\ComponentOne Studio.NET 2.0\C1Preview2\C1Preview2\Samples\

The bottom pane will list the code and licensing files that have been converted as well as the saved locations for the original and converted files. By default the original file will be saved in the same directory as **FileName.original** and the converted file will be saved as the original file name (overwriting that file).

7. Close the utility after successfully converting all files.

# Key Features

Build customized reviews and reports using **ComponentOne Reports for WinForms** and take advantage of the many features of the **C1Report** and **C1PrintDocument** components, and the six visual previewing controls, including:

## C1Report Features

The **C1Report** control's features include the following:

- **Report Designer Application**
  Quickly create, edit, preview, load, and save report definition files without writing a single line of code. The familiar Microsoft Access-like user interface of the **C1ReportDesigner** application yields fast adaptation.

- **C1Report Wizard**
  You don't have to be an expert to create reports using the **C1Report Wizard**. Effortlessly create a new report from start to finish in five easy steps. Select the data source, report fields, and layout of your report with the **C1Report Wizard** guiding you through each step.

- **Banded Report Model**
  Reports uses a banded report model based on groups, sections, and fields. The banded report model allows for a highly-organized report layout.

- **30+ Built-in Report Templates**
  The enhanced report designer application now includes 34 report templates. Simply select a report theme in the **C1Report Wizard** and you get a professionally styled report. No coding required - your colorful report is just a click away!

- **Microsoft Access and Crystal Reports Compatibility**
  Reports supports features found in Microsoft Access and Crystal Reports. With the click of a button, import Access report files (MDB) and Crystal report files (RPT) using the **C1ReportDesigner**.

- **Flexible Data Binding**
  Specify a connection string and an SQL statement in your report definition and Reports will load the data automatically for you. Optionally, use XML files, custom collections, and other data sources.

- **Parameters for Adding/Limiting Data**
  Reports may contain parameterized queries, allowing users to customize the report by adding/limiting the data that should be included in the report before it is rendered. Specify a value for a report field, filter data, control sorting and grouping, and more. Display only necessary data using report parameters.

- **Combine Several Reports Into One**
  Reports may contain nested reports to arbitrary levels (subreports). You can use the main report to show detailed information and use subreports to show summary data at the beginning of each group.

- **VBScript Expression**
  Reports may include embedded VBScript event handlers, making them self-contained. Format fields according to value, update a page count, hide a section without data, and more when the report is rendered.

- **Formatting, Grouping, Filtering, Sorting, and More**
  Use VBScript expressions to retrieve, calculate, display, group, filter, sort, parameterize, and format the contents of a report, including extensions for aggregate expressions (sum, max, average, and more).

- **Chart Fields**
  Embed charts into your reports to graphically display numerical data. The Report's **Chart** field is implemented using the **C1Chart** control and can display multiple series of data. The supported chart types include **Bar**, **Area**, **Scatter**, **Pie**, **Line** and **Column**.

- **Aggregated Charting**
  Create charts that automatically aggregate data values (ValueY) that have the same category (ValueX) using an aggregate function of your choice. The chart field's Aggregate property tells the chart how to aggregate values that have the same category into a single point in the chart. It can be set to perform any of the common aggregation functions on the data: sum, average, count, maximum, minimum, standard deviation, and variance.

- **Export Formats**
  Render your reports directly to a printer or preview control or export your reports to various portable formats: Excel (XLS, XLSX), PDF, HTML, Rich Text and Compressed Metafiles.

- **Automated Reports**
  Automate reports using the **C1ReportsScheduler**, a stand-alone application used to schedule report creation to run in the background. Using the **C1ReportsScheduler** application, you can choose what reports to export or print, formats to export the report to, and the schedule and frequency for creating reports.

### C1PrintDocument Features

The **C1PrintDocument** component's features include the following:

- **Powerful Document Oriented Object Model**

  The C1PrintDocument component provides a flexible hierarchical document object model with powerful automatic layout, formatting, and pagination control features so there's no need to manually calculate the layout, insert page breaks, and so on.

- **Rich Formatting Options**

  Control the look of your document with support for paragraphs of text with multiple fonts, text and background colors, text positioning (subscript, superscript), inline images, various text alignment (including justified text), and more.

- **Powerful Table Layouts**

  Use the **C1PrintDocument** tables to layout elements in your documents. Apply styles to tables, modify row and column headers, and more. Tables support an Excel-like object model, with a logically infinite number of columns and rows. Simply accessing a table element instantiates it, so you never have to worry about specifying the correct table size.

- **Flexible Sizing and Positioning of Elements**

  Document element size and position can be specified as absolute values, relative to other elements' sizes and positions, or as simple expressions combining absolute and relative values. For example, specify the width of an element as a percentage of the parent element or of the current page width.

- **Automatically Generate TOC and Word Index**

  **C1PrintDocument** supports automated generation of Table of Contents (TOC) and alphabetical word index. Depending on your output format, links in both the TOC and index are clickable and take the user to the referenced page.

- **Add Hyperlinks**

  Make documents interactive by adding hyperlinks, link targets, and more. Any document element can be a hyperlink, or a hyperlink jump target.

- **Data Binding Support**

  Documents can be completely created in code, or bound to a database via a powerful and flexible data binding model.

- **Import Report Definitions**

Combine **C1PrintDocument** with the powerful **C1Report** component, which exposes a rich object model for creating, customizing, loading, and saving report definitions. You can quickly import and generate report definitions with the **C1PrintDocument** component.

- **SQL Server Reporting Services**

  Report Definition Language (RDL) is the reporting scheme commonly used in SQL Server Reporting Services. **C1PrintDocument** allows you to import an SSRS definition file (.rdl). The result is a data bound document representation of the imported report.

- **Export Formats**

  Multiple export format options make saving and sharing documents easy. Export your documents to Adobe Portable Document Format (PDF), Excel (XLS and XLSX), Word (RTF and DOCX), HTML, and several image formats.

- **Create Adobe Acroforms**

  Documents can include interactive forms (to be filled out by the end user). Add text boxes, list boxes, drop-down lists, check, radio and push buttons to **C1PrintDocument**. These controls become interactive when viewed inside **C1PrintPreviewControl** (see Print Preview). You can also export these documents to Adobe Acroforms.

- **Control Exported PDF Display**

  You can control how an exported PDF file is displayed in Adobe Acrobat. For example, set the how pages are viewed (for instance, one page at a time or two pages in columns) and the visibility of various elements (that is, if thumbnail images or a document outline view is visible).

- **C1DX File Format for Smaller File Sizes**

  A new OPC-based file format for **C1PrintDocument** objects, C1D OpenXML (C1DX) complies with Microsoft Open Packaging Conventions and is similar to the Microsoft Office 2007 OpenXML format. Due to built-in compression, resulting files are smaller in size. The current Preview for WinForms C1D format is also fully supported for backwards compatibility.

- **Multiple Page Layouts**

  Several page layouts accommodating different paper sizes, page settings, number of columns, page headers, and so on can be predefined and selected at run time by setting a single property.

- **Combine Multiple Large Documents**

  Use **C1MultiDocument** to combine multiple **C1PrintDocument**s which will be rendered as a whole continuous document with shared page numbering, a common TOC, word index, page count and inter-document hyperlinks. This allows you to create and export very large documents that cannot be handled by a single **C1PrintDocument** object due to memory limitations.

- **Hierarchical Styles**

  Hierarchical styles control the look of all document elements, with intelligent support for ambient and non-ambient style attributes. Specify individual font attributes (such as boldness or font size), table grid lines, and more.

- **GDI+ Text Rendering**

  By setting one property you can render text using the GDI+ text API – with GDI+ text rendering text looks similar to text in Microsoft Office 2007 and matches the default text layout in XPS.

- **Embed True Type Fonts**

  Embed fonts to guarantee text is rendered correctly on any system – even if the fonts used are not installed on the system used to preview or print the document.

- **Dictionary Support**

Store resources (such as images used throughout the document) in the document dictionary to save space and time.

### Print Preview Features

The **Reports for WinForms** visual previewing controls' features include the following:

- **Full-Featured Preview Controls**

  Integrated **C1PrintPreviewControl** and **C1PrintPreviewDialog** controls provide a ready-to-use full-featured UI with thumbnail and outline views, text search, and predefined toolbars right out of the box.

- **Preview Reports**

  Easily integrate Reports with **Reports for WinForms** to add previewing, formatting, printing and exporting functionality to your reports. Just set the **Document** property on the **C1PrintPreviewControl** or **C1PreviewPane** to your reporting control and you are finished.

- **PrintDocument Compatibility**

  In addition to the **ComponentOne Reporting** controls, Reports supports the standard .NET **PrintDocument** component, and can even export to a number of external formats (such as PDF). So you can easily upgrade your applications with minimal effort.

- **Thumbnail Views**

  Reports includes built-in thumbnail views of all pages within any rendered document. The thumbnail view allows quick navigation to any page. Thumbnails are generated on the fly as pages are created so you instantly get thumbnails even if all pages have not finished rendering. Use the separate **C1PreviewThumbnailView** control and attach it to a **C1PreviewPane**, or just use the all-inclusive **C1PrintPreviewControl** to display automatic thumbnail views.

- **Interactive Document Reflow**

  End-users can interactively change the document (for example change page margins or orientation) at run time and the document will automatically reflow to accommodate the changes.

- **Text Search**

  Perform text searching at run-time without any additional coding. The **C1PrintPreviewControl** includes a built-in text search panel. Search results include page numbers and link to found locations.

- **Several Built-in Toolbar Sets**

  Choose from several preset toolbar image collections, or choose your own. The **C1PrintPreviewControl** includes themes to match Adobe, XP, Classic Windows, and the Mac operating system.

- **Zooming Tools**

  Reports supports many different zooming options that you would find in Microsoft Word. Predefined views include: actual size, page width, text width and whole page. You can also use percentages to define a specific zoom value. The **C1PrintPreviewControl** also has a zoom-in and zoom-out tool which allows the user to specify where on the page to zoom in or out.

- **C1PrintDocument Feature Support**

  All preview controls fully support the **C1PrintDocument** component's features such as hyperlinks and outlines.

- **Code-free Development**

  Reports includes extensive design-time support, including ComponentOne SmartDesigner® technology with floating toolbars, allowing you to easily customize your preview window without writing code.

- **Flexible Modular Design**

Use the separate, specialized controls (preview pane, thumbnail and outline views, text search panel) to customize your document view. Combine Reports controls with other **Studio for WinForms** controls such as **Ribbon for WinForms** to create a custom preview window that fits any UI.

- **Localization**

  Create localized versions of all end-user visible strings for different cultures at design time and switch between languages at run time.

### C1ReportDesigner Features

The **C1ReportDesigner** application's features include the following:

- **Easily Accessible**

  While the **C1ReportDesigner** is a stand-alone application, you can easily launch and navigate to it from within Visual Studio. Just select **Edit Report** on the **C1Report** component's smart tag and it will open up the C1ReportDesigner application.

- **Create New Reports**

  Use the **C1Report Wizard** to quickly and easily create a new report. To create a report, simply:

  a. Select the data source for the new report

  b. Select the fields you want to include in the report

  c. Set the layout, style and title for the new report

- **Import Existing Reports**

  The ability to take your existing reports and turn them into ComponentOne Reports is one of the most powerful features of the **C1ReportDesigner**. Import your existing report definitions from Microsoft Access files (.mdb and .adp) or Crystal Reports (.rpt).

- **Design and Modify Reports**

  The Access-style WYSIWYG design surface makes designing reports easy and intuitive. Drag and drop report fields from the toolbar onto the report's design surface. Banded regions mark each area of the report such as header, body and footer. Set all field related properties directly in the application itself. You can even write custom VBScript code from the Properties window.

- **Export Reports**

  Directly export a report to any of the supported file formats: HTML, PDF, RTF, XLS, XLSX, TIF, TXT or ZIP.

- **Save and Distribute Report Definitions**

  Once you have created or imported a report you can save it out to an XML-based report definition file. Package the definition files with the **C1Report** component in your published applications to distribute the reports to your end-users. To distribute or customize the **C1ReportDesigner** application itself, you must use the **ComponentOne Reports for .NET Designer Edition**.

### C1ReportsScheduler Features

The **C1ReportsScheduler** application's features include the following:

- **Generate and Export ComponentOne Reports**

  The **C1ReportsScheduler** is a stand-alone scheduling application that is included with **Reports for WinForms**. It is designed to generate and export ComponentOne Reports in the background on set schedules.

- **Run in Background or Control Schedules**

The C1ReportsScheduler application consists of two interacting parts: a front-end and a Windows service. The Windows service runs in the background, executing specified tasks according to their schedules. The front-end can be used to view or edit the task list, start or stop schedules, and control the service. While the front-end is used to install and setup the service, it is not needed for the service to run.

- **Export to Various Formats**

  **ComponentOne Reports** can be exported to many formats including: PDF, Rich Text, Open XML Word, Excel, HTML, Metafile, images and more.

- **Schedule Reporting Output in 4 Easy Steps**

  To schedule reporting, you would simply need to complete the following:

  a. Select any number of **C1Report** definition files (.xml) from your machine.

  b. Choose any number of actions such as export or print for each report in the **Task Actions** pane.

  c. Set a one-time or recurring schedule for each report.

  d. Press **Start** to initiate the background service. Your reports will be generated in the specified outputs at the scheduled times.

- **Full Source Included**

  You can ship the pre-built application and service to your end-users "as-is," or modify the UI and functionality to fit your needs. We provide complete source code for the **C1ReportsScheduler** as a sample.

- **C1ReportsScheduler Windows Service**

  The **C1ReportsScheduler** Windows service is provided so reports can be generated at any scheduled time because the front-end application will not always be running. The service can be easily installed and uninstalled on your machine through the front-end application. When the **C1ReportsScheduler** is run for the first time, a dialog pops up asking whether you would like to install the service. The service does not need to be installed if the front-end application is to remain running.

### C1MultiDocument Features

The **C1MultiDocument** component's features include the following:

- **Handle Large Documents**

  **C1MultiDocument** can handle large documents that would be otherwise impossible to create/export/print due to memory limitations.

- **Combine Multiple Documents**

  Use of compression and temporary disk storage allows **C1MultiDocument** to combine several **C1PrintDocument** objects into a large multi-document that would cause an out of memory condition if all pages belonged to a single **C1PrintDocument**.

### C1RdlReport Features

The **C1RdlReport** component's features include the following:

- **Exposes Full RDL Object Model**
- The **C1RdlReport** component exposes the full RDL object model following the latest RDL 2008 specification. This allows you to modify existing reports or even create new RDL reports completely in code. This is not possible through Microsoft Reporting Services alone.
- **Generate RDL Reports from any Data Source**
- You are not constrained to using SQL Server data as your data source. **C1RdlReport** can generate RDL reports using any data source, such as an Access database.
- **No External Dependencies**

- C1RdlReport provides a self-contained RDL reporting solution without external dependencies such as the need for a Microsoft Reporting Services server.
- **Seamless Integration with C1Reports**
- **C1RdlReport** provides seamless integration with the entire **ComponentOne Reports** suite. Use **C1RdlReport** with **C1PrintPreviewControl** to provide previewing, formatting, printing and exporting functionality for your reports.
- **Support for RDL Objects and Properties**
- **C1RdlReport** supports most of the common Microsoft reporting features such as subreports, parameters, hyperlinks, charts, shapes, images, text boxes and more.

# Reports for WinForms Components and Controls

Please note that though the functionality is similar to the old version of the product, the object model of both the C1PrintDocument component and the print preview control have been significantly changed, so the new components are not binary- or code-compatible with the old components.

**Reports for WinForms** consists of the following assemblies:

### C1.C1Report.2.dll

**C1.C1Report.2.dll** provides the C1PrintDocument and C1Report components and other public classes providing non-Windows Forms specific services, such as export and more. Components in this assembly include:

- C1Report

  The C1Report component generates data-based banded reports. Render reports directly to a printer or preview control, or export to various portable formats (including XLS, PDF, HTML, text, and images). The component exposes a rich object model for creating, customizing, loading, and saving report definitions.

- C1PrintDocument

  The C1PrintDocument component allows you to create complex documents that can be printed, previewed, persisted in a disc file, or exported to a number of external formats including PDF (Portable Document Format) and RTF (Rich Text File).

- C1MultiDocument

  The C1MultiDocument component is designed to allow creating, persisting and exporting large documents that cannot be handled by a single C1PrintDocument object due to memory limitations.

- C1RdlReport

  C1RdlReport allows you to generate RDL reports that can consume any data source (such as .mdb files) – not only SQL server data as Microsoft Reporting Services. The C1RdlReport component supports RDL files based on the RDL 2008 specifications.

This assembly does not reference other ComponentOne DLL files. Most of the classes in this assembly are in the C1.C1Preview, C1.C1Report, and C1.C1Rdl.Rdl2008 namespaces.

### C1.Win.C1Report.2.dll

**C1.Win.C1Report.2.dll** provides Windows Forms controls that can work with C1PrintDocument and other types of documents, including:

- C1PreviewPane

  The preview pane. This control shows the pages of the document being previewed, allows panning, zooming and other preview operations. In the forms designer, standard toolbars and status bar can be created on the current form via context menu items.

- C1PrintPreviewControl

  The integrated print preview control. Contains the preview pane, toolbars with the standard preview related operations, navigation panel with thumbnails and outline pages, and the collapsible text search panel.

- C1PrintPreviewDialog

  A dialog box form with the nested print preview control.

- C1PreviewThumbnailView

  Panel that can be attached to a preview pane (via the PreviewPane property) to show the navigation page thumbnails.

- C1PreviewOutlineView

  Panel that can be attached to a preview pane (via the PreviewPane property) to show the document outline.

- C1PreviewTextSearchPanel

  Panel that can be attached to a preview pane (via the PreviewPane property) to perform text search in the document.

This assembly references C1.C1PrintDocument. Most of the classes in this assembly are in the C1.Win.C1Preview namespace.

**Included Applications**

In addition to the reporting components and controls included, **Reports for WinForms** includes two stand-alone applications, **C1ReportDesigner** and **C1ReportsScheduler**:

- **C1ReportDesigner**

  The **C1ReportDesigner** application is a tool used for creating and editing C1Report report definition files. The Designer allows you to create, edit, load, and save files (XML) that can be read by the C1Report component.

- **C1ReportsScheduler**

  **C1ReportsScheduler** is a stand-alone application used to schedule report creation to run in the background. Using the **C1ReportsScheduler** application, you can choose what reports to export or print, formats to export the report to, and the schedule and frequency for creating reports.

This product requires .NET version 2.0 or later.

# Getting Started with Reports for WinForms

The current release of **ComponentOne Reports for WinForms** includes a new component, C1.C1Report.C1Report. This component provides a complete compatible replacement for C1.Win.C1Report.C1Report found in previous releases. The only difference, from the user code point of view, is the namespace change (C1.C1Report instead of C1.Win.C1Report). The C1.C1Report namespace also provides all other familiar public **C1Report** classes such as Field, Section, and so on. Again, the only expected difference affecting the user is the namespace change.

Internally, the new **C1Report** works differently from the old version. Whereas the old **C1Report** engine in the usual (preview/print) scenario generated metafile page images, the new **C1Report** builds a **C1PrintDocument** representing the report. That document is accessible via the new public read-only property on **C1Report**: `C1PrintDocument C1Report.C1Document {get;}`.

The document can then be used in any of the usual ways; for example, it can be exported to one of the formats supported by **C1PrintDocument**.

**Export filters:**

Along with other public members exposed by the old version, the new **C1Report** provides the familiar WinForms **C1Report's** export filters, so the following code is still completely valid:

```
report.Load(...);
report.RenderToFile("MyReport.rtf",C1.C1Report.FileFormatEnum.RTF);
```

It is important to note that the file produced by the code above (in our example, an RTF file) will differ from the file produced by exporting the **C1PrintDocument** exposed by the report:

```
report.Load(...);
report.C1Document.Export("MyReport.rtf");
```

Usually, the RenderToFile would yield better results, unless the target format is a fixed layout format (such as PDF), in which case the results should be identical.

Note also that the RenderToFile method does not support the new XPS format, the only way to generate an XPS file is to export the **C1PrintDocument** exposed by **C1Report**.

## Generating Reports (C1Report vs. C1PrintDocument)

The **Reports for WinForms** assembly now provides two distinctly different methods for generating reports:

- Using the C1PrintDocument.ImportC1Report method.
- Using the C1Report component.

While many existing reports can be correctly generated using either method, there are some important differences between the two. Some of the key differences are listed below.

**Data binding:**

The ImportC1Report method creates a data-bound C1PrintDocument component. The C1PrintDocument component is then generated with data being fetched. The resulting document can be refreshed with data refreshing. When the C1Report component is used, the generated document already contains embedded data fetched during document generation, and the resulting document is not data-bound (but of course, the report can be rendered again, refreshing the data).

**Document structure:**

When a report is imported in the resulting C1PrintDocument structure, all fields are represented by RenderField, and all sections by RenderSection objects. If a report definition contains groups, each group is represented by a RenderArea object, with nested RenderSection objects for the group header and footer, and a RenderArea for the nested group.

When the **C1Report** component is used, each report section is rendered into a RenderArea object. The fields are rendered as follows:

- If **LineSlant** is specified, a RenderLine is generated;
- If image or barcode is specified, a RenderImage is generated;
- If **RTF** is **True**, a RenderRichText is generated;
- In all other cases, a RenderText is generated.

**Page size:**

When a report with unset (or set to zeros) CustomWidth and CustomHeight properties is imported on a system without a printer, the default page size depends on the locale, as always in C1PrintDocument (for example, for US and Canada, the page size will be Letter, for Russia A4, and so on).

When such report is loaded into the C1Report component, the page size is always set to Letter (8.5 x 11 inches), which is the behavior of C1.Win.C1Report.C1Report.

**Default printer:**

On a system with one or more printers installed, the default page size for an imported report is determined using the same logic as for a regular **C1PrintDocument** (in particular, the MeasurementPrinterName property is used to determine which printer to use). The main reason for this behavior is to avoid long wait times on systems with default network printers. When the **C1Report** component is used, the default page size is determined by the system default printer.

**Limitations of import:**

The import method has some inherent limitations which are not likely to ever be lifted. These limitations include:

- When import is used, the **C1Report** object model is not available and report events are not invoked. Hence the main limitation of import: reports depending on C#/VB event handlers cannot be rendered correctly when imported.
- Scripting limitations:
  - **C1PrintDocument object:**
    - The **Font** property is read-only.
  - **Field object:**
    - Section, Font, and Subreport properties are read-only;
    - **LineSpacing**, **SubreportHasData**, and **LinkValue** properties are not supported; if the LinkTarget property contains an expression, it is not evaluated and will be inserted into the generated report as-is.
  - **Layout object:**
    - The **ColumnLayout** property is not supported, columns always go top to bottom and left to right.
    - **LabelSpacingX**, **LabelSpacingY**, and **OverlayReplacements** are not supported.
    - **ForcePageBreak** cannot be used in the **OnPrint** handler.
- The dialog box is not shown for parameterized reports. Instead, default values depending on the parameter type are used (0 for numbers, empty string for strings, the current date for dates and so on).
- **PageHeader/PageFooter** cannot reference database fields.
- In multi-column reports, the report header is printed only above the first column (in **C1Report**, the header spans all columns).
- Left to right orientation for columns is not supported – columns are always oriented from top to bottom.
- In **C1Report**, a Field containing an image, with CanGrow set to **True** and PictureScale set to **Clip**, has the width equal to that of the image. When imported, the width of such a field is scaled proportionally to the width of the image.

### *Deciding on Report Generation Method*

Because two ways of generating reports are available (using the **C1Report** component vs. importing the report into a **C1PrintDocument**), you may ask, "Which method is preferable?" Our recommendations are as follows:

- If any of the limitations of import (see Generating Reports (C1Report vs. C1PrintDocument) for a list of limitations) are critical to your application, use the **C1Report** component.
- If you are a user of a previous version of **C1Report** and are not familiar with the **C1PrintDocument** object model, you may still continue to use the **C1Report** component provided by **C1Preview**.
- If, on the other hand, you have some experience with **C1PrintDocument**, or are starting a new project, using import is the preferred approach, due to the following considerations:
    - **C1PrintDocument** integration: when a report definition has been imported into a **C1PrintDocument**, the resulting document can be manipulated as any other **C1PrintDocument**. For example, user code can add content to the document body, modify document properties, and so on. Such changes will persist even when the document is refreshed.
    - Some problems existing in **C1Report** are solved by import; specifically, in C1Report side-by-side objects cannot be correctly split between pages, and borders are not rendered correctly on objects split between pages. Neither of these problems exists when a report is imported into **C1PrintDocument**.
    - Import is slightly more efficient both memory- and speed-wise.
    - Future enhancements: it is likely that some future enhancements will affect only import.

## Getting Started with Reporting

In this section you will learn how to use the basic C1Report functionality to create simple reports. This section is not supposed to be a comprehensive tutorial on all features of C1Report, but rather provide a quick start and highlight some general approaches to using the component.

### *C1Report Quick Start*

Although you can use C1Report in many different scenarios, on the desktop and on the Web, the main sequence of steps is always the same:

1. **Create a report definition**

   This can be done directly with the **C1Report Designer** or using the report designer in Microsoft Access and then importing it into the **C1Report Designer**. You can also do it using code, either using the object model to add groups and fields or by writing a custom XML file.

2. **Load the report into the C1Report component**

   This can be done at design time, using the **Load Report** context menu, or programmatically using the **C1Report.Load** method. If you load the report at design time, it will be persisted (saved) with the control and you won't need to distribute the report definition file.

3. **Render the report (desktop applications)**

   If you are writing a desktop application, you can render the report into a **C1PrintPreview** control (or a **Microsoft PrintPreview control**) using the **C1Report.Document** property. The preview control will display the report on the screen, and users will be able to preview it with full zooming, panning, and so on.

4. **Render the report (Web applications)**

   If you are writing a Web application, you can render reports into HTML or PDF files using the **C1Report.RenderToFile** method, and your users will be able to view them using any browser.

The following steps will show you how to create a report definition, load the report into the C1Report component, and render the report.

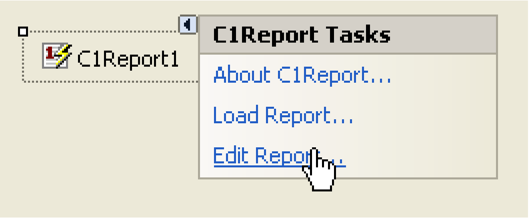
## Step 1 of 4: Creating a Report Definition

The following topic shows how you can create a report definition using the **C1ReportDesigner**. Note that creating a report definition is not the same as rendering a report. To render a report, you can simply load an existing definition and call the **C1Report.Render** method. The easiest way to create a report definition is to use the **C1ReportDesigner**, which is a stand-alone application similar to the report designer in Microsoft Access.

> **Note:** You can also create reports from scratch, using code. This approach requires some extra work, but it gives you complete flexibility. You can even write your own report designer or ad-hoc report generator. For details, see Creating a Report Definition.

The **C1Report Wizard** walks you through the steps of creating a new report from start to finish. To begin, complete the following steps:

1. To begin, create a .NET project and add the **C1Report** component to your Toolbox. For more information on creating a new project, see Creating a .NET Project.
2. From the Toolbox, double-click the **C1Report** icon to add the component to your project. Note that the component will appear below the form in the Component Tray.
3. Click the **C1Report** component's smart tag and select **Edit Report** from its Tasks menu.



The **C1ReportDesigner** opens and the **C1Report Wizard** is ready to guide you through five easy steps.

From the **C1Report Wizard**, complete the five following steps to create your report:

1. **Select the data source for the new report.**

   Use this page to select the **DataSource.ConnectionString** and **DataSource.RecordSource** that will be used to retrieve the data for the report.

   You can specify the **DataSource.ConnectionString** in three ways:

   - Type the string directly into the editor.
   - Use the drop-down list to select a recently used connection string (the Designer keeps a record of the last eight connection strings).
   - Click the **ellipsis** button **(...)** to bring up the standard connection string builder.

   You can specify the **DataSource.RecordSource** string in two ways:

   - Click the **Table** option and select a table from the list.
   - Click the **SQL** option and type (or paste) an SQL statement into the editor.

   **Complete Step 1:**

   Complete the following steps:

Click the **ellipsis** button to bring up the standard connection string builder.

Select the **Provider** tab and select a data provider from the list. For this example, select **Microsoft Jet 4.0 OLE DB Provider**.

Click the **Next** button or select the **Connection** tab. Now you must choose a data source.

To select a database, click the **ellipsis** button. The **Select Access Database** dialog box appears. For this example, select the **Nwind.mdb** located in the **Common** folder in the **ComponentOne Samples** directory (by default installed in the **Documents** or **My Documents** folder). Note that this directory reflects the default installation path and its path may be different if you made changes to the installation path.

Click **Open**. You can test the connection and click **OK**.

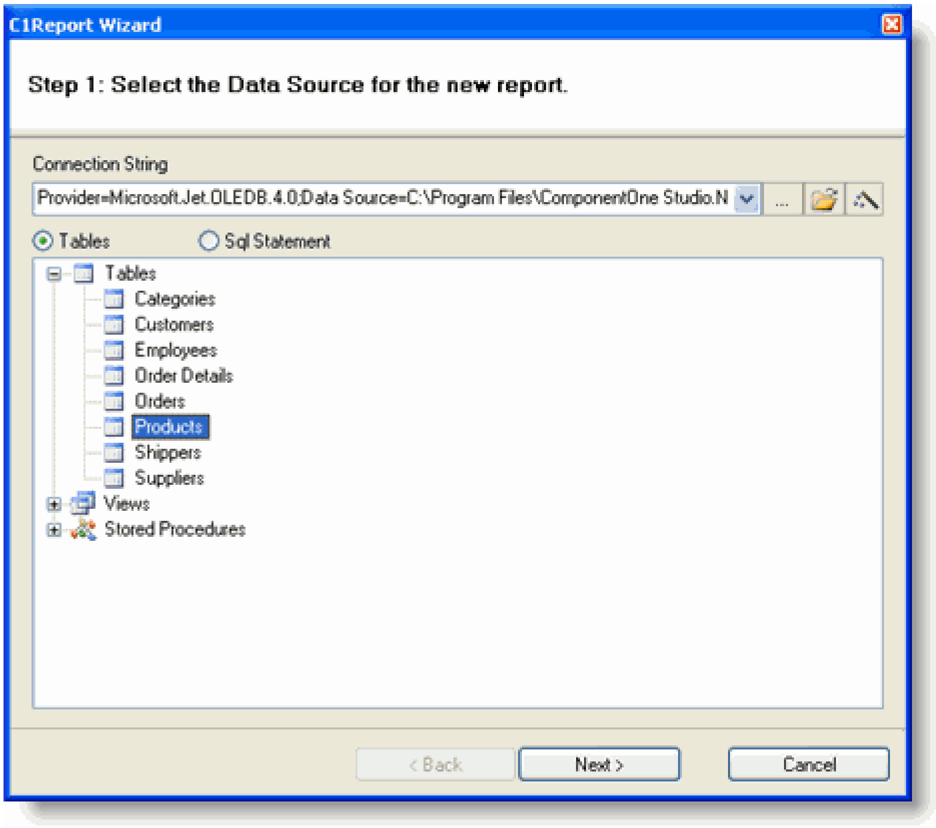
Click **OK** to close the **Select Access Database** dialog box.

Once you have selected your data source, you can select a table, view, or stored procedure to provide the actual data. You can specify the **DataSource.RecordSource** string in two ways:

- Click the **Table** option and select the **Products** table from the list.
- Click the **SQL** option and type (or paste) an SQL statement into the editor.

  For example:

  ```
  select * from products
  ```



C1Report Wizard

Step 1: Select the Data Source for the new report.

Connection String: Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\ComponentOne Studio.N

● Tables ○ Sql Statement

Click **Next**. The wizard will walk you through the remaining steps.

2. **Select the fields you want to include in the report.**

   This page contains a list of the fields available from the recordset you selected in Step 1, and two lists that define the group and detail fields for the report. Group fields define how the data will be sorted and summarized, and detail fields define what information you want to appear in the report.

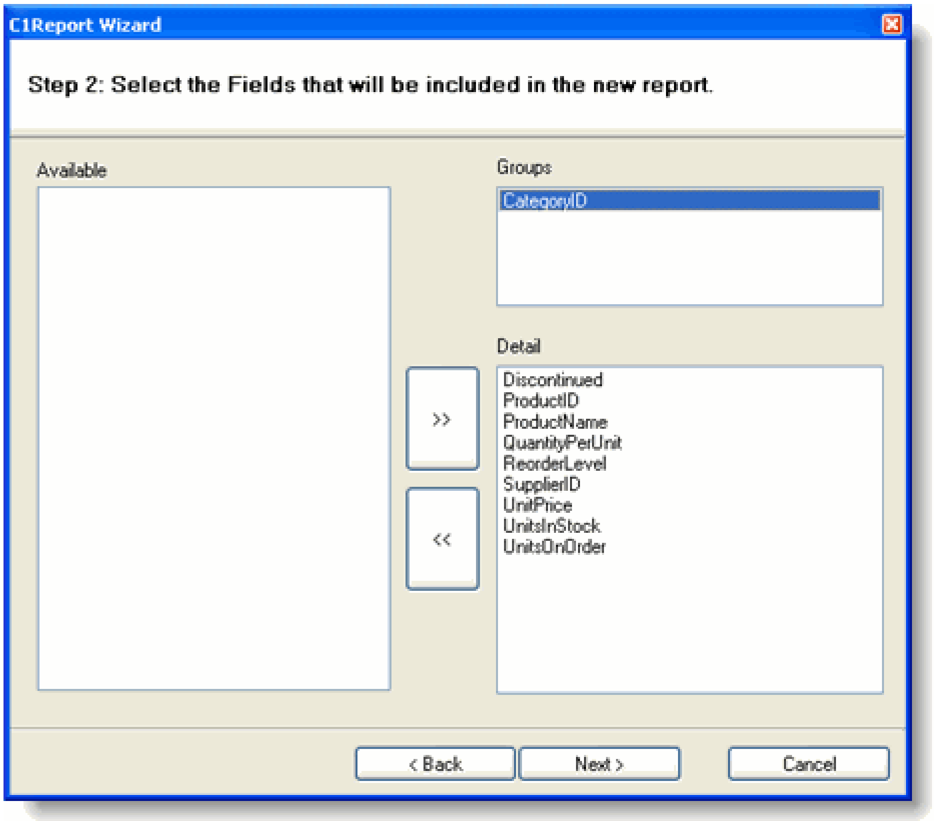
   You can move fields from one list to another by dragging them with your mouse pointer. Drag fields into the **Detail** list to include them in the report, or drag within the list to change their order. Drag fields back into the **Available** list to remove them from the report.

**Complete Step 2:**

Complete the following steps:

a. With your mouse pointer, select the **CategoryID** field and drag it into the **Groups** list.

Press the >> button to move the remaining fields into the **Detail** list.



Click **Next**. The wizard will walk you through the remaining steps.

3. **Select the layout for the new report.**

This page offers you several options to define how the data will be organized on the page. When you select a layout, a thumbnail preview appears on the left to give you an idea of what the layout will look like on the page. There are two groups of layouts, one for reports that have no groups and one for reports with groups. Select the layout that best approximates what you want the final report to look like.

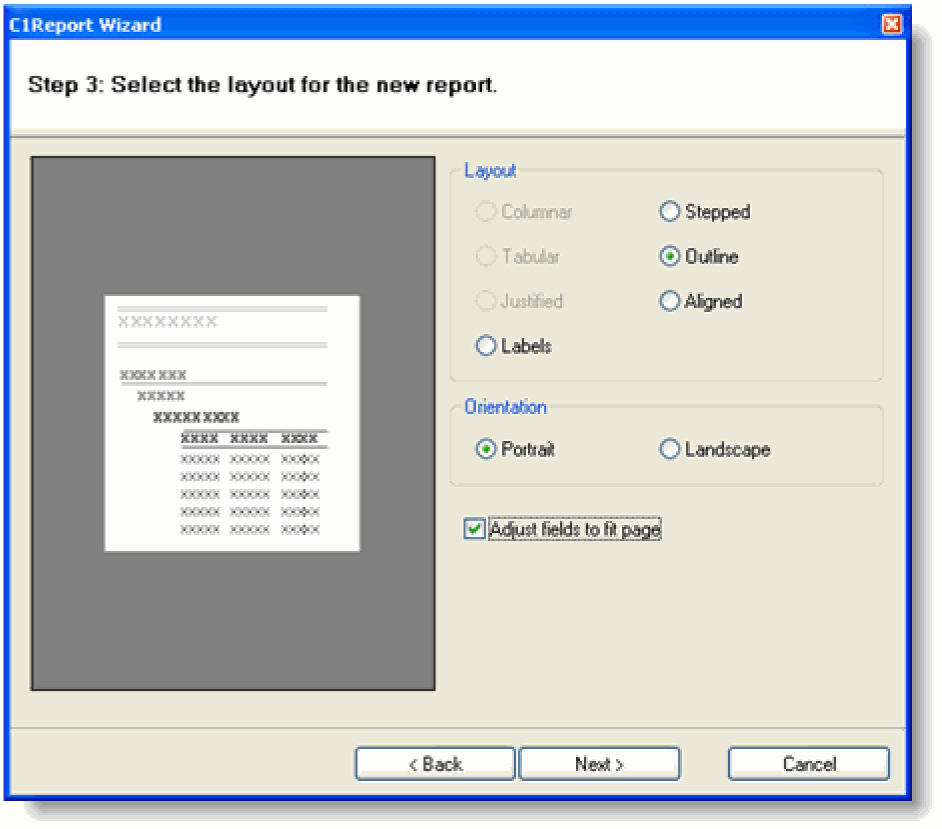
This page also allows you to select the page orientation and whether fields should be adjusted to fit the page width.

The **Labels** layout option is used to print Avery-style labels. If you select this option, you will see a page that prompts you for the type of label you want to print.

**Complete Step 3:**

Complete the following steps:

a. Keep the **Outline** layout.



C1Report Wizard

Step 3: Select the layout for the new report.

Layout

○ Columnar ○ Stepped

○ Tabular ● Outline

○ Justified ○ Aligned

○ Labels

Orientation

● Portrait ○ Landscape

☑ Adjust fields to fit page

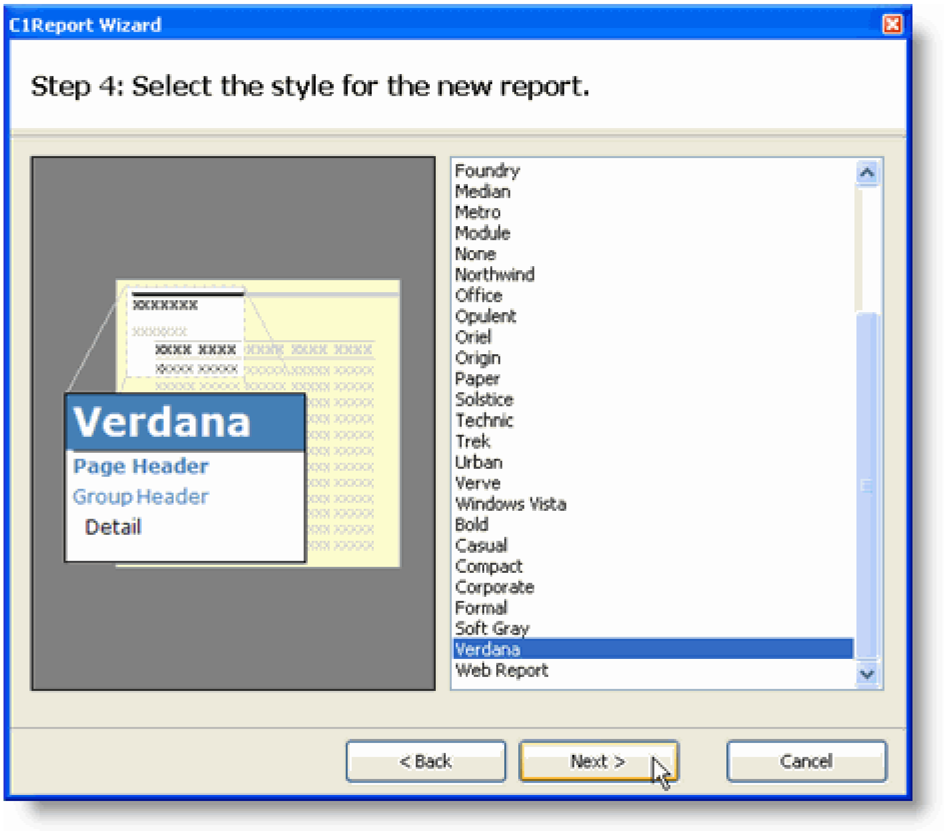
Click **Next**. The wizard will walk you through the remaining steps.

4. **Select the style for the new report.**

   This page allows you to select the fonts and colors that will be used in the new report. Like the previous page, it shows a preview to give you an idea of what each style looks like. Select the one that you like best (and remember, you can refine it and adjust the details later).

   **Complete Step 4:**

   a. Select the **Verdana** style.



Step 4: Select the style for the new report.

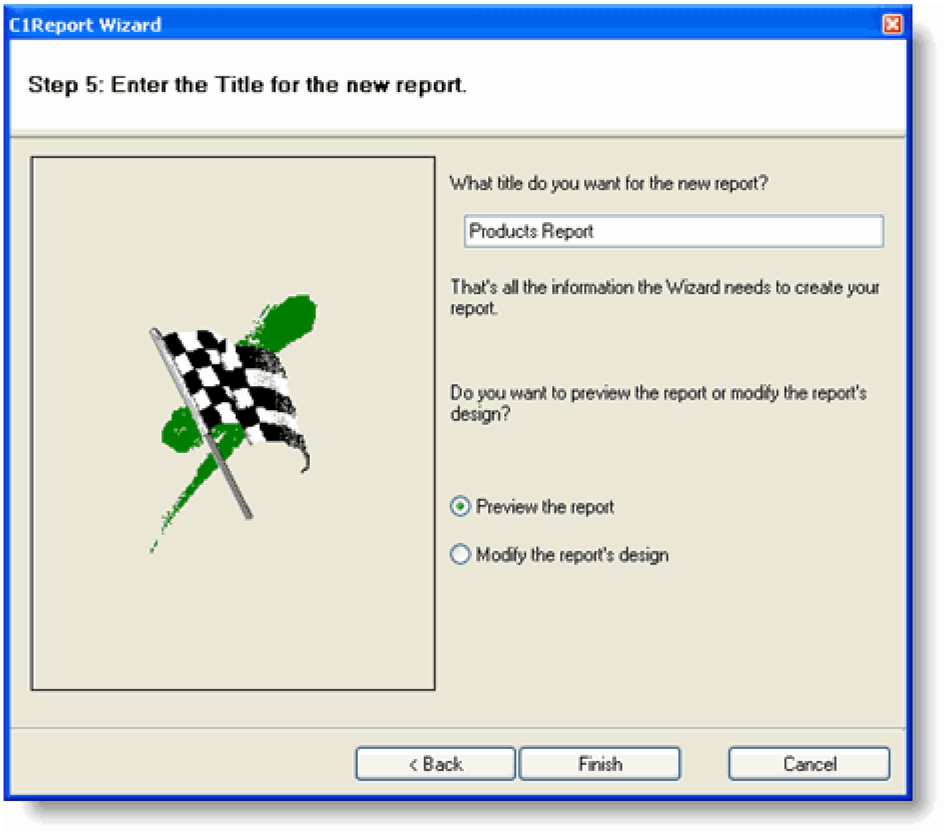
Click **Next**. The wizard will walk you through the remaining steps.

5. **Select a title for the new report.**

   This last page allows you to select a title for the new report and to decide whether you would like to preview the new report right away or whether you would like to go into edit mode and start improving the design before previewing it.

   **Complete Step 5:**

   a. Enter a title for the new report, **Products Report**, for example.



Step 5: Enter the Title for the new report.

What title do you want for the new report?

Products Report

That's all the information the Wizard needs to create your report.

Do you want to preview the report or modify the report's design?

● Preview the report

○ Modify the report's design

Choose to **Preview the report** and click **Finish**.

You will immediately see the report in the preview pane of the **Designer**.

You will notice that the report will require some adjustments. In the next step, you will learn how to modify the report.

## Step 2 of 4: Modifying the Report

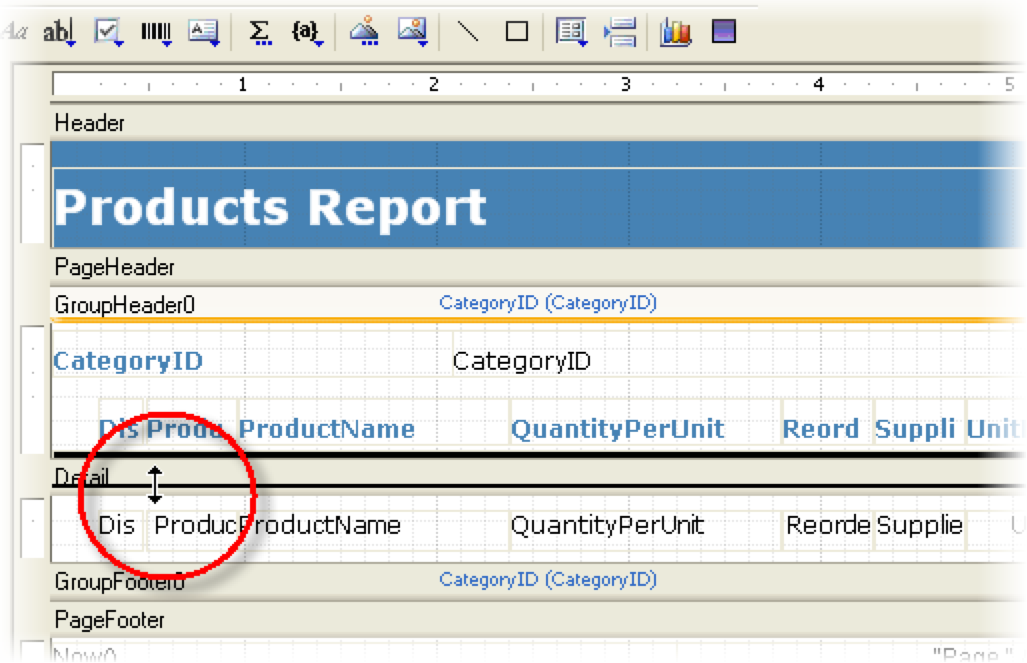
With the **Designer** in preview mode you cannot make any adjustments to the report.

1. Click the **Close Print Preview** button in the **Close Preview** tab to switch to Design mode and begin making modifications.
2. After selecting the **Close Print Preview** button, the right pane of the main window switches from Review mode into Design mode, and it shows the controls and fields that make up the report.

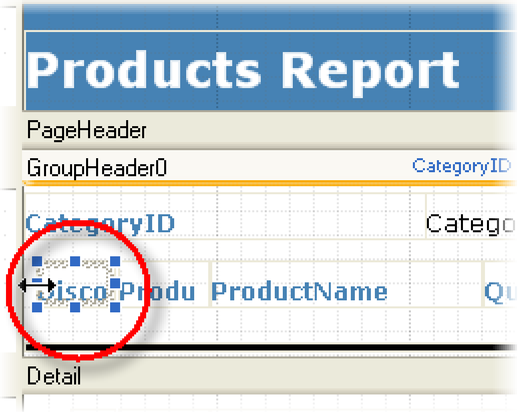
**Modify the Report:**

For this example, we will resize the Group Header section and fields as well as format a field value. To do this, complete the following steps:

1. To resize the Group Header section, select its border and with your mouse pointer drag to the position where you want it.

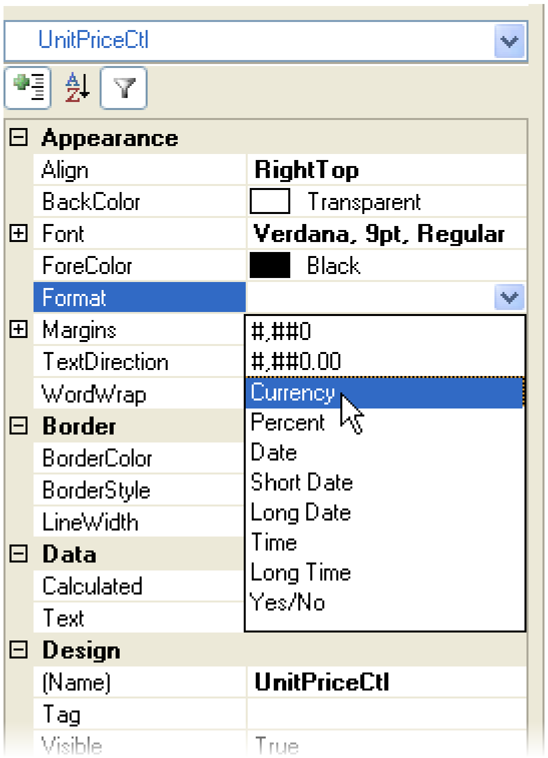


2. With your mouse pointer, drag field corners to resize fields.



**Tip:** If text is not fitting in the field, set the **Appearance.WordWrap** property for the field to **True** in the Properties window.

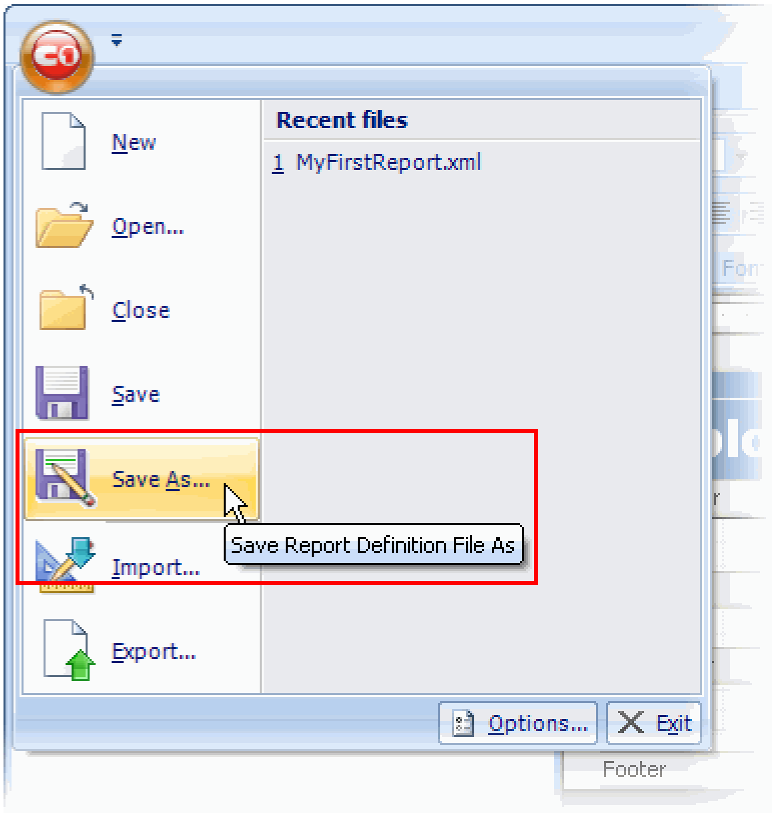
3. In the Detail section under the *Unit Price* column, select the **UnitPriceCtl** field.
4. In the Properties window, set the Apperance.**Format** property for the field to **Currency**.



UnitPriceCtl

| | |
|---|---|
| ⊟ **Appearance** | |
| Align | **RightTop** |
| BackColor | □ Transparent |
| ⊞ Font | **Verdana, 9pt, Regular** |
| ForeColor | ■ Black |
| Format | |
| ⊞ Margins | |
| TextDirection | |
| WordWrap | |
| ⊟ **Border** | |
| BorderColor | |
| BorderStyle | |
| LineWidth | |
| ⊟ **Data** | |
| Calculated | |
| Text | |
| ⊟ **Design** | |
| (Name) | **UnitPriceCtl** |
| Tag | |
| Visible | True |

5. Click the **Preview** button to switch to Preview mode and see your modifications.
6. Click **Close Print Preview** in the **Close Preview** tab to close the preview and switch to **Design** view.
7. Click the **Application** button and select **Save As** from the menu that appears.



8. In the **Save Report Definition File** dialog box, enter **ProductsReport.xml** in the **File name** box. Save the file to a location that you will remember for later use.
9. Close the **Designer** and return to your Visual Studio .NET project.

You have successfully created a report definition file; in the next step you will load the report in the **C1Report** component.

## Step 3 of 4: Loading the Report in the C1Report Component

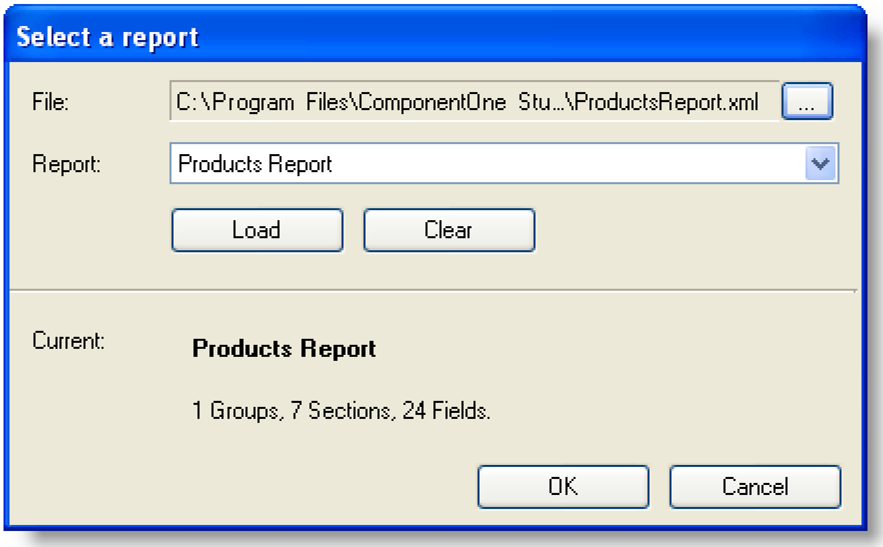
To load a report definition from a file at design time, complete one of the following tasks:

- Right-click the **C1Report** component and select the **Load Report** menu option.

  OR
- Click the smart tag (▶) above the **C1Report** component and select **Load Report** from the **C1Report Tasks** menu.

Using the **Select a report** dialog box to select the report you want, complete the following tasks:

1. Click the **ellipsis** button. The **Open** dialog box appears.
2. Browse to the location that you just saved your **ProductsReport.xml** file, select it, and click **Open**.
3. The available report definitions are listed in the **Report** drop-down box. Select the **Products Report** definition to load.
4. Click **Load** and **OK** to close the dialog box.



File: C:\Program Files\ComponentOne Stu...\ProductsReport.xml

Report: Products Report

Current: **Products Report**

1 Groups, 7 Sections, 24 Fields.

In the next step you will render the report into a preview control.

## Step 4 of 4: Rendering the Report

Once the report definition has been created, a data source defined, and loaded into the **C1Report** component, you can render the report to the printer, into preview controls, or to report files.

To preview the report, use the use the C1Report.**Document** property. Assign it to the **Document** property in the **C1PrintPreviewControl** or to the .NET **PrintPreview** or **PrintPreviewDialog** controls and the preview controls will display the report and allow the user to browse, zoom, or print it.

> **Note: C1Report** works with the .NET preview components, but it is optimized to work with the included **Reports for WinForms** preview controls. When used with the included controls, you can see each report page as it is generated. With the standard controls, you have to wait until the entire report is ready before the first page is displayed.

To complete this step:

1. From the Toolbox, double-click the **C1PrintPreviewControl** icon to add the component to your project.
2. From the Properties window, set the **C1PrintPreviewControl.Dock** property to **Fill**.
3. Select the Windows Form with your mouse and drag to resize it. For this example, we resized the Form to **600x500** so it better reveals the preview panel.
4. Double-click the form and enter the following code in the **Form_Load** event handler:

   - Visual Basic

```
' load report definition
C1Report1.Load("C:\ProductsReport.xml", "Products Report")

' preview the document
C1PrintPreviewControl1.Document = C1Report1.Document
```

   - C#

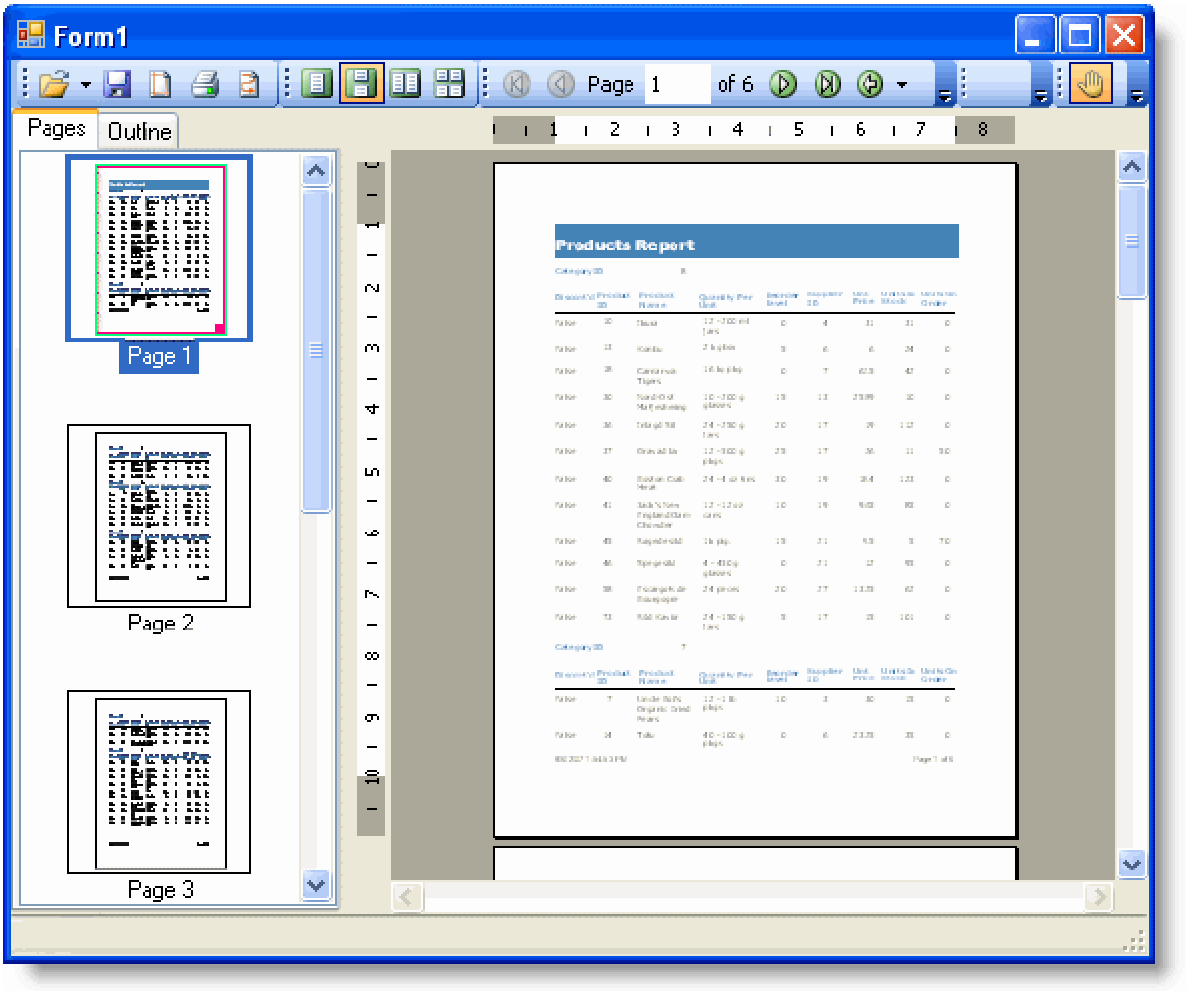
```
// load report definition
c1Report1.Load(@"C:\ProductsReport.xml", "Products Report");

// preview the document
c1PrintPreviewControl1.Document = c1Report1.Document;
```

You will need to change the directory above to the location the **ProductsReport.xml** file is saved.

Note that the **C1Report.Load(String, String)** method has the following parameters:

- *fileName*: Full name of the XML report definition file.
- *reportName*: Name of the report to retrieve from the file (case-insensitive).

5. Click the **Start Debugging** button to run the application. The report renders in the preview control:



Congratulations! You just created a simple report definition, modified the report, loaded into the **C1Report** component, and rendered the report into a preview control. Read the following section to learn how you can further customize your report definition using VBScript expressions.

## Creating VBScript Expressions

Expressions are widely used throughout a report definition to retrieve, calculate, display, group, sort, filter, parameterize, and format the contents of a report. Some expressions are created for you automatically (for example, when you drag a field from the Toolbox onto a section of your report, an expression that retrieves the value of that field is displayed in the text box). However, in most cases, you create your own expressions to provide more functionality to your report.

**C1Report** relies on VBScript to evaluate expressions in calculated fields and to handle report events.

VBScript is a full-featured language, and you have access to all its methods and functions when writing **C1Report** expressions. For the intrinsic features of the VBScript language, refer to the Microsoft Developer's Network (MSDN). **C1Report** extends VBScript by exposing additional objects, variables, and functions.

The following topics demonstrate how you can create your own expressions to provide more functionality to your report.

## Formatting a Field According to Its Value

Formatting a field according to its value is probably the most common use for the **Section.OnPrint** property. Take for example a report that lists order values grouped by product. Instead of using an extra field to display the quantity in stock, the report highlights products that are below the reorder level by displaying their name in bold red characters.

**To highlight products that are below the reorder level using code:**

To highlight products that are below the reorder level by displaying their name in bold red characters, use an event script that looks like this:

- Visual Basic

```
Dim script As String = _
  "If UnitsInStock < ReorderLevel Then" & vbCrLf & _
  "ProductNameCtl.ForeColor = RGB(255,0,0)" & vbCrLf & _
  "ProductNameCtl.Font.Bold = True" & vbCrLf & _
  "Else" & vbCrLf & _
  "ProductNameCtl.ForeColor = RGB(0,0,0)" & vbCrLf & _
  "ProductNameCtl.Font.Bold = False" & vbCrLf & _
  "End If"
c1r.Sections.Detail.OnPrint = script
```

- C#

```
string  script =
  "if (UnitsInStock < ReorderLevel) then\r\n" +
  "ProductNameCtl.ForeColor = rgb(255,0,0)\r\n" +
  "ProductNameCtl.Font.Bold = true\r\n" +
  "else\r\n" +
  "ProductNameCtl.ForeColor = rgb(0,0,0)\r\n" +
  "ProductNameCtl.Font.Bold = false\r\n" +
  "end if\r\n";
c1r.Sections.Detail.OnPrint = script;
```

The code builds a string containing the VBScript event handler, and then assigns it to the section's **Section.OnPrint** property.

**To highlight products that are below the reorder level using the C1ReportDesigner:**

Alternatively, instead of writing the code, you can use the **C1ReportDesigner** to type the following script code directly into the VBScript Editor of the Detail section’s **Section.OnPrint** property. Complete the following steps:

1. Select **Detail** from the Properties window drop-down list in the Designer. This reveals the section's available properties.

2. Click the empty box next to the Section.**OnPrint** property, then click the drop-down arrow, and select **Script Editor** from the list.

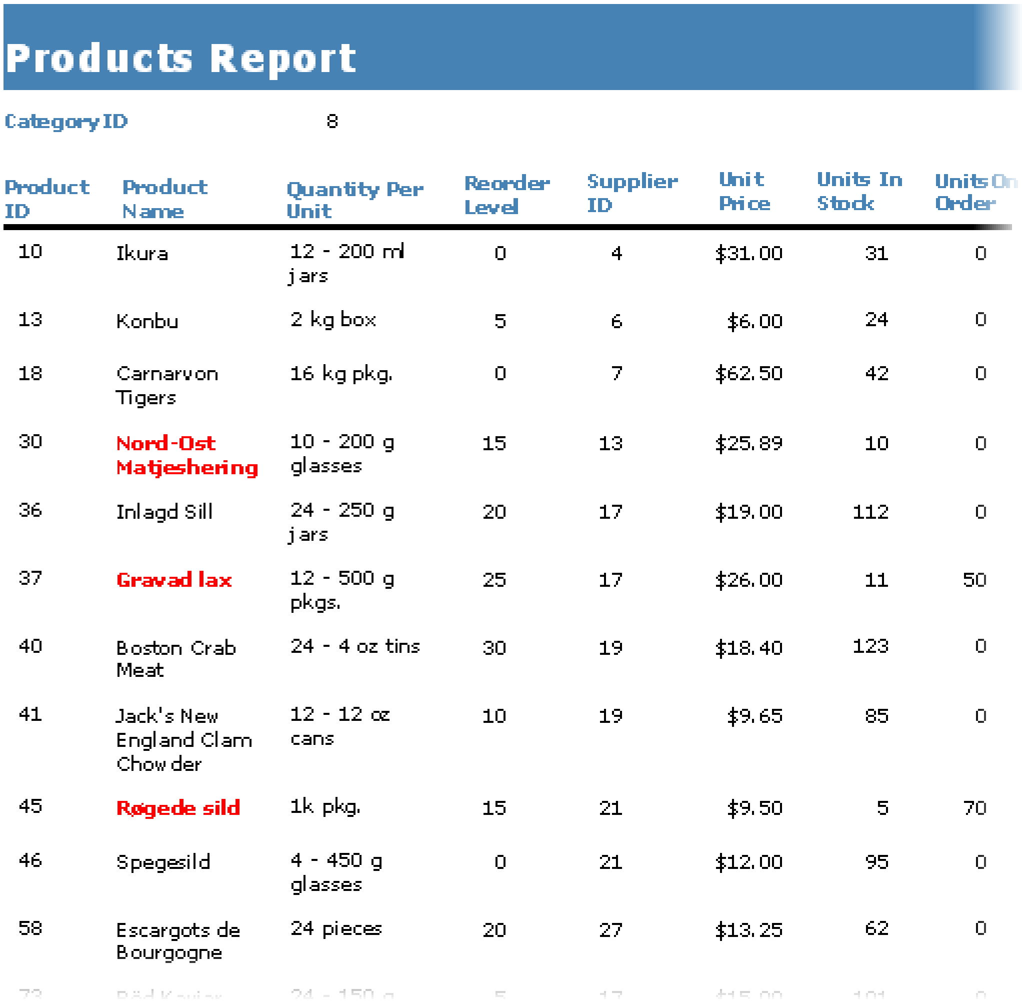
3. In the **VBScript Editor**, simply type the following script in the window:

```
If UnitsInStock < ReorderLevel Then
  ProductNameCtl.ForeColor = RGB(255,0,0)
  ProductNameCtl.Font.Bold = True
Else
  ProductNameCtl.ForeColor = RGB(0,0,0)
  ProductNameCtl.Font.Bold = False
End If
```

4. Click **OK** to close the editor.

The control executes the VBScript code whenever the section is about to be printed. The script gets the value of the "ReorderLevel" database field and sets the "ProductName" report field's Field.Font.**Bold** and Field.**ForeColor** properties according to the value. If the product is below reorder level, its name becomes bold and red.

The following screen capture shows a section of the report with the special effects:



**Products Report**

Category ID 8

| Product ID | Product Name | Quantity Per Unit | Reorder Level | Supplier ID | Unit Price | Units In Stock | Units On Order |
|---|---|---|---|---|---|---|---|
| 10 | Ikura | 12 - 200 ml jars | 0 | 4 | $31.00 | 31 | 0 |
| 13 | Konbu | 2 kg box | 5 | 6 | $6.00 | 24 | 0 |
| 18 | Carnarvon Tigers | 16 kg pkg. | 0 | 7 | $62.50 | 42 | 0 |
| 30 | **Nord-Ost Matjeshering** | 10 - 200 g glasses | 15 | 13 | $25.89 | 10 | 0 |
| 36 | Inlagd Sill | 24 - 250 g jars | 20 | 17 | $19.00 | 112 | 0 |
| 37 | **Gravad lax** | 12 - 500 g pkgs. | 25 | 17 | $26.00 | 11 | 50 |
| 40 | Boston Crab Meat | 24 - 4 oz tins | 30 | 19 | $18.40 | 123 | 0 |
| 41 | Jack's New England Clam Chowder | 12 - 12 oz cans | 10 | 19 | $9.65 | 85 | 0 |
| 45 | **Røgede sild** | 1k pkg. | 15 | 21 | $9.50 | 5 | 70 |
| 46 | Spegesild | 4 - 450 g glasses | 0 | 21 | $12.00 | 95 | 0 |
| 58 | Escargots de Bourgogne | 24 pieces | 20 | 27 | $13.25 | 62 | 0 |
| 73 | Röd Kaviar | 24 - 150 g | 5 | 17 | $15.00 | 101 | 0 |

## Resetting the Page Counter

The **C1Report.Page** variable is created and automatically updated by the control. It is useful for adding page numbers to page headers or footers. In some cases, you may want to reset the page counter when a group starts. For example, in a report that groups records by country. You can do this by adding code or using the Designer.

**Using Code:**

To reset the page counter when a group (for example, a new country) starts, set the PageFooter field's **Text** property. Enter the following code:

- Visual Basic

```
C1Report1.Fields("PageFooter").Text = "[ShipCountry] & "" "" & [Page]"
```

- C#

```
c1Report1.Fields("PageFooter").Text = "[ShipCountry] + "" "" + [Page]";
```

**Using the C1ReportDesigner:**

To reset the page counter when a group (for example, a new country) starts, set the PageFooter field's **Text** property by completing the following steps:

1. Select the PageFooter's page number field from the Properties window drop-down list in the Designer or select the field from the design pane. This reveals the field's available properties.

2. Click the box next to the **Text** property, then click the drop-down arrow, and select **Script Editor** from the list.

3. In the **VBScript Editor**, simply type the following script in the window:
   ```
   [ShipCountry] & " -  Page " & [Page]
   ```

4. Click **OK** to close the editor.

## Hiding a Section if There is No Data for It

You can change a report field's format based on its data by specifying an expression for the Detail section's **Section.OnFormat** property.

For example, your Detail section has fields with an image control and when there is no data for that record's image you want to hide the record. To hide the Detail section when there is no data, in this case a record's image, add the following script to the Detail section's **Section.OnFormat** property:

```
If isnull(PictureFieldName) Then
Detail.Visible = false
Else
Detail.Visible = true
End If
```

**To hide a section if there is no data for it using code:**

To hide a section if there is no data, in this case a record's image, for it, use an event script that looks like this:

- Visual Basic
  ```
  C1Report1.Sections.Detail.OnPrint = "Detail.Visible = not
  isnull(PictureFieldName)"
  ```

- C#
  ```
  c1Report1.Sections.Detail.OnPrint = "Detail.Visible = not
  isnull(PictureFieldName)";
  ```

**To hide a section if there is no data for it using the C1ReportDesigner:**

Alternatively, instead of writing the code, you can use the **C1ReportDesigner** to type the following script code directly into the VBScript Editor of the Detail section's **OnFormat** property. Complete the following steps:

1. Select **Detail** from the Properties window drop-down list in the Designer. This reveals the section's available properties.

2. Click the empty box next to the **Section.OnFormat** property, then click the drop-down arrow, and select **Script Editor** from the list.

3. In the **VBScript Editor**:

   - Simply type the following script in the window:
     ```
     If isnull(PictureFieldName) Then
     Detail.Visible = false
     Else
     Detail.Visible = true
     End If
     ```

   - Or you could use the more concise version:
     ```
     Detail.Visible = not isnull(PictureFieldName)
     ```

# Getting Started with Printing and Previewing

In this section you will learn how to use the basic C1PrintDocument functionality to create simple documents. This section is not supposed to be a comprehensive tutorial on all features of C1PrintDocument, but rather provide a quick start and highlight some general approaches to using the component.

## C1PrintDocument Quick Start

In this quick start guide we'll follow tradition by creating a simple "Hello World!" document. In the following steps, you'll add **Reports for WinForms** printing and previewing controls to a project, set up the preview, and explore some of the run-time interactions available in **Reports for WinForms**' previewing controls.

### *Step 1 of 4: Adding Previewing Controls to the Form*

In this step you'll add **Reports for WinForms** controls to the form and set the form up to create a simple "Hello World!" document preview. The simplest possible document is one that prints the sentence "Hello, World!". Complete the following steps to set up the form to create such a document:

> **Note:** The sample code fragments in this topic assume that the "using C1.C1Preview;" directive (in C# syntax; or an equivalent for other languages) has been inserted into the file, so that instead of fully qualified type names (such as C1.C1Preview.RenderText) we can use just the class name part (RenderText).

1. Create a new .NET Windows application, and name it **HelloWorld**. For information about creating a new project, see Creating a .NET Project.
2. Navigate to the Toolbox and double-click **C1PrintPreviewControl** to add it to your form.

   A print preview control called **C1PrintPreviewControl1**and showing a sample document will appear on your form. The **C1PrintPreviewControl** control is a composite control containing a preview pane, navigation and text search panels, and toolbars with predefined buttons.

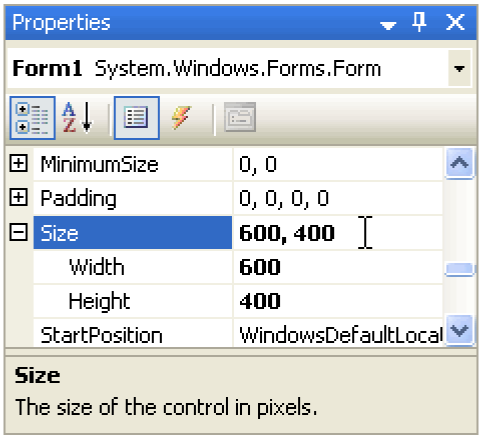
   In addition, two additional entries will appear in the References part of your project: C1.C1Report.2 and C1.Win.C1Report.2.
3. In the Toolbox, double-click the **C1PrintDocument** icon to add the control to your project. The new component will be named **C1PrintDocument1**, by default, and will appear in the component tray below the form.

You've created a project and added **Reports for WinForms** previewing components to your project and completed step 1 of the quick start guide. In the next step you'll set up the form and controls.

### *Step 2 of 4: Setting Up the Form and Controls*

Now that you've added the **Reports for WinForms** controls to the form, you'll set up the form and the controls. Complete the following steps:

1. Right-click **Form1** to select it and select **Properties**; in the Properties window set the **Size.Width** property to **600** pixels and the **Size.Height** property to **400** pixels to accommodate the size of the C1PrintDocument.



Properties

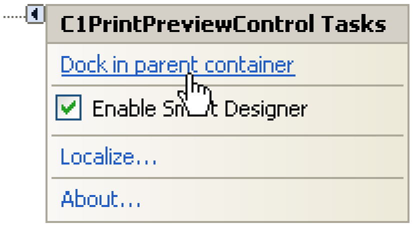
Form1 System.Windows.Forms.Form

| | |
|---|---|
| ⊞ MinimumSize | 0, 0 |
| ⊞ Padding | 0, 0, 0, 0 |
| ⊟ Size | **600, 400** |
| Width | **600** |
| Height | **400** |
| StartPosition | WindowsDefaultLocal |

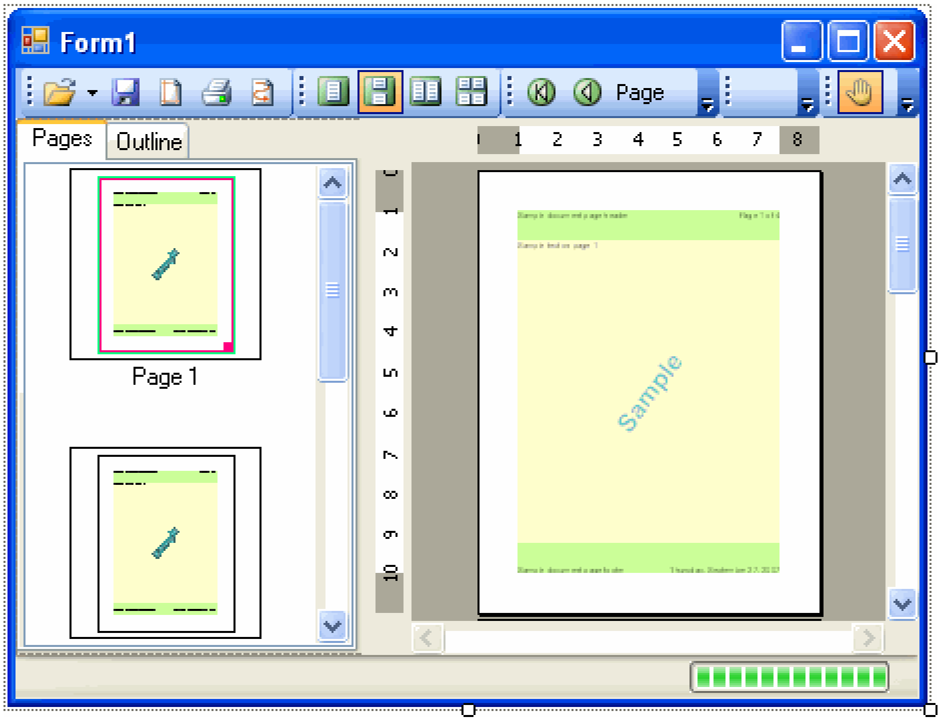
**Size**

The size of the control in pixels.

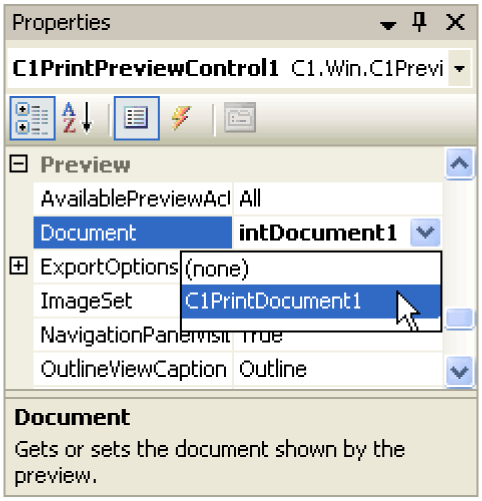
2. Click **C1PrintPreviewControl1**'s smart tag to open the **C1PrintPreviewControl Tasks** menu and select **Dock in parent container**.



The form designer should now look similar to the following:



3. Click **C1PrintPreviewControl1** to select it and in the Properties window set its **Document** property to the newly added **C1PrintDocument1** (which appears in the **Document** property drop-down list).



AvailablePreviewAct All
Document intDocument1
ExportOptions (none)
ImageSet C1PrintDocument1
NavigationPanelVisit True
OutlineViewCaption Outline

**Document**
Gets or sets the document shown by the preview.

The selected document, here **C1PrintDocument1**, will show in the preview window at run time.

You've set up the forms and controls and completed step 2 of the printing and previewing quick start guide. In the next step you'll add code to the project.

### *Step 3 of 4: Adding Code to the Project*

Now that you've added the **Reports for WinForms** controls to the form and customized the form and controls, there's only one last step left before running the project. In this step you'll add code to the project to set the text that appears in the project.

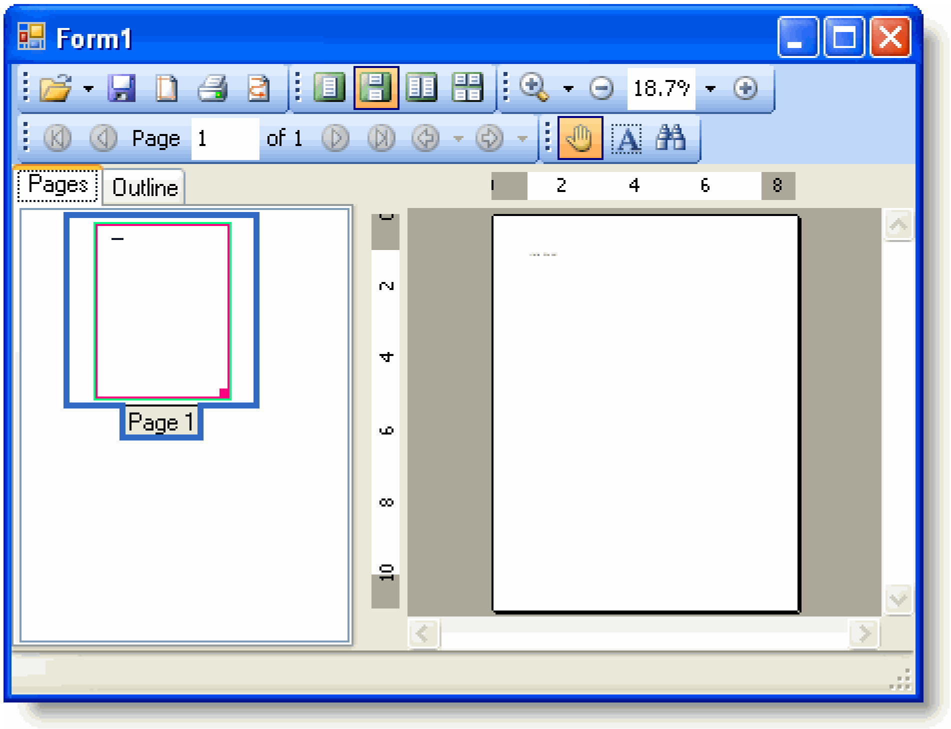
1. Double-click the form's title bar to switch to code view and create a handler for the **Form_Load** event.
2. Add the following code to the **Form_Load** event to add text to the preview document:
   - Visual Basic
     ```
     Me.C1PrintDocument1.Body.Children.Add(New
     C1.C1Preview.RenderText("Hello, World!"))
     ```
   - C#
     ```
     this.c1PrintDocument1.Body.Children.Add(new RenderText("Hello,
     World!"));
     ```
3. Add the following code to the **Form_Load** event to generate the document:
   - Visual Basic
     ```
     Me.C1PrintDocument1.Generate()
     ```
   - C#
     ```
     this.c1PrintDocument1.Generate();
     ```

You've added code to the project and completed step 3 of the printing and previewing quick start guide. In the last step you'll run the program.

### *Step 4 of 4: Running the Program*

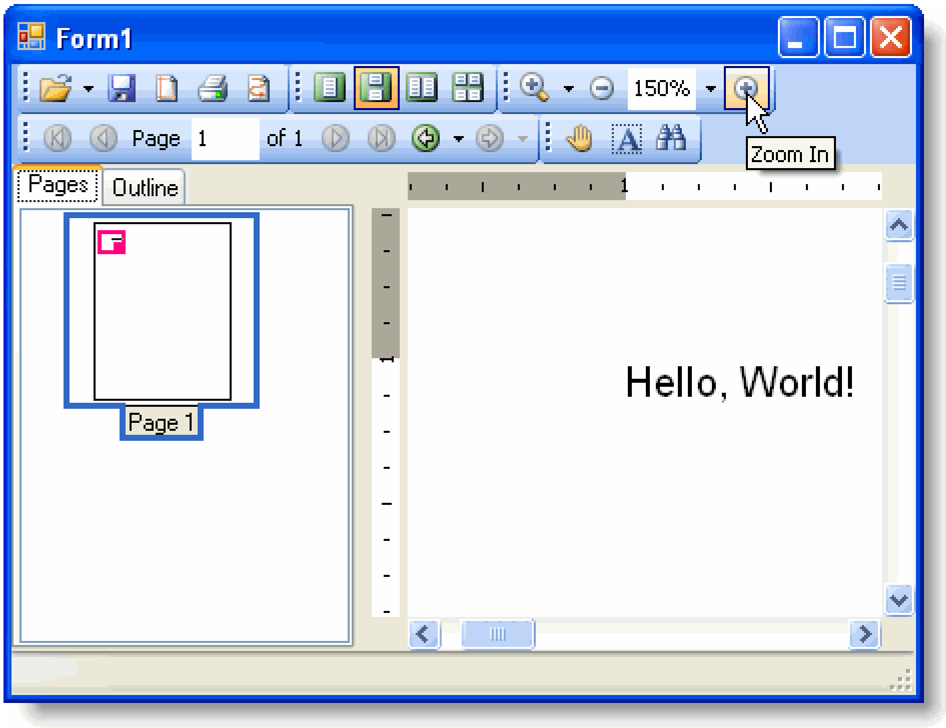
You've set up the form and the **Reports for WinForms** components, set the properties for the components, and added code to the project. All that's left is to run the program and observe some of the run-time interactions available in **Reports for WinForms** previewing controls.

1. Run the program and observe the following:



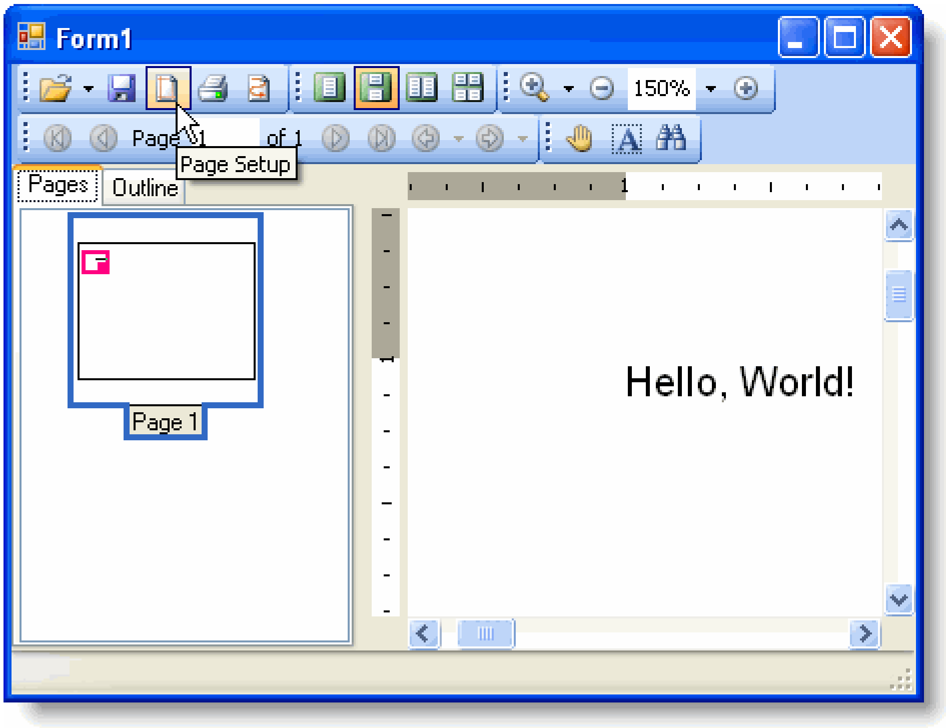
   The preview should show the document consisting of one page, with the words "Hello World!" in the upper left corner of the document.

2. Click the **Zoom In** button and repeat until the document text is more easily visible.



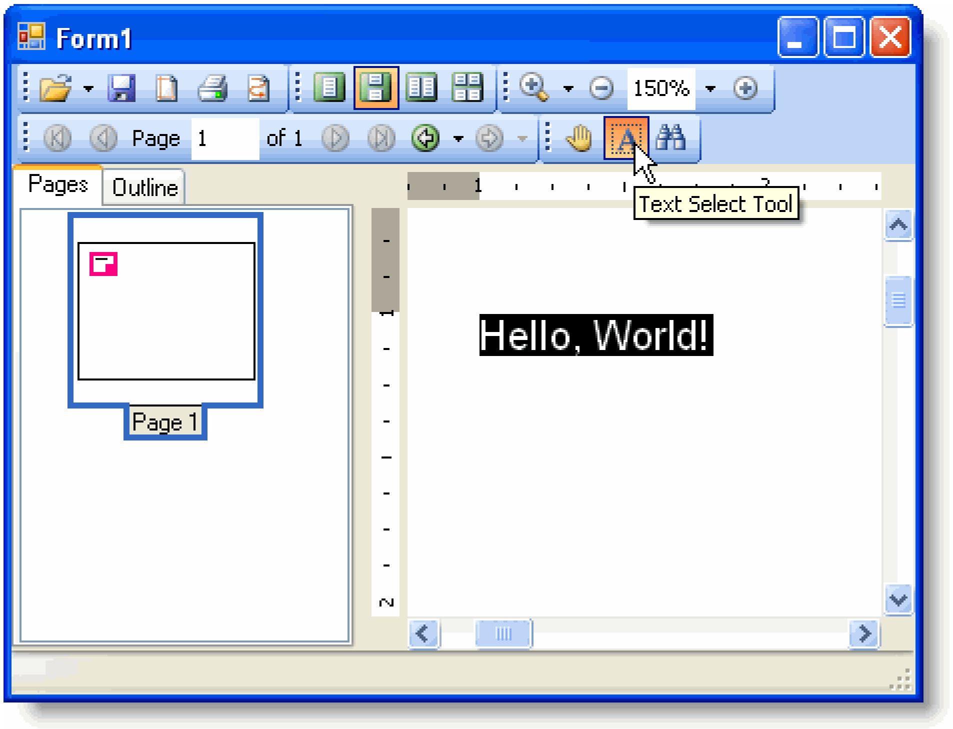
Hello, World!

3. Click the **Page Setup** button to open the **Page Setup** dialog box. In the **Page Setup** dialog box, select **Landscape** under Orientation and click **OK** to apply landscape orientation to the document.

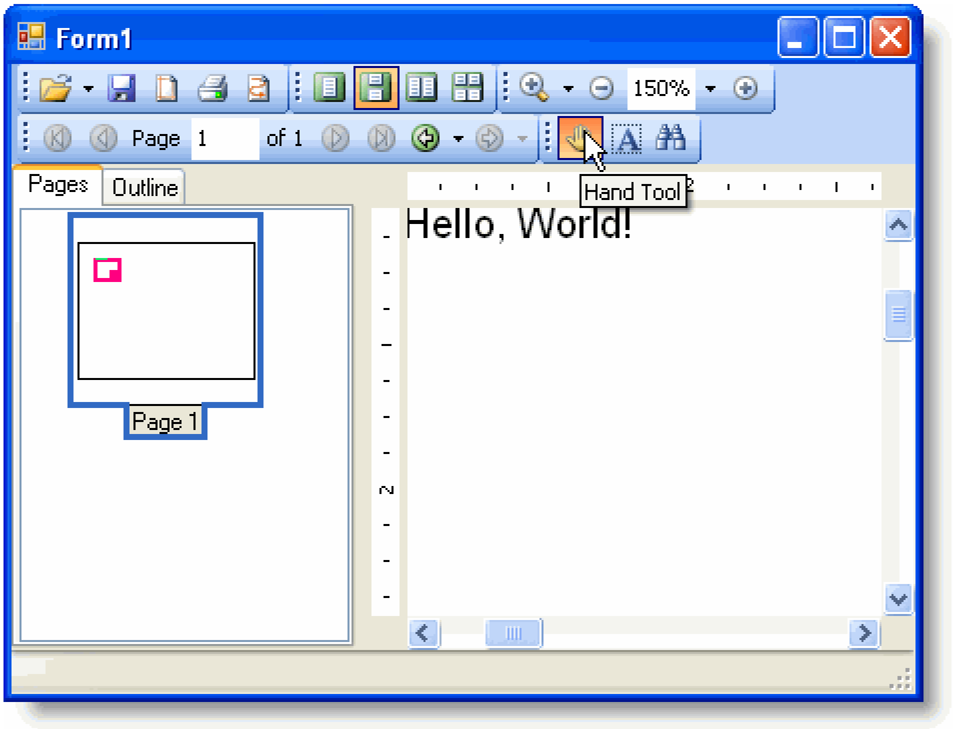


Hello, World!

4. Click the **Text Select** button to highlight the document text.



5. Click the **Hand Tool** button and perform a drag-and-drop operation on the document body so that the text appears in a different location.



6. You can continue to experiment with the preview by clicking Save or Print to open other dialog boxes

Congratulations, you have just created a "Hello, World!" document and completed the printing and previewing quick start guide! Save your project, in the following getting started tutorials you will continue to add to the project.

### *Creating Tables*

This topic describes how to create a simple table and a table with three columns and rows, add text to cells in the table, add images to specific cells in the table, add text below the table in your document, create borders around rows and columns in the table, and create a background color for specific cells in the table.

> **Note:** The following topics use the sample application created in the C1PrintDocument Quick Start topic as a base.

## Making a Simple Table

Tables provide one of the most useful features in documents. They can used both for tabular presentation of data, and for layout of other elements of a document. C1PrintDocument provides full-featured tables. In this section you will learn how to start using tables. We will use the "Hello, World!" sample application created in the C1PrintDocument Quick Start topic as our base, and add a table to it.

> **Note:** The sample code fragments in this topic assume that the "using C1.C1Preview" directive (in C# syntax; or an equivalent for other languages) has been inserted into the file, so that instead of fully qualified type names (such as C1.C1Preview.RenderText) we can use just the class name part (RenderText).

1. Open the **HelloWorld** application created in the C1PrintDocument Quick Start topic (alternatively, you can create a new application as described in the previous section).
2. Switch to code view and in the **Form_Load** event handler (create it if necessary) add the following code before the call to Generate method:

   - Visual Basic

     ```
     Dim rt As New RenderTable()
     Me.C1PrintDocument1.Body.Children.Add(rt)

     Dim row As Integer = 0
     Do While (row < 10)
         Dim col As Integer = 0
         Do While (col < 6)
             rt.Cells.Item(row, col).Text = String.Format("Cell ({0},{1})",
     row, col)
             col += 1
         Loop
         row += 1
     Loop
     ```

   - C#

     ```
     RenderTable rt = new RenderTable();
     this.c1PrintDocument1.Body.Children.Add(rt);

     for (int row = 0; row < 10; ++ row)
     {
         for (int col = 0; col < 6; ++ col)
         {
             rt.Cells[row, col].Text = string.Format("Cell ({0},{1})", row,
     col);
         }
     }
     ```

3. Do not forget to call the **Generate** method on the document.

- Visual Basic

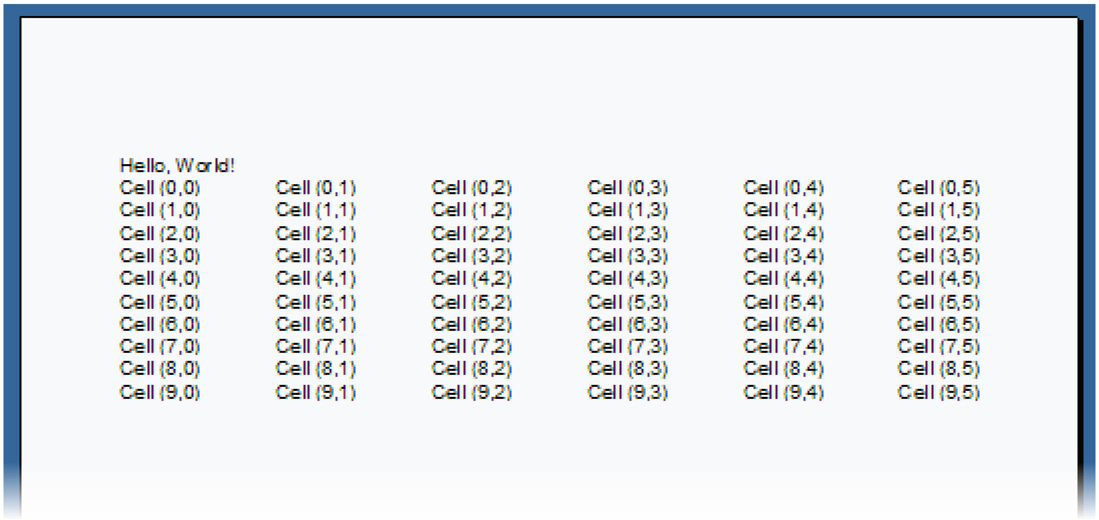
```
Me.C1PrintDocument1.Generate()
```

- C#

```
this.c1PrintDocument1.Generate();
```

**Run the program and observe:**

The preview will show the document similar to the one in the following picture:



Hello, World!

| | | | | | |
|---|---|---|---|---|---|
| Cell (0,0) | Cell (0,1) | Cell (0,2) | Cell (0,3) | Cell (0,4) | Cell (0,5) |
| Cell (1,0) | Cell (1,1) | Cell (1,2) | Cell (1,3) | Cell (1,4) | Cell (1,5) |
| Cell (2,0) | Cell (2,1) | Cell (2,2) | Cell (2,3) | Cell (2,4) | Cell (2,5) |
| Cell (3,0) | Cell (3,1) | Cell (3,2) | Cell (3,3) | Cell (3,4) | Cell (3,5) |
| Cell (4,0) | Cell (4,1) | Cell (4,2) | Cell (4,3) | Cell (4,4) | Cell (4,5) |
| Cell (5,0) | Cell (5,1) | Cell (5,2) | Cell (5,3) | Cell (5,4) | Cell (5,5) |
| Cell (6,0) | Cell (6,1) | Cell (6,2) | Cell (6,3) | Cell (6,4) | Cell (6,5) |
| Cell (7,0) | Cell (7,1) | Cell (7,2) | Cell (7,3) | Cell (7,4) | Cell (7,5) |
| Cell (8,0) | Cell (8,1) | Cell (8,2) | Cell (8,3) | Cell (8,4) | Cell (8,5) |
| Cell (9,0) | Cell (9,1) | Cell (9,2) | Cell (9,3) | Cell (9,4) | Cell (9,5) |

This simple example shows several important aspects of using tables in C1PrintDocument:

- Tables are represented by the RenderTable class, which inherits from RenderObject.
- Tables follow the model used in Microsoft Excel: their size is unlimited initially; the real size of the table when it is rendered is determined by the cell with the largest row and column numbers whose content was set. In our example, the table is 10 rows high and 6 columns wide, because the cell with the largest row and column indices that was set was the cell at position (9,5) (indices are zero-based). If you modify the code to set for example the text of the cell at position (10,7), the table will grow to 11 rows and 8 columns:
  - Visual Basic

    ```
    rt.Cells(10, 7).Text = "text at row 10, column 7"
    ```

  - C#

    ```
    rt.Cells[10, 7].Text = "text at row 10, column 7";
    ```

- By default, tables do not have visible grid lines (the "grid lines" term is used in **Reports for WinForms** for lines used to draw tables, as opposed to borders which can be drawn around any render object). To add grid lines (drawn with a black 0.5pt pen), add the following line of code to the **Form Load** event handler:
  - Visual Basic

    ```
    rt.Style.GridLines.All = LineDef.Default
    ```

  - C#

    ```
    rt.Style.GridLines.All = LineDef.Default;
    ```

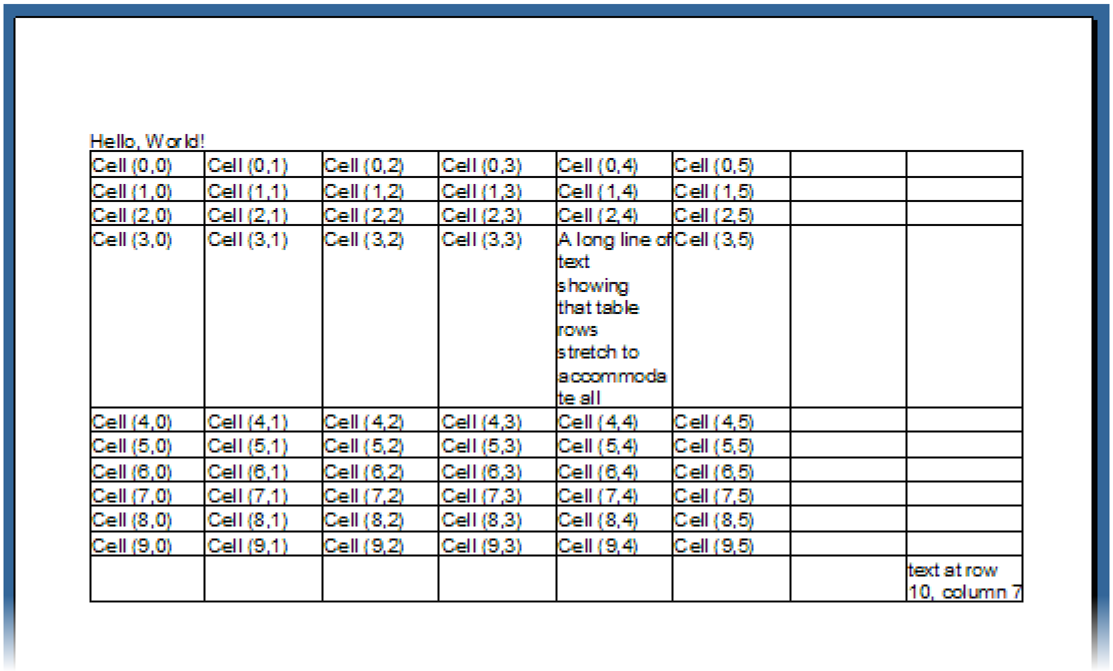
  By default, a table is as wide as its parent's client area (in this case, the whole page), with equally-sized columns. Rows' heights are automatic, though. So, if you add a line setting the text of an arbitrary cell in a table to a long text, you will see that the row containing that cell will grow vertically to accommodate all text. For example, adding this code to our example will produce a table looking like this (this table includes both changes described above):

- Visual Basic

```
rt.Cells(3, 4).Text = "A long line of text showing that table rows
stretch " + "to accommodate all content."
```

- C#

```
rt.Cells[3, 4].Text = "A long line of text showing that table rows
stretch " + "to accommodate all content.";
```



Hello, World!

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Cell (0,0) | Cell (0,1) | Cell (0,2) | Cell (0,3) | Cell (0,4) | Cell (0,5) | | |
| Cell (1,0) | Cell (1,1) | Cell (1,2) | Cell (1,3) | Cell (1,4) | Cell (1,5) | | |
| Cell (2,0) | Cell (2,1) | Cell (2,2) | Cell (2,3) | Cell (2,4) | Cell (2,5) | | |
| Cell (3,0) | Cell (3,1) | Cell (3,2) | Cell (3,3) | A long line of text showing that table rows stretch to accommodate all | Cell (3,5) | | |
| Cell (4,0) | Cell (4,1) | Cell (4,2) | Cell (4,3) | Cell (4,4) | Cell (4,5) | | |
| Cell (5,0) | Cell (5,1) | Cell (5,2) | Cell (5,3) | Cell (5,4) | Cell (5,5) | | |
| Cell (6,0) | Cell (6,1) | Cell (6,2) | Cell (6,3) | Cell (6,4) | Cell (6,5) | | |
| Cell (7,0) | Cell (7,1) | Cell (7,2) | Cell (7,3) | Cell (7,4) | Cell (7,5) | | |
| Cell (8,0) | Cell (8,1) | Cell (8,2) | Cell (8,3) | Cell (8,4) | Cell (8,5) | | |
| Cell (9,0) | Cell (9,1) | Cell (9,2) | Cell (9,3) | Cell (9,4) | Cell (9,5) | | |
| | | | | | | | text at row 10, column 7 |

For your reference, here is the complete text of the form load event handler that produced the above document:

- Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Me.C1PrintDocument1.Body.Children.Add(New RenderText("Hello, World!"))
    Dim rt As New RenderTable()
    Me.C1PrintDocument1.Body.Children.Add(rt)

Dim row As Integer = 0
Do While (row < 10)
    Dim col As Integer = 0
    Do While (col < 6)
        rt.Cells.Item(row, col).Text = String.Format("Cell ({0},{1})",
row, col)
        col += 1
    Loop
    row += 1
Loop
    rt.Cells(3, 4).Text = "A long line of text showing that table rows " +
"stretch to accommodate all content."
    rt.Cells(10, 7).Text = "text at row 10, column 7"
    rt.Style.GridLines.All = LineDef.Default
    Me.C1PrintDocument1.Generate()
```

```
End Sub
```

- C#

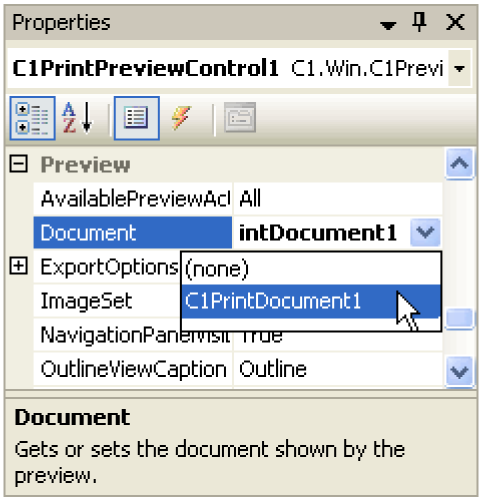
```
private void Form1_Load(object sender, EventArgs e)
{
    this.c1PrintDocument1.Body.Children.Add(new RenderText("Hello,
World!"));
    RenderTable rt = new RenderTable();
    this.c1PrintDocument1.Body.Children.Add(rt);
    for (int row = 0; row < 10; ++row)
    {
        for (int col = 0; col < 6; ++col)
        {
            rt.Cells[row, col].Text = string.Format("Cell ({0},{1})", row,
col);
        }
    }
    rt.Cells[3, 4].Text = "A long line of text showing that table rows " +
"stretch to accommodate all content.";
    rt.Cells[10, 7].Text = "text at row 10, column 7";
    rt.Style.GridLines.All = LineDef.Default;
    this.c1PrintDocument1.Generate();
}
```

## Creating a Table with Three Columns and Rows

This topic illustrates the basics of setting up a table with three rows and three columns. Complete the following steps:

1. First, set up the basic framework for the sample that will allow generating and previewing the document. Create a new .NET Windows Application project. Add a C1PrintPreviewControl and a C1PrintDocument component on the form.

2. Set the C1PrintPreviewControl's **Dock** property to **Fill** (the preview will be the only control on our form). Set the C1PrintPreviewControl's **Document** property to the **C1PrintDocument** as shown in the following picture:

   AvailablePreviewAc All
   Document intDocument1
   C1PrintDocument1
   ExportOptions
   ImageSet
   NavigationPanelVisi True
   OutlineViewCaption Outline

   **Document**
   Gets or sets the document shown by the preview.

   This will make the C1PrintPreviewControl show the C1PrintDocument.

3. Double-click the form's title bar to switch to Code view and create a new handler for the **Form_Load** event in the source code.

4. Second, create a new C1.C1PrintDocument.RenderTable object and assign it to a variable by adding the following code to the **Form1_Load** event:

   - Visual Basic

   ```
   Dim table As C1.C1Preview.RenderTable = New
   C1.C1Preview.RenderTable(Me.C1PrintDocument1)
   ```

   - C#

   ```
   C1.C1Preview.RenderTable table = new
   C1.C1Preview.RenderTable(this.c1PrintDocument1);
   ```

5. Now, add 3 columns to the table and 3 rows to the table's body by adding the following code after the code added in the previous step:

   - Visual Basic

   ```
   ' Add 3 rows.
   Dim r As Integer = 3

   ' Add 3 columns.
   Dim c As Integer = 3

   Dim row As Integer
   Dim col As Integer

   For row = 0 To r - 1 Step +1
       For col = 0 To c - 1 Step +1
           Dim celltext As C1.C1Preview.RenderText = New
   C1.C1Preview.RenderText(Me.C1PrintDocument1)

           ' Add empty cells.
           celltext.Text = String.Format("", row, col)
           table.Cells(row, col).RenderObject = celltext
       Next
   Next
   ```

   - C#

   ```
   // Add 3 rows.
   const int r = 3;

   // Add 3 columns.
   const int c = 3;

   for (int row = 0; row < r; ++row)
   {
       for (int col = 0; col < c; ++col)
       {
           C1.C1Preview.RenderText celltext = new
   C1.C1Preview.RenderText(this.c1PrintDocument1);
           celltext.Text = string.Format("", row, col);

           // Add empty cells.
           table.Cells[row, col].RenderObject = celltext;
       }
   }
   ```

   Note also that while we added columns "directly" to the table, rows were added to the table's body. That is because a RenderTable always consists of 3 "bands": **Header**, **Body** and **Footer**. Any of the three bands may be empty in the table. If you just want a simple table you can add rows to the body as we do in this example.

6. Make the table's width and height fifteen centimeters long, by adding the following code:

    - Visual Basic

    ```
    table.Height = New C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm)
    table.Width = New C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm)
    ```

    - C#

    ```
    table.Height = new C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm);
    table.Width = new C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm);
    ```

7. By default, tables have no borders. Add a dark gray gridlines to the table:

    - Visual Basic

    ```
    table.Style.GridLines.All = New C1.C1Preview.LineDef(Color.DarkGray)
    ```

    - C#

    ```
    table.Style.GridLines.All = new C1.C1Preview.LineDef(Color.DarkGray);
    ```

8. When you have created the render object(s), you need to add them to your document. The way to do it is to first call the **Add** method on the document to add the table to the body of the document, and then use the Generate method to create the document. Here is the code:

    - Visual Basic

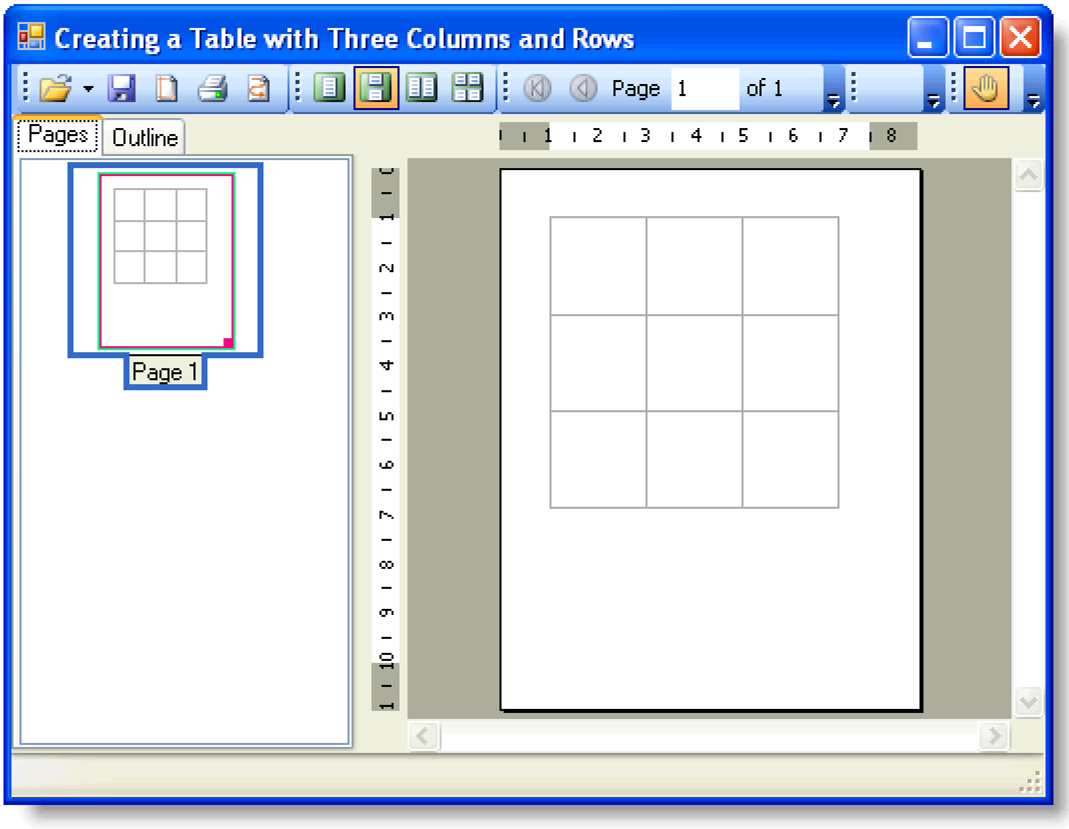
    ```
    Me.C1PrintDocument1.Body.Children.Add(table)
    Me.C1PrintDocument1.Generate()
    ```

    - C#

    ```
    this.c1PrintDocument1.Body.Children.Add(table);
    this.c1PrintDocument1.Generate();
    ```

**Run the program and observe the following:**

Your application will appear similar to the image below at run time:



### Adding Text to Table Cells

This topic shows how to use the RenderText class to add text into specific cells of the table.

1. The following code to set up a table with dark gray gridlines should already exist in your source file:

   - Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    ' Make a table.
    Dim table As C1.C1Preview.RenderTable = New
C1.C1Preview.RenderTable(Me.C1PrintDocument1)
    table.Style.GridLines.All = New
C1.C1Preview.LineDef(Color.DarkGray)

    ' Generate the document.
    Me.C1PrintDocument1.Body.Children.Add(table)
    Me.C1PrintDocument1.Generate()
End Sub
```

   - C#

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Make a table.
    C1.C1Preview.RenderTable table = new
C1.C1Preview.RenderTable(this.c1PrintDocument1);
    table.Style.GridLines.All = new
C1.C1Preview.LineDef(Color.DarkGray);
```

```
    // Generate the document.
    this.c1PrintDocument1.Body.Children.Add(table);
    this.c1PrintDocument1.Generate();
}
```

2. Showing any kind of content in a table cell is done by assigning the render object representing that content to the cell's RenderObject property. But, because showing text in table cells is such a common task, cells have an additional specialized property RenderText that we will use. In order to set the texts of all cells in the table, you need to loop over the table's rows, and inside that loop do another loop over the row's columns. In the body of the nested loop set the Text property to the desired text as follows (for the sake of this sample, we leave cells (1,1) and (1,2) empty):

   - Visual Basic

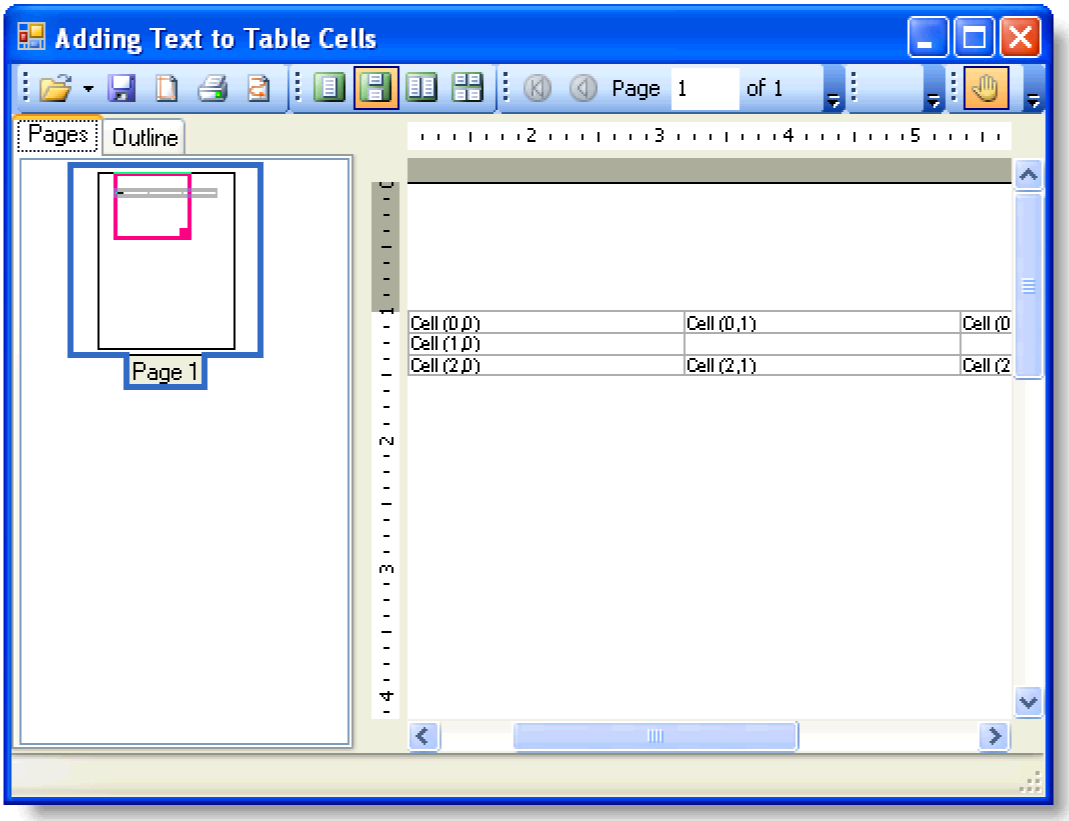
```
' Add 3 rows.
Dim r As Integer = 3

' Add 3 columns.
Dim c As Integer = 3

Dim row As Integer
Dim col As Integer

For row = 0 To r - 1 Step +1
    For col = 0 To c - 1 Step +1
        If (Not (row = 1 And col = 1)) And (Not (row = 1 And col = 2))
Then
            Dim celltext As C1.C1Preview.RenderText = New
C1.C1Preview.RenderText(Me.C1PrintDocument1)
            celltext.Text = String.Format("Cell ({0},{1})", row, col)

            ' Add cells with text.
            table.Cells(row, col).RenderObject = celltext
        End If
    Next
Next
```

   - C#

```
// Add 3 rows.
const int r = 3;

// Add 3 columns.
const int c = 3;

for (int row = 0; row < r; ++row)
{
    for (int col = 0; col < c; ++col)
    {
        if (!(row == 1 && col == 1) && !(row == 1 && col == 2))
        {
            C1.C1Preview.RenderText celltext = new
C1.C1Preview.RenderText(this.c1PrintDocument1);
            celltext.Text = string.Format("Cell ({0}, {1})", row, col);

            // Add cells with text.
            table.Cells[row, col].RenderObject = celltext;
        }
    }
}
```

**Run the program and observe the following:**

Your table should look similar to the table below:



|  |  |  |
|---|---|---|
| Cell (0,0) | Cell (0,1) | Cell (0 |
| Cell (1,0) |  |  |
| Cell (2,0) | Cell (2,1) | Cell (2 |

## Adding Two Images to Specific Cells of the Table

This topic demonstrates how to add two different images to specific cells in an existing table by using the RenderImage class. It also shows how to align images in cells using the ImageAlignHorzEnum. Note that the sample below uses the empty 3 by 3 table which was built in Creating a Table with Three Columns and Rows and that you'll need to have two GIF or JPEG images on hand to complete the steps in this topic. Complete the following steps:

1. The following code should already exist in your source file:

    - Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    ' Make a table.
    Dim table As C1.C1Preview.RenderTable = New
C1.C1Preview.RenderTable(Me.C1PrintDocument1)
    table.Style.GridLines.All = New
C1.C1Preview.LineDef(Color.DarkGray)

    Dim r As Integer = 3
    Dim c As Integer = 3
    Dim row As Integer
    Dim col As Integer
    For row = 0 To r - 1 Step +1
```

```
        For col = 0 To c - 1 Step +1
            Dim celltext As C1.C1Preview.RenderText = New
C1.C1Preview.RenderText(Me.C1PrintDocument1)

            ' Add empty cells.
            celltext.Text = String.Format("", row, col)
            table.Cells(row, col).RenderObject = celltext
        Next
    Next

    ' Generate the document.
    Me.C1PrintDocument1.Body.Children.Add(table)
    Me.C1PrintDocument1.Generate()
End Sub
```

- C#

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Make a table.
    C1.C1Preview.RenderTable table = new
C1.C1Preview.RenderTable(this.c1PrintDocument1);
    table.Style.GridLines.All = new
C1.C1Preview.LineDef(Color.DarkGray);

    const int r = 3;
    const int c = 3;
    for (int row = 0; row < r; ++row)
    {
        for (int col = 0; col < c; ++col)
        {
            C1.C1Preview.RenderText celltext = new
C1.C1Preview.RenderText(this.c1PrintDocument1);
            celltext.Text = string.Format("", row, col);

            // Add empty cells.
            table.Cells[row, col].RenderObject = celltext;
        }
    }

    // Generate the document.
    this.c1PrintDocument1.Body.Children.Add(table);
    this.c1PrintDocument1.Generate();
}
```

2. Add the following code after the line adding the rows (the new code will fix the center cell's size in the table):

   - Visual Basic

```
' Fix the center cell's size.
table.Rows(1).Height = New C1.C1Preview.Unit(5,
C1.C1Preview.UnitTypeEnum.Cm)
table.Cols(1).Width = New C1.C1Preview.Unit(8,
C1.C1Preview.UnitTypeEnum.Cm)
```

   - C#

```
// Fix the center cell's size.
table.Rows[1].Height = new C1.C1Preview.Unit(5,
C1.C1Preview.UnitTypeEnum.Cm);
```

```
table.Cols[1].Width = new C1.C1Preview.Unit(8,
C1.C1Preview.UnitTypeEnum.Cm);
```

3. Create two JPEG or GIF images or use images that already exist.

4. Add two **PictureBox** controls onto your form. Set their **Image** properties to the two images created in the previous step. Also, make the two picture boxes invisible (set **Visible** to **False**) so that they won't clutter the form (those controls are used only as storage for the images. The images will be rendered into the C1PrintDocument).

5. Use the **TableCell.CellStyle** property to modify the base styles for the cells' content. In this sample we will modify the ImageAlign property for the cells. Enter the following code to set up the image alignment:

   - Visual Basic

   ```
   ' Set up image alignment.
   table.CellStyle.ImageAlign.StretchHorz = False
   table.CellStyle.ImageAlign.StretchVert = False
   table.CellStyle.ImageAlign.AlignHorz =
   C1.C1Preview.ImageAlignHorzEnum.Center
   ```

   - C#

   ```
   // Set up image alignment.
   table.CellStyle.ImageAlign.StretchHorz = false;
   table.CellStyle.ImageAlign.StretchVert = false;
   table.CellStyle.ImageAlign.AlignHorz =
   C1.C1Preview.ImageAlignHorzEnum.Center;
   ```

6. In C1PrintDocument, images are rendered using the RenderImage class (which subclasses the **RenderObject**). Create two new **RenderImage** objects for the two images as follows:

   - Visual Basic

   ```
   Dim img1 As C1.C1Preview.RenderImage = New
   C1.C1Preview.RenderImage(Me.C1PrintDocument1)
   Dim img2 As C1.C1Preview.RenderImage = New
   C1.C1Preview.RenderImage(Me.C1PrintDocument1)
   ```

   - C#

   ```
   C1.C1Preview.RenderImage img1 = new
   C1.C1Preview.RenderImage(this.c1PrintDocument1);
   C1.C1Preview.RenderImage img2 = new
   C1.C1Preview.RenderImage(this.c1PrintDocument1);
   ```

7. Now, set the RenderImage's **Image** properties to the images stored in the picture boxes:

   - Visual Basic

   ```
   img1.Image = Me.PictureBox1.Image
   img2.Image = Me.PictureBox2.Image
   ```

   - C#

   ```
   img1.Image = this.pictureBox1.Image;
   img2.Image = this.pictureBox2.Image;
   ```

8. Assign the RenderImage objects to the RenderObject properties of the cells so that the images will render in those cells:

   - Visual Basic

   ```
   table.Cells(1, 1).RenderObject = img1
   table.Cells(1, 2).RenderObject = img2
   ```

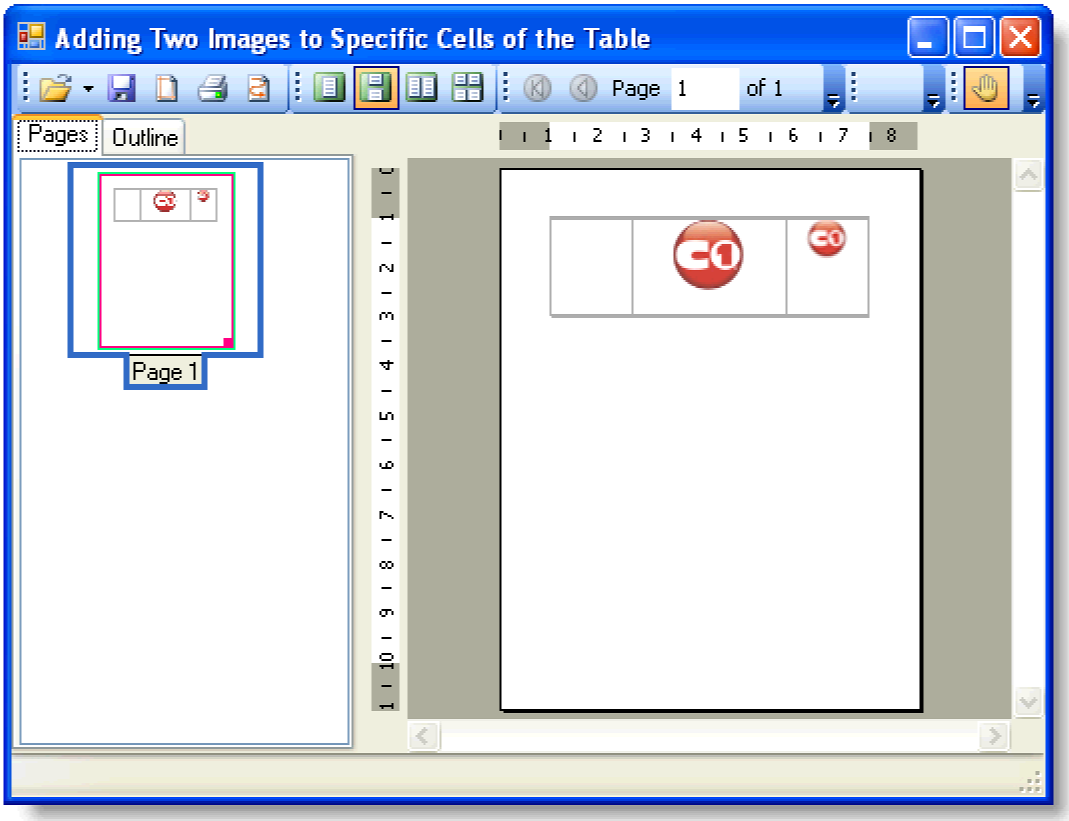
   - C#

   ```
   table.Cells[1, 1].RenderObject = img1;
   table.Cells[1, 2].RenderObject = img2;
   ```

**Note:** The top left cell of the table is at row 0, column 0.

**Run the program and observe the following:**

Your table should look similar to the table below:



## Creating Borders Around Rows and Columns in Your Table

This topic demonstrates how to create distinct borders around a row and a column by using the LineDef class. This topic assumes you already have a table with three columns and three rows.

1. The following code should already exist in your source file:

   - Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    ' Make a table.
    Dim table As C1.C1Preview.RenderTable = New
C1.C1Preview.RenderTable(Me.C1PrintDocument1)
    table.Style.GridLines.All = New
C1.C1Preview.LineDef(Color.DarkGray)

    Dim r As Integer = 3
    Dim c As Integer = 3
    Dim row As Integer
    Dim col As Integer
    For row = 0 To r - 1 Step +1
```

```
        For col = 0 To c - 1 Step +1
            Dim celltext As C1.C1Preview.RenderText = New
C1.C1Preview.RenderText(Me.C1PrintDocument1)

            ' Add empty cells.
            celltext.Text = String.Format("", row, col)
            table.Cells(row, col).RenderObject = celltext
        Next
    Next

    ' Generate the document.
    Me.C1PrintDocument1.Body.Children.Add(table)
    Me.C1PrintDocument1.Generate()
End Sub
```

- C#

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Make a table.
    C1.C1Preview.RenderTable table = new
C1.C1Preview.RenderTable(this.c1PrintDocument1);
    table.Style.GridLines.All = new
C1.C1Preview.LineDef(Color.DarkGray);

    const int r = 3;
    const int c = 3;
    for (int row = 0; row < r; ++row)
    {
        for (int col = 0; col < c; ++col)
        {
        C1.C1Preview.RenderText celltext = new
C1.C1Preview.RenderText(this.c1PrintDocument1);
            celltext.Text = string.Format("", row, col);

            // Add empty cells.
            table.Cells[row, col].RenderObject = celltext;
        }
    }

    // Generate the document.
    this.c1PrintDocument1.Body.Children.Add(table);
    this.c1PrintDocument1.Generate();
}
```

2. Add the following code to your project to make the table's width and height fifteen centimeters long:

   - Visual Basic

```
table.Height = New C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm)
table.Width = New C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm)
```

   - C#

```
table.Height = new C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm);
table.Width = new C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm);
```

3. Add the following code to your project to assign a new instance of the LineDef class to the borders of the third row as follows (note that the constructor we use specifies that the new border will be red and 2 points wide):

   - Visual Basic

```
table.Rows(2).Style.Borders.All = New C1.C1Preview.LineDef("2pt",
Color.Red)
```

- C#

```
table.Rows[2].Style.Borders.All = new C1.C1Preview.LineDef("2pt",
Color.Red);
```

4. Assign a new instance of the LineDef class to the borders of the first column as follows (note that the constructor we use specifies that the new border will be blue and 6 points wide):

- Visual Basic

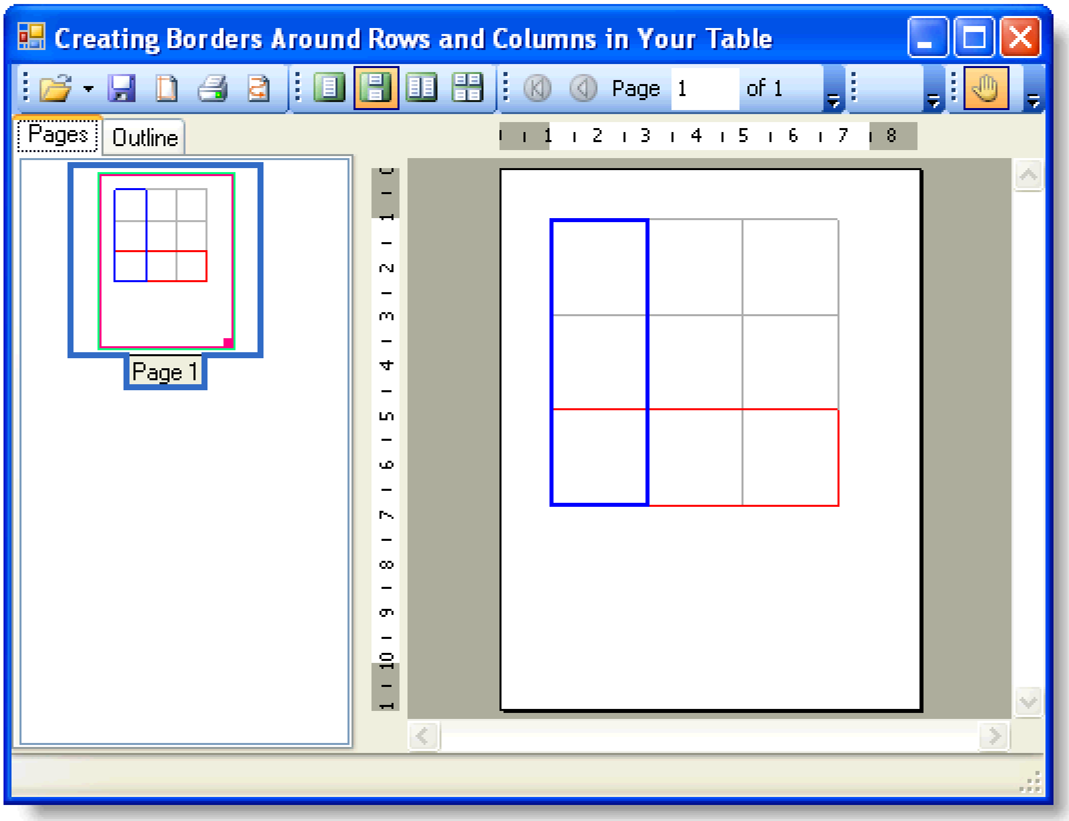
```
table.Cols(0).Style.Borders.All = New C1.C1Preview.LineDef("6pt",
Color.Blue)
```

- C#

```
table.Cols[0].Style.Borders.All = new C1.C1Preview.LineDef("6pt",
Color.Blue);
```

**Run the program and observe the following:**

Your borders will appear similar to the table below at run time:



## Creating a Background Color for Specific Cells in the Table

This topic demonstrates how to create background colors for specific cells in the table. It also demonstrates how to use the **TableCell.CellStyle** property to set the styles used in the table that will be rendered. This topic assumes you have a table with three columns and three rows.

1. The following code below should already exist in your source file:

- Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    ' Make a table.
    Dim table As C1.C1Preview.RenderTable = New
C1.C1Preview.RenderTable(Me.C1PrintDocument1)
    table.Style.GridLines.All = New
C1.C1Preview.LineDef(Color.DarkGray)

    Dim r As Integer = 3
    Dim c As Integer = 3
    Dim row As Integer
    Dim col As Integer
    For row = 0 To r - 1 Step +1
        For col = 0 To c - 1 Step +1
            Dim celltext As C1.C1Preview.RenderText = New
C1.C1Preview.RenderText(Me.C1PrintDocument1)

            ' Add empty cells.
            celltext.Text = String.Format("", row, col)
            table.Cells(row, col).RenderObject = celltext
        Next
    Next

    ' Generate the document.
    Me.C1PrintDocument1.Body.Children.Add(table)
    Me.C1PrintDocument1.Generate()
End Sub
```

- C#

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Make a table.
    C1.C1Preview.RenderTable table = new
C1.C1Preview.RenderTable(this.c1PrintDocument1);
    table.Style.GridLines.All = new
C1.C1Preview.LineDef(Color.DarkGray);

    const int r = 3;
    const int c = 3;
    for (int row = 0; row < r; ++row)
    {
        for (int col = 0; col < c; ++col)
        {
            C1.C1Preview.RenderText celltext = new
C1.C1Preview.RenderText(this.c1PrintDocument1);
            celltext.Text = string.Format("", row, col);

            // Add empty cells.
            table.Cells[row, col].RenderObject = celltext;
        }
    }

    // Generate the document.
    this.c1PrintDocument1.Body.Children.Add(table);
    this.c1PrintDocument1.Generate();
}
```

2. Make the table's width and height fifteen centimeters long:

   - Visual Basic

     ```
     table.Height = New C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm)
     table.Width = New C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm)
     ```

   - C#

     ```
     table.Height = new C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm);
     table.Width = new C1.C1Preview.Unit(15, C1.C1Preview.UnitTypeEnum.Cm);
     ```

3. Add the following code below the last line of code listed above. This code will create a crimson background color for row 1, column 2.

   - Visual Basic

     ```
     table.Cells(1, 2).CellStyle.BackColor = Color.Crimson
     ```

   - C#

     ```
     table.Cells[1, 2].CellStyle.BackColor = Color.Crimson;
     ```

   **Note:** The rows and columns begin with 0. The code above uses the **TableCell.CellStyle** property to set the cell's style.

4. Create a blue-violet background color for row 0, column 1. Enter the following code:

   - Visual Basic

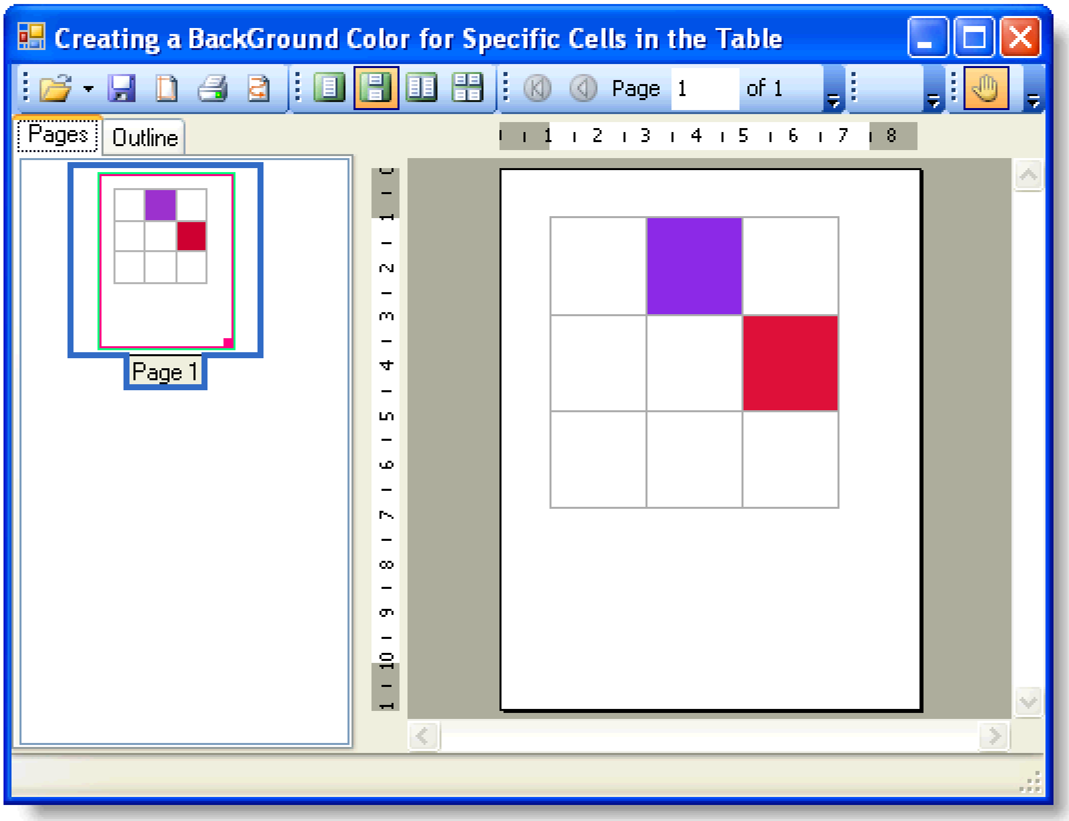
     ```
     table.Cells(0, 1).CellStyle.BackColor = Color.BlueViolet
     ```

   - C#

     ```
     table.Cells[0, 1].CellStyle.BackColor = Color.BlueViolet;
     ```

**Run the program and observe the following:**

Your table should appear similar to the table below:



### *Adding Text*

The following topics describe how to add paragraphs, add text below a table, and modify the font and style of text.

## Adding Paragraphs to the Document

All content of a C1PrintDocument is represented by render objects. The **Reports for WinForms** assembly provides a hierarchy of classes derived from RenderObject, designed to represent content of various types, such as text, images and so on. For instance, above we used the RenderText class to add a line of text to the document. In this section we will show how to create paragraphs of text (which may combine fragments of text drawn with different styles, inline images, and hyperlinks) using the RenderParagraph class.

> **Note:** The sample code fragments in this topic assume that "using C1.C1Preview;" directive (in C# syntax; or an equivalent for other languages) has been inserted into the file, so that instead of fully qualified type names (such as C1.C1Preview.RenderText) we can use just the class name part (RenderText).

A RenderParagraph can be created with the following line of code:

- Visual Basic

  ```
  Dim rp As New RenderParagraph()
  ```

- C#

  ```
  RenderParagraph rp = new RenderParagraph();
  ```

Paragraphs should be used rather than RenderText in any of the following cases:

- You need to show text in different styles within the same paragraph.
- Inline images (usually small icon-like images) must be inserted in the text flow.

- A hyperlink must be associated with a portion of the text (for example, a word) rather than with the whole text (please see the Anchors and Hyperlinks section).

The content of a paragraph is comprised of ParagraphObject objects. ParagraphObject is the abstract base class; the two inherited classes are ParagraphText and ParagraphImage, representing fragments of text and inline images, correspondingly. You can fill a paragraph with content by creating objects of those two types and adding them to the **RenderParagraph.Content** collection. Various constructor overloads and properties are provided for convenient creation/setup of those objects. In-paragraph hyperlinks can be created by specifying the Hyperlink property of the paragraph object which should be a hyperlink. An alternative approach is to use various overloads of the shortcut methods AddText, AddImage and AddHyperlink, as shown in the following example:

- Visual Basic

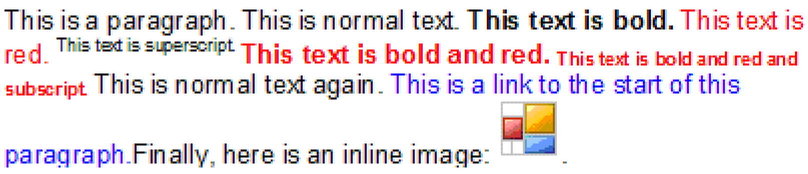
```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    ' Create a paragraph.
    Dim rpar As New RenderParagraph()
    Dim f As New Font(rpar.Style.Font, FontStyle.Bold)

    rpar.Content.AddText("This is a paragraph. This is normal text. ")
    rpar.Content.AddText("This text is bold. ", f)
    rpar.Content.AddText("This text is red. ", Color.Red)
    rpar.Content.AddText("This text is superscript. ",
TextPositionEnum.Superscript)
    rpar.Content.AddText("This text is bold and red. ", f, Color.Red)
    rpar.Content.AddText("This text is bold and red and subscript. ", f,
Color.Red, TextPositionEnum.Subscript)
    rpar.Content.AddText("This is normal text again. ")
    rpar.Content.AddHyperlink("This is a link to the start of this
paragraph.", rpar.Content(0))
    rpar.Content.AddText("Finally, here is an inline image: ")
    rpar.Content.AddImage(Me.Icon.ToBitmap())
    rpar.Content.AddText(".")

    ' Add the paragraph to the document.
    Me.C1PrintDocument1.Body.Children.Add(rpar)
    Me.C1PrintDocument1.Generate()
End Sub
```

- C#

```
private void Form1_Load(object sender, EventArgs e)
{
    // Create a paragraph.
    RenderParagraph rpar = new RenderParagraph();
    Font f = new Font(rpar.Style.Font, FontStyle.Bold);

    rpar.Content.AddText("This is a paragraph. This is normal text. ");
    rpar.Content.AddText("This text is bold. ", f);
    rpar.Content.AddText("This text is red. ", Color.Red);
    rpar.Content.AddText("This text is superscript. ",
TextPositionEnum.Superscript);
    rpar.Content.AddText("This text is bold and red. ", f, Color.Red);
    rpar.Content.AddText("This text is bold and red and subscript. ", f,
Color.Red, TextPositionEnum.Subscript);
    rpar.Content.AddText("This is normal text again. ");
    rpar.Content.AddHyperlink("This is a link to the start of this
paragraph.", rpar.Content[0]);
    rpar.Content.AddText("Finally, here is an inline image: ");
    rpar.Content.AddImage(this.Icon.ToBitmap());
    rpar.Content.AddText(".");

    // Add the paragraph to the document.
    this.c1PrintDocument1.Body.Children.Add(rpar);
    this.c1PrintDocument1.Generate();
}
```

**Run the program and observe the following:**

The following image shows the document generated by this code:



This is a paragraph. This is normal text. **This text is bold.** This text is red. This text is superscript **This text is bold and red.** This text is bold and red and subscript This is normal text again. This is a link to the start of this paragraph. Finally, here is an inline image: .

## Adding Text Below the Table in the Document

This topic shows how to use the C1.C1PrintDocument.RenderTable object to render the text into the block flow. It also demonstrates how to use the Padding property to advance the block position so that the next render object (in this case, text) is rendered there. In this topic we will use the Padding property to put the text 1 cm below the table in your document. This topic assumes you already have a table created.

1. Use the C1.C1PrintDocument.RenderTable object to create the text to be displayed:

   - Visual Basic

     ```
     Dim caption As C1.C1Preview.RenderText = New
     C1.C1Preview.RenderText(Me.C1PrintDocument1)
     caption.Text = "In the table above, there are three rows and three
     columns."
     ```

   - C#

     ```
     C1.C1Preview.RenderText caption = new
     C1.C1Preview.RenderText(this.c1PrintDocument1);
     caption.Text = "In the table above, there are three rows and three
     columns.";
     ```

2. Use the Padding property to position the text 1 cm below the table:

   - Visual Basic

     ```
     caption.Style.Padding.Top = New C1.C1Preview.Unit(1,
     C1.C1Preview.UnitTypeEnum.Cm)
     ```

   - C#

     ```
     caption.Style.Padding.Top = new C1.C1Preview.Unit(1,
     C1.C1Preview.UnitTypeEnum.Cm);
     ```

3. Add the text below the table using the **Add** method. Insert the **Add** method for the text below the **Add** method for the table, as shown in the following code:

   - Visual Basic

     ```
     Me.C1PrintDocument1.Body.Children.Add(table)
     Me.C1PrintDocument1.Body.Children.Add(caption)
     ```

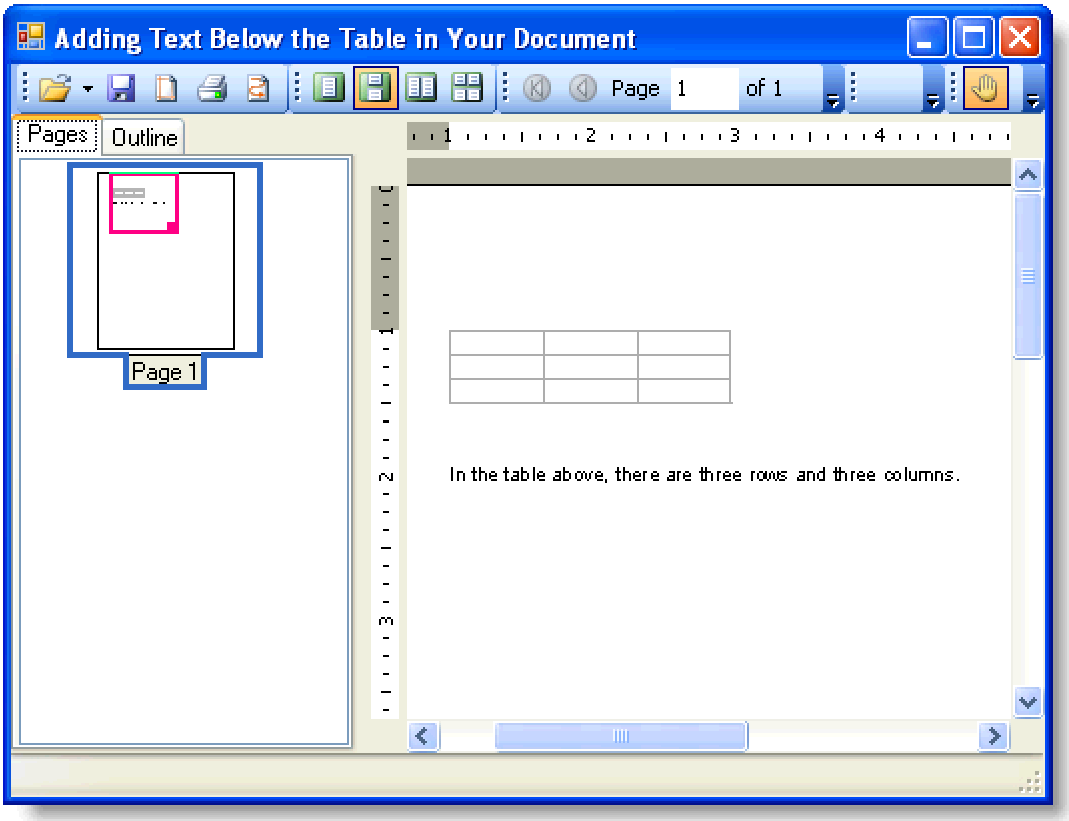
   - C#

```
this.c1PrintDocument1.Body.Children.Add(table);
this.c1PrintDocument1.Body.Children.Add(caption);
```

**Note:** Placing the **Add** method for the text below the **Add** method for the table inserts the text below the table. If it is placed above the **Add** method for the table, the text will appear above the table.

**Run the program and observe the following:**

Your document should appear similar to the document below:



In the table above, there are three rows and three columns.

## Modifying the Font and Style of the Text

C1PrintDocument contains a **RenderInLineText** method that renders the specified string without starting a new paragraph into the block flow. The **RenderInLineText** method automatically wraps the text. This topic illustrates how to use the **RenderInLineText** method.

1. Create a new Windows Forms application. Add a **C1PrintPreview** control onto your form. Add a C1PrintDocument component onto your form – it will appear in the components' tray below the form. The preview will have the default name **C1PrintPreview1**, the document **C1PrintDocument1**. Set the value of the **Document** property of C1PrintPreview1 control to **C1PrintDocument1**, so that the preview will show the document when the application runs.

2. Double click the form to create a handler for the **Form_Load** event – this is where all code shown below will be written.

3. Begin the document with the StartDoc method and create a line of text using the default font. For example:

- Visual Basic

  ```
  Me.C1PrintDocument1.StartDoc()
  Me.C1PrintDocument1.RenderInlineText("With C1PrintDocument you can
  print ")
  ```

- C#

  ```
  this.c1PrintDocument1.StartDoc();
  this.c1PrintDocument1.RenderInlineText("With C1PrintDocument you can
  print ");
  ```

4. Continue with a different font and color and then go back to the default font and color:

   - Visual Basic

     ```
     Me.C1PrintDocument1.RenderInlineText("Line by Line", New Font("Times
     New Roman", 30, FontStyle.Bold), Color.FromArgb(0, 0, 125))
     Me.C1PrintDocument1.RenderInlineText(" and modify text attributes as
     you go.")
     ```

   - C#

     ```
     this.c1PrintDocument1.RenderInlineText("Line by Line", new Font("Times
     New Roman", 30, FontStyle.Bold), Color.FromArgb(0, 0, 125));
     this.c1PrintDocument1.RenderInlineText(" and modify text attributes as
     you go.");
     ```

5. Make the last few words of the line a green color.

   - Visual Basic

     ```
     Me.C1PrintDocument1.RenderInlineText(" The text wraps automatically, so
     your life becomes easier.", Color.Green)
     ```

   - C#

     ```
     this.c1PrintDocument1.RenderInlineText(" The text wraps automatically,
     so your life becomes easier.", Color.Green);
     ```

6. End the document with the EndDoc method.

   - Visual Basic

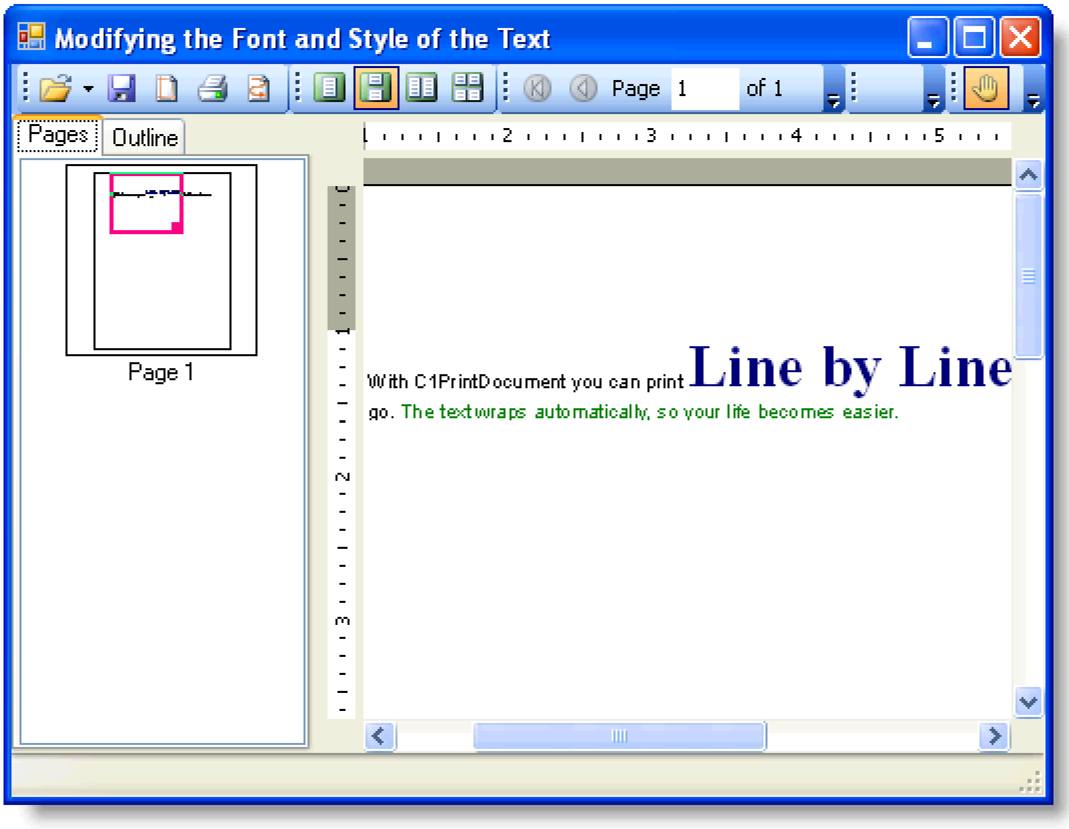
     ```
     Me.C1PrintDocument1.EndDoc()
     ```

   - C#

     ```
     this.c1PrintDocument1.EndDoc();
     ```

**Run the program and observe the following:**

The text should appear similar to the text below:



With C1PrintDocument you can print **Line by Line** go. The text wraps automatically, so your life becomes easier.

### *Creating Page Headers in C1PrintDocument*

The following topics describe how to create a page header with three parts, and add a background color to the page header.

## Creating a Page-Header with Three Parts

This topic demonstrates how to create a header that is divided into three columns. The following key points are shown in this topic:

- Creating a table with one row and three columns in **C1PrintDocument**.
- Setting up the text alignment in each section of the page header.
- The TextAlignHorz property of the Style class is used to specify the horizontal alignment of the text. You can assign a member (left, right, justify or center) of the AlignHorzEnum to the TextAlignHorz property.

The following detailed steps demonstrate how to create a header with three parts.

1. Create a new Windows Forms application.
2. Add a **C1PrintPreview** control onto your form.
3. Add a **C1PrintDocument** component onto your form – it will appear in the components' tray below the form. The preview will have the default name **C1PrintPreview1**, the document **C1PrintDocument1**.
4. Set the value of the **Document** property of the C1PrintPreview1 control to **C1PrintDocument1**, so that the preview will show the document when the application runs.
5. Double click the form to create a handler for the **Form_Load** event – this is where all code shown below will be written. In the **Form_Load** event, we will set up our document. Create a RenderTable for the page header:

- Visual Basic

  ```
  Me.C1PrintDocument1.StartDoc()
  Dim theader As New C1.C1Preview.RenderTable(Me.C1PrintDocument1)
  ```

- C#

  ```
  this.c1PrintDocument1.StartDoc();
  C1.C1Preview.RenderTable theader = new
  C1.C1Preview.RenderTable(this.c1PrintDocument1);
  ```

6. Add one row to its body, and 3 columns for the left, middle, and right parts of the header. We will use the TextAlignHorz property to set the alignment of the text in each column of the page header. We will also assign a new font style for the text in our page header. Note, in this example the font type will be Arial and it will be 14 points in size.

   - Visual Basic

     ```
     ' Set up alignment for the parts of the header.
     theader.Cells(0, 0).Style.TextAlignHorz =
     C1.C1Preview.AlignHorzEnum.Left
     theader.Cells(0, 1).Style.TextAlignHorz =
     C1.C1Preview.AlignHorzEnum.Center
     theader.Cells(0, 2).Style.TextAlignHorz =
     C1.C1Preview.AlignHorzEnum.Right
     theader.CellStyle.Font = New Font("Arial", 14)
     ```

   - C#

     ```
     // Set up alignment for the columns of the header.
     theader.Cells[0, 0].Style.TextAlignHorz =
     C1.C1Preview.AlignHorzEnum.Left;
     theader.Cells[0, 1].Style.TextAlignHorz =
     C1.C1Preview.AlignHorzEnum.Center;
     theader.Cells[0, 2].Style.TextAlignHorz =
     C1.C1Preview.AlignHorzEnum.Right;
     theader.CellStyle.Font = new Font("Arial", 14);
     ```

7. We will draw the text into each column of the table for the page header. Set the RenderObject property of the document's PageHeader to **theader**. Finish generating the document by calling the EndDoc method.

   - Visual Basic

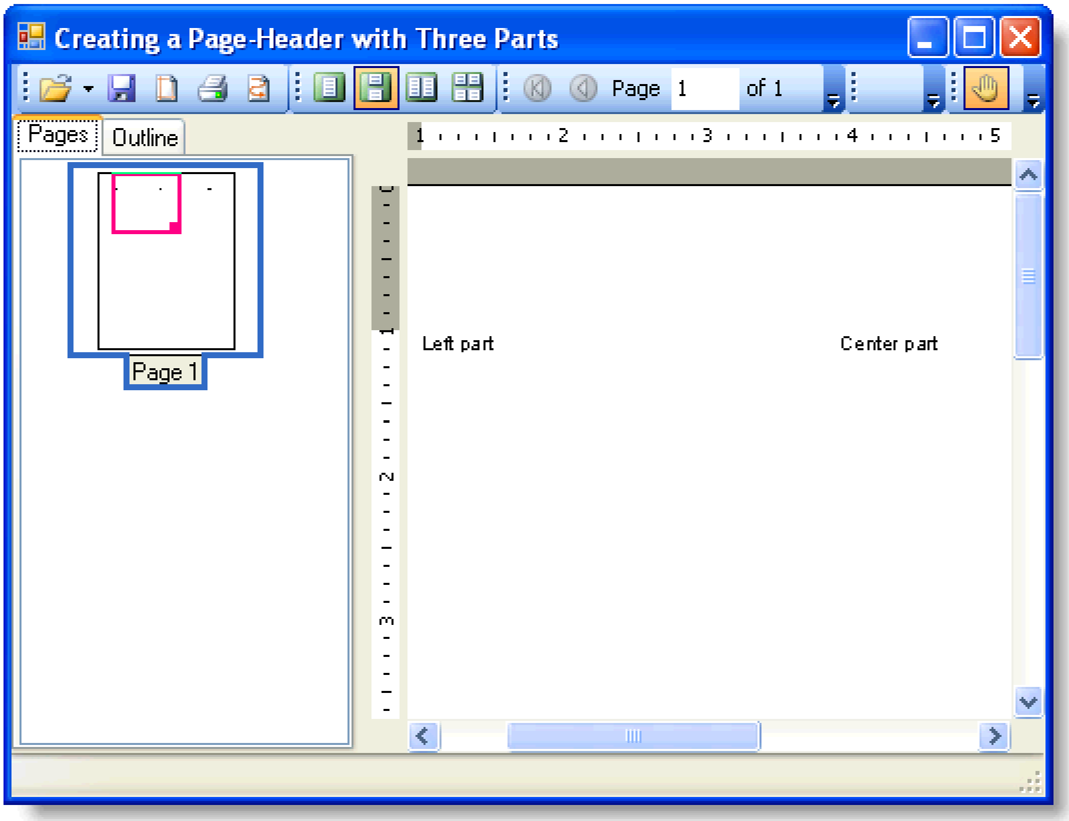
     ```
     theader.Cells(0, 0).Text = "Left part"
     theader.Cells(0, 1).Text = "Center part"
     theader.Cells(0, 2).Text = "Right part"
     Me.C1PrintDocument1.RenderBlock(theader)
     Me.C1PrintDocument1.EndDoc()
     ```

   - C#

     ```
     theader.Cells[0, 0].Text = "Left part";
     theader.Cells[0, 1].Text = "Center part";
     theader.Cells[0, 2].Text = "Right part";
     this.c1PrintDocument1.RenderBlock(theader);
     this.c1PrintDocument1.EndDoc();
     ```

**Run the program and observe the following:**

Your new page header with three parts will appear similar to the header below at run time:



## Adding a Background Color to the Page Header

This topic demonstrates how to add a background color to the page header, and uses the following objects:

- **C1PrintDocument**
- **C1DocStyle**
- **RenderTable**

Complete the following steps:

1. Create a new Windows Forms application.
2. Add a **C1PrintPreview** control onto your form.
3. Add a **C1PrintDocument** component onto your form – it will appear in the components' tray below the form. The preview will have the default name C1PrintPreview1, the document C1PrintDocument1.
4. Set the value of the **Document** property of C1PrintPreview1 control to **C1PrintDocument1**, so that the preview will show the document when the application runs.
5. Double click the form to create a handler for the **Form_Load** event – this is where you will add all the code shown below. Call the StartDoc method to start generating the document. We'll assign the table to the RenderTable class.

   - Visual Basic

     ```
     Me.C1PrintDocument1.StartDoc()
     Dim theader As New C1.C1Preview.RenderTable(Me.C1PrintDocument1)
     ```

   - C#

     ```
     this.c1PrintDocument1.StartDoc();
     ```

```
C1.C1Preview.RenderTable theader = new
C1.C1Preview.RenderTable(this.c1PrintDocument1);
```

6. In this example, we will assign a gold color to our page header in our table. We will use the Style property of the TableRow class to apply the background color to the table, theader.

   - Visual Basic

   ```
   theader.Style.BackColor = Color.Gold
   ```

   - C#

   ```
   theader.Style.BackColor = Color.Gold;
   ```

7. We'll need to create a table with one row and one column. We will draw some text onto our table for our page header. We will make our text right-aligned. We create a new font type of Arial and font size of 14 points, for our text.

   - Visual Basic

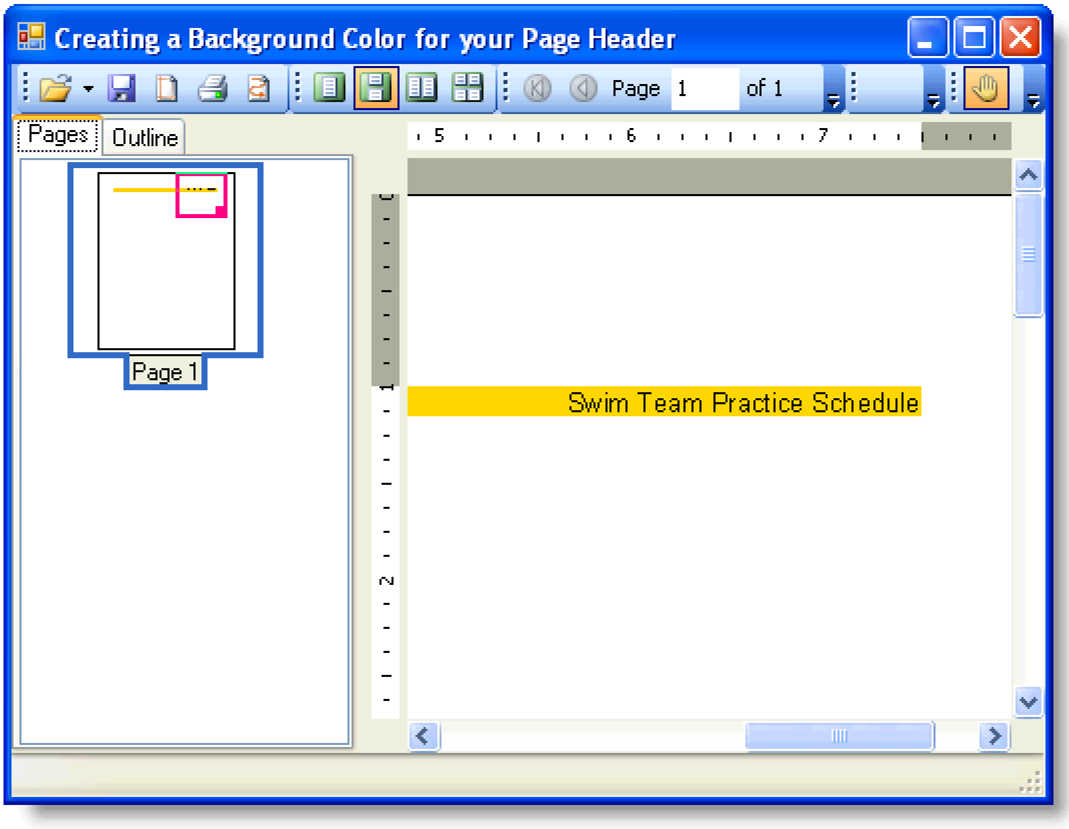
   ```
   theader.Cells(0, 0).Style.TextAlignHorz =
   C1.C1Preview.AlignHorzEnum.Right

   ' Set up some styles.
   theader.CellStyle.Font = New Font("Arial", 14)

   ' Assign text.
   theader.Cells(0, 0).Text = "Swim Team Practice Schedule"

   ' Set the table as the page header.
   Me.C1PrintDocument1.RenderBlock(theader)
   Me.C1PrintDocument1.EndDoc()
   ```

   - C#

   ```
   theader.Cells[0, 0].Style.TextAlignHorz =
   C1.C1Preview.AlignHorzEnum.Right;

   // Set up some styles.
   theader.CellStyle.Font = new Font("Arial", 14);

   // Assign text.
   theader.Cells[0, 0].Text = "Swim Team Practice Schedule";

   // Set the table as the page header.
   this.c1PrintDocument1.RenderBlock(theader);
   this.c1PrintDocument1.EndDoc();
   ```

**Run the program and observe the following:**

Your header should look similar to the following header:


Creating a Background Color for your Page Header
Page 1 of 1
Pages
Outline
Page 1
Swim Team Practice Schedule

### Drawing a Horizontal Line

This topic shows how to draw a horizontal line in the C1PrintDocument. Complete the following steps:

1. Create a new Windows Forms application.
2. Add a C1PrintPreviewControl control onto your form.
3. Add a C1PrintDocument component onto your form – it will appear in the components' tray below the form. The preview will have the default name C1PrintPreview1, the document C1PrintDocument1.
4. Set the value of the **Document** property of C1PrintPreview1 control to **C1PrintDocument1**, so that the preview will show the document when the application runs.
5. Double click the form to create an event handler for the **Form_Load** event – this is where all code shown below will be written.
6. A line can be drawn in C1PrintDocument using the RenderLine render object. Its two main properties are the two points – the start and the end of the line. Because we will render the line into the block flow, and want it to be horizontal, we leave the Y coordinates of the two points as default zeroes. The start X coordinate is also left at zero. The end coordinate is set to the page width. This will result in a RenderLine object that renders a horizontal line across the whole page at the current Y coordinate of the block flow:

   - Visual Basic

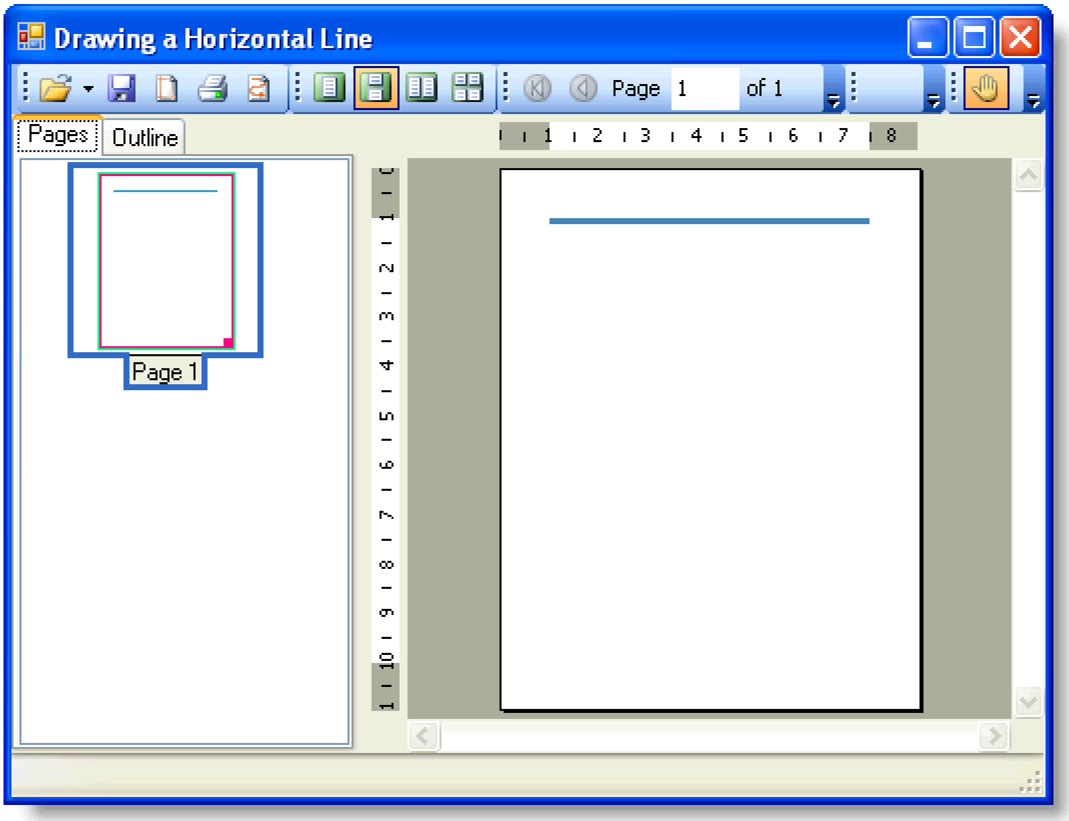
   ```
   Me.C1PrintDocument1.StartDoc()
   Dim rl As New C1.C1Preview.RenderLine(Me.C1PrintDocument1)
   rl.X = Me.C1PrintDocument1.PageLayout.PageSettings.Width
   Dim ld As New C1.C1Preview.LineDef("4mm", Color.SteelBlue)
   Me.C1PrintDocument1.RenderBlockHorzLine(rl.X, ld)
   Me.C1PrintDocument1.EndDoc()
   ```

   - C#

```
this.c1PrintDocument1.StartDoc();
C1.C1Preview.RenderLine rl = new
C1.C1Preview.RenderLine(this.c1PrintDocument1);
rl.X = this.c1PrintDocument1.PageLayout.PageSettings.Width;
C1.C1Preview.LineDef ld = new C1.C1Preview.LineDef("4mm",
Color.SteelBlue);
this.c1PrintDocument1.RenderBlockHorzLine(rl.X, ld);
this.c1PrintDocument1.EndDoc();
```

**Run the program and observe the following:**

Your document will appear similar to the document below:



## *Creating Page Footers*

This topic demonstrates how to create a table footer that is divided into two columns. The following key points are shown in this topic:

- Adding a footer to a table with multiple rows and columns in C1PrintDocument.
- Setting up the table footer at the end of each page.

  The **Count** property of the TableVectorCollection class is used to insert the footer at the end of the table on each page.
- Setting up the row and column spans for each section of the page footer.

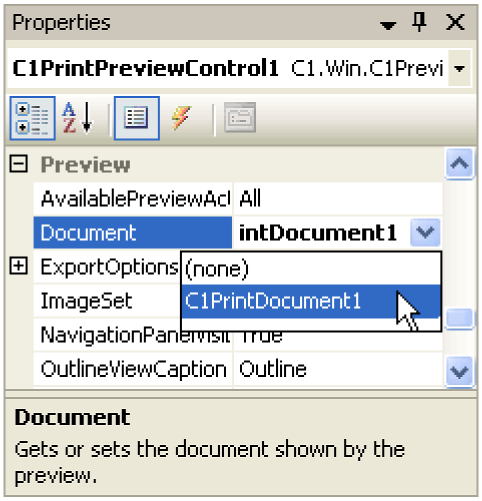
  The SpanRows and SpanCols properties of the TableCell class are used to specify the row and column spans.
- Setting up the text alignment in each section of the page footer.

The TextAlignHorz and TextAlignVert properties of the Style class are used to specify the horizontal and vertical alignment of the text. You can assign a member (left, right, justify, or center) of the AlignHorzEnum to the TextAlignHorz property or a member (bottom, center, justify, or top) of the AlignVertEnum to the TextAlignVert property.

**Note:** The sample code fragments in this topic assume that the "using C1.C1Preview;" directive (in C# syntax; or an equivalent for other languages) has been inserted into the file, so that instead of fully qualified type names (such as C1.C1Preview.RenderText) we can use just the class name part (RenderText).

Complete the following steps to create a table footer with two parts:

1. Create a new Windows Forms application.
2. From the Toolbox, add a C1PrintPreview control onto your form. Add a C1PrintDocument component onto your form – it will appear in the components' tray below the form. The preview will have the default name **C1PrintPreview1**, the document **C1PrintDocument1**.
3. Set the value of the **Document** property of C1PrintPreview1 control to **C1PrintDocument1**, so that the preview will show the document when the application runs.



Properties

C1PrintPreviewControl1 C1.Win.C1Previ

Preview

AvailablePreviewAct All

Document intDocument1

ExportOptions (none)

ImageSet C1PrintDocument1

NavigationPanelVisi True

OutlineViewCaption Outline

**Document**

Gets or sets the document shown by the preview.

4. Double click the form to switch to code view and create a handler for the **Form_Load** event. In the **Form_Load** event, we will set up our document.
5. Add the following code to the **Form_Load** event to create a RenderTable for the page footer and add a table with 4 columns, 100 rows, and sample text:

- Visual Basic

```
Dim rt1 As New C1.C1Preview.RenderTable(Me.C1PrintDocument1)

' Create a table with 100 rows, 4 columns, and text.
Dim r As Integer = 100
Dim c As Integer = 4
Dim row As Integer
Dim col As Integer
For row = 0 To r - 1 Step +1
    For col = 0 To c - 1 Step +1
        Dim celltext As C1.C1Preview.RenderText = New
C1.C1Preview.RenderText(Me.C1PrintDocument1)
        celltext.Text = String.Format("Cell ({0},{1})", row, col)
        rt1.Cells(row, col).RenderObject = celltext
    Next
Next

' Add the table to the document.
Me.C1PrintDocument1.Body.Children.Add(rt1)
```

- C#

```
C1.C1Preview.RenderTable rt1 = new
C1.C1Preview.RenderTable(this.c1PrintDocument1);

// Create a table with 100 rows, 4 columns, and text.
const int r = 100;
const int c = 4;
for (int row = 0; row < r; ++row)
{
    for (int col = 0; col < c; ++col)
    {
        C1.C1Preview.RenderText celltext = new
C1.C1Preview.RenderText(this.c1PrintDocument1);
        celltext.Text = string.Format("Cell ({0},{1})", row, col);
        rt1.Cells[row, col].RenderObject = celltext;
    }
}

// Add the table to the document.
this.c1PrintDocument1.Body.Children.Add(rt1);
```

6. Add the following code to set the font type to Arial, 10 points and the background color to lemon chiffon:

   - Visual Basic

   ```
   ' Set up the table footer.
   rt1.RowGroups(rt1.Rows.Count - 2, 2).PageFooter = True
   rt1.RowGroups(rt1.Rows.Count - 2, 2).Style.BackColor =
   Color.LemonChiffon
   rt1.RowGroups(rt1.Rows.Count - 2, 2).Style.Font = New Font("Arial", 10,
   FontStyle.Bold)
   ```

   - C#

   ```
   // Set up the table footer.
   rt1.RowGroups[rt1.Rows.Count - 2, 2].PageFooter = true;
   rt1.RowGroups[rt1.Rows.Count - 2, 2].Style.BackColor =
   Color.LemonChiffon;
   rt1.RowGroups[rt1.Rows.Count - 2, 2].Style.Font = new Font("Arial", 10,
   FontStyle.Bold);
   ```

   Here we reserved the last two rows of the page for the footer using the **Count** property and grouped the rows together using the **RowGroups** property. We then assigned a new font style for the text and a new background color for the cells in our page footer.

7. Next, we'll use the TextAlignHorz and TextAlignVert properties to set the alignment of the text in each column of the page footer. We'll span the footer over the last two rows and create two columns using the SpanRows and SpanCols properties. We will draw the text into each column of the table for the page footer. Finish by using the Generate method to create the document.

   - Visual Basic

   ```
   ' Add table footer text.
   rt1.Cells(rt1.Rows.Count - 2, 0).SpanRows = 2
   rt1.Cells(rt1.Rows.Count - 2, 0).SpanCols = rt1.Cols.Count - 1
   ```

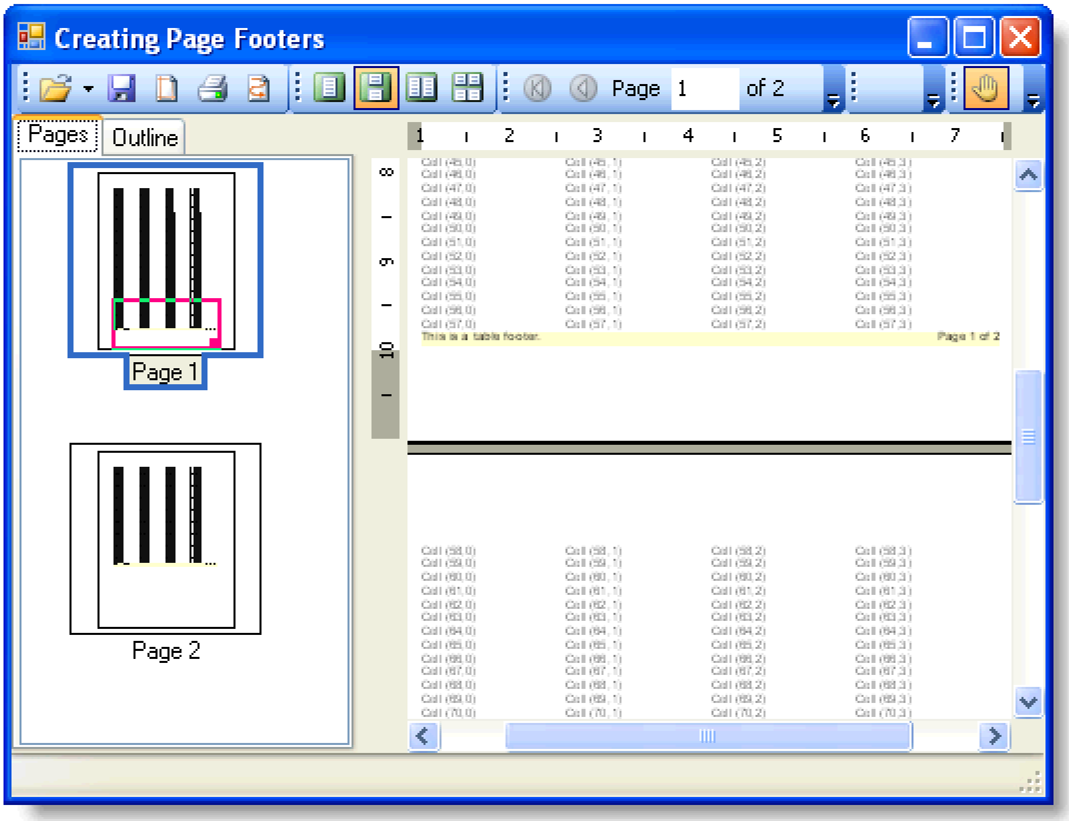
```
rt1.Cells(rt1.Rows.Count - 2, 0).Style.TextAlignHorz =
C1.C1Preview.AlignHorzEnum.Left
rt1.Cells(rt1.Rows.Count - 2, 0).Style.TextAlignVert =
C1.C1Preview.AlignVertEnum.Center
Dim tf As C1.C1Preview.RenderText = New
C1.C1Preview.RenderText(Me.C1PrintDocument1)
tf = CType(rt1.Cells(rt1.Rows.Count - 2, 0).RenderObject,
C1.C1Preview.RenderText)
tf.Text = "This is a table footer."

' Add page numbers.
rt1.Cells(rt1.Rows.Count - 2, 3).SpanRows = 2
rt1.Cells(rt1.Rows.Count - 2, 3).Style.TextAlignHorz =
C1.C1Preview.AlignHorzEnum.Right
rt1.Cells(rt1.Rows.Count - 2, 3).Style.TextAlignVert =
C1.C1Preview.AlignVertEnum.Center

' Tags (such as page no/page count) can be inserted anywhere in the
document.
Dim pn As C1.C1Preview.RenderText = New
C1.C1Preview.RenderText(Me.C1PrintDocument1)
pn = CType(rt1.Cells(rt1.Rows.Count - 2, 3).RenderObject,
C1.C1Preview.RenderText)
pn.Text = "Page [PageNo] of [PageCount]"

Me.C1PrintDocument1.Generate()
```

- C#

```
// Add table footer text.
rt1.Cells[rt1.Rows.Count - 2, 0].SpanRows = 2;
rt1.Cells[rt1.Rows.Count - 2, 0].SpanCols = rt1.Cols.Count - 1;
rt1.Cells[rt1.Rows.Count - 2, 0].Style.TextAlignHorz =
C1.C1Preview.AlignHorzEnum.Center;
rt1.Cells[rt1.Rows.Count - 2, 0].Style.TextAlignVert =
C1.C1Preview.AlignVertEnum.Center;
((C1.C1Preview.RenderText)rt1.Cells[rt1.Rows.Count - 2,
0].RenderObject).Text = "This is a table footer.";

// Add page numbers.
rt1.Cells[rt1.Rows.Count - 2, 3].SpanRows = 2;
rt1.Cells[rt1.Rows.Count - 2, 3].Style.TextAlignHorz =
C1.C1Preview.AlignHorzEnum.Right;
rt1.Cells[rt1.Rows.Count - 2, 3].Style.TextAlignVert =
C1.C1Preview.AlignVertEnum.Center;

// Tags (such as page no/page count) can be inserted anywhere in the
document.
((C1.C1Preview.RenderText)rt1.Cells[rt1.Rows.Count - 2,
3].RenderObject).Text = "Page [PageNo] of [PageCount]";

this.c1PrintDocument1.Generate();
```

**Run the program and observe the following:**

Your new page footer with two parts should appear to the footer below:



| | | | |
|---|---|---|---|
| Cell (45,0) | Cell (45, 1) | Cell (45,2) | Cell (45,3) |
| Cell (46,0) | Cell (46, 1) | Cell (46,2) | Cell (46,3) |
| Cell (47,0) | Cell (47, 1) | Cell (47,2) | Cell (47,3) |
| Cell (48,0) | Cell (48, 1) | Cell (48,2) | Cell (48,3) |
| Cell (49,0) | Cell (49, 1) | Cell (49,2) | Cell (49,3) |
| Cell (50,0) | Cell (50, 1) | Cell (50,2) | Cell (50,3) |
| Cell (51,0) | Cell (51, 1) | Cell (51,2) | Cell (51,3) |
| Cell (52,0) | Cell (52, 1) | Cell (52,2) | Cell (52,3) |
| Cell (53,0) | Cell (53, 1) | Cell (53,2) | Cell (53,3) |
| Cell (54,0) | Cell (54, 1) | Cell (54,2) | Cell (54,3) |
| Cell (55,0) | Cell (55, 1) | Cell (55,2) | Cell (55,3) |
| Cell (56,0) | Cell (56, 1) | Cell (56,2) | Cell (56,3) |
| Cell (57,0) | Cell (57, 1) | Cell (57,2) | Cell (57,3) |

This is a table footer. Page 1 of 2

| | | | |
|---|---|---|---|
| Cell (58,0) | Cell (58, 1) | Cell (58,2) | Cell (58,3) |
| Cell (59,0) | Cell (59, 1) | Cell (59,2) | Cell (59,3) |
| Cell (60,0) | Cell (60, 1) | Cell (60,2) | Cell (60,3) |
| Cell (61,0) | Cell (61, 1) | Cell (61,2) | Cell (61,3) |
| Cell (62,0) | Cell (62, 1) | Cell (62,2) | Cell (62,3) |
| Cell (63,0) | Cell (63, 1) | Cell (63,2) | Cell (63,3) |
| Cell (64,0) | Cell (64, 1) | Cell (64,2) | Cell (64,3) |
| Cell (65,0) | Cell (65, 1) | Cell (65,2) | Cell (65,3) |
| Cell (66,0) | Cell (66, 1) | Cell (66,2) | Cell (66,3) |
| Cell (67,0) | Cell (67, 1) | Cell (67,2) | Cell (67,3) |
| Cell (68,0) | Cell (68, 1) | Cell (68,2) | Cell (68,3) |
| Cell (69,0) | Cell (69, 1) | Cell (69,2) | Cell (69,3) |
| Cell (70,0) | Cell (70, 1) | Cell (70,2) | Cell (70,3) |

# Reports for WinForms Top Tips

This topic lists tips and best practices that may be helpful when working with **ComponentOne Reports for WinForms**. The following tips were compiled from frequently asked user questions posted in the C1DataObjects newsgroup and forum. Additionally the tips detail some newer and/or less used but very useful features.

The following topics detail tips for the C1PrintDocument component, tips for the C1Report component, and tips for the WinForms preview controls (C1PrintPreviewControl and so on). Note that C1PrintDocument and C1Report components' tips may apply also to **ComponentOne Reports for WPF** and C1Report tips may apply to **ComponentOne WebReports for ASP.NET**.

## C1PrintDocument Tips

The following tips relate to the C1PrintDocument component.

### Tip 1: Setting RenderTable Width and Auto-sizing

When setting up RenderTable objects it is important to take into account that by default, the width of a RenderTable is set to the width of its containing object – which usually defaults to the width of the page. This sometimes leads to unexpected results, in particular if you want the table's columns to auto-size; to do so the table's **Width** must be explicitly set to **Auto** (which for a table means the sum of the columns' widths).

For example, the following code builds a document with an auto-sized table:

- Visual Basic

```
Dim doc As New C1PrintDocument()

' Create a RenderTable object:
Dim rt As New RenderTable()

' Adjust table's properties so that columns are auto-sized:
' 1) By default, table width is set to parent (page) width,
' for auto-sizing we must change it to auto (so based on content):
rt.Width = Unit.Auto
' 2) Set ColumnSizingMode to Auto (default means Fixed for columns):
rt.ColumnSizingMode = TableSizingModeEnum.Auto
' That's it, now the table's columns will be auto-sized.

' Turn table grid lines on to better see auto-sizing, add some padding:
rt.Style.GridLines.All = LineDef.[Default]
rt.CellStyle.Padding.All = "2mm"

' Add the table to the document
doc.Body.Children.Add(rt)

' Add some data
rt.Cells(0, 0).Text = "aaa"
rt.Cells(0, 1).Text = "bbbbbbbbbb"
rt.Cells(0, 2).Text = "cccccc"
rt.Cells(1, 0).Text = "aaa aaa aaa"
rt.Cells(1, 1).Text = "bbbbb"
rt.Cells(1, 2).Text = "cccccc cccccc"
rt.Cells(2, 2).Text = "zzzzzzzzzzzzzzz zz z"
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
```

```
// Create a RenderTable object:
 RenderTable rt = new RenderTable();

// adjust table's properties so that columns are auto-sized:
// 1) By default, table width is set to parent (page) width,
// for auto-sizing you must change it to auto (so based on content):
rt.Width = Unit.Auto;
// 2) Set ColumnSizingMode to Auto (default means Fixed for columns):
rt.ColumnSizingMode = TableSizingModeEnum.Auto;
// That's it, now the table's columns will be auto-sized.

// Turn table grid lines on to better see auto-sizing, add some padding:
rt.Style.GridLines.All = LineDef.Default;
rt.CellStyle.Padding.All = "2mm";

// Add the table to the document
doc.Body.Children.Add(rt);

// Add some data
rt.Cells[0, 0].Text = "aaa";
rt.Cells[0, 1].Text = "bbbbbbbbbb";
rt.Cells[0, 2].Text = "cccccc";
rt.Cells[1, 0].Text = "aaa aaa aaa";
rt.Cells[1, 1].Text = "bbbbb";
rt.Cells[1, 2].Text = "cccccc cccccc";
rt.Cells[2, 2].Text = "zzzzzzzzzzzzzzz zz z";
```

For a complete example see the **AutoSizeTable** sample installed with **ComponentOne Reports for WinForms**.

### Tip 2: Using Parent/Ambient Parent Styles to Optimize Memory Usage

When rendered, paragraphs and other **C1PrintDocument** objects have styles that can be modified "inline". For example, like this:

- Visual Basic

```
Dim RenderText rt As New RenderText("testing...")
rt.Style.TextColor = Color.Red
```

- C#

```
RenderText rt = new RenderText("testing...");
rt.Style.TextColor = Color.Red;
```

This is fine for small documents or styles used just once in the whole document. For large documents and styles used throughout the document, it is much better to use parent styles using the following pattern:

1. Identify the styles you will need. For instance, if you are building a code pretty printing application, you may need the following styles:

   - Default code style

   - Language keyword style

   - Comments style

2. Add a child style to the document's root **Style** for each of the styles identified in step 1, like this for example:

   - Visual Basic

   ```
   Dim doc As New C1PrintDocument()
   ' Add and set up default code style:
   Dim sDefault As Style = doc.Style.Children.Add()
   sDefault.FontName = "Courier New"
   ```

```
sDefault.FontSize = 10
' Add and set up keyword style:
Dim sKeyword As Style = doc.Style.Children.Add()
sKeyword.FontName = "Courier New"
sKeyword.FontSize = 10
sKeyword.TextColor = Color.Blue
' Add and set up comments style:
Dim sComment As Style = doc.Style.Children.Add()
sComment.FontName = "Courier New"
sComment.FontSize = 10
sComment.FontItalic = True
sComment.TextColor = Color.Green
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
// Add and set up default code style:
Style sDefault = doc.Style.Children.Add();
sDefault.FontName = "Courier New";
sDefault.FontSize = 10;
// Add and set up keyword style:
Style sKeyword = doc.Style.Children.Add();
sKeyword.FontName = "Courier New";
sKeyword.FontSize = 10;
sKeyword.TextColor = Color.Blue;
// Add and set up comments style:
Style sComment = doc.Style.Children.Add();
sComment.FontName = "Courier New";
sComment.FontSize = 10;
sComment.FontItalic = true;
sComment.TextColor = Color.Green;
```

3. In your code, whenever you create a **C1PrintDocument** element representing a part of source code you're pretty printing, assign the corresponding style to the element style's Parent, for example:

- Visual Basic

```
Dim codeLine As New RenderParagraph()
MessageBox.Show("Hello World!")
' say hi to the world
Dim p1 As New ParagraphText("MessageBox")
p1.Style.AmbientParent = sKeyword
codeLine.Content.Add(p1)
Dim p2 As New ParagraphText(".Show(""Hello World!""); ")
p2.Style.AmbientParent = sDefault
codeLine.Content.Add(p2)
Dim p3 As New ParagraphText("// say hi to the world")
p3.Style.AmbientParent = sComment
codeLine.Content.Add(p3)
doc.Body.Children.Add(codeLine)
```

- C#

```
RenderParagraph codeLine = new RenderParagraph();
MessageBox.Show("Hello World!"); // say hi to the world
ParagraphText p1 = new ParagraphText("MessageBox");
p1.Style.AmbientParent = sKeyword;
codeLine.Content.Add(p1);
ParagraphText p2 = new ParagraphText(".Show(\"Hello World!\"); ");
p2.Style.AmbientParent = sDefault;
codeLine.Content.Add(p2);
```

```
ParagraphText p3 = new ParagraphText("// say hi to the world");
p3.Style.AmbientParent = sComment;
codeLine.Content.Add(p3);
doc.Body.Children.Add(codeLine);
```

That's it, you're done. If you consistently assign your predefined styles to **AmbientParent** (or **Parent**, see below) properties of various document elements, your code will be more memory efficient (and more easily manageable).

You may have noted that you assigned your predefined styles to the **AmbientParent** property of the elements' styles. Remember, in **C1PrintDocument** styles, ambient properties affect content of elements, and by default propagate via elements' hierarchies – so nested objects inherit ambient style properties from their parents (unless a style's **AmbientParent** property is explicitly set). In contrast to that, non-ambient properties affect elements' "decorations" and propagate via styles' own hierarchy determined by styles' parents – so for a non-ambient style property to affect a child object, its style's **Parent** property must be set.

The usefulness of this distinction is best demonstrated by an example: suppose you have a **RenderArea** containing a number of **RenderText** objects. To draw a border around the whole render area you would set the area's **Style.Borders**. Because **Borders** is a non-ambient property, it will draw the border around the area but will not propagate to the nested text objects and will not draw borders around each text – which is normally what you'd want. On the other hand, to set the font used to draw all texts within the area, you again would set the area's **Style.Font**. Because **Font**, unlike **Borders**, is an ambient property it will propagate to all nested text objects and affect them – again usually achieving the desired result. So when you are not setting styles' parent/ambient parent properties – things normally "just work". But when you do use styles' parents – you must take the distinction between ambient and non-ambient style properties into consideration.

Note that for cases when you want to affect both ambient and non-ambient properties of an object, you may use the **Style.Parents** (note the plural) property – it sets both **Parent** and **AmbientParent** properties on a style to the specified value.

### Tip 3: Using Expressions to Customize Page Headers

Sometimes it is necessary to use a different page header for the first or last page of a document. While C1PrintDocument provides a special feature for that (see the PageLayouts – note the plural – property), for cases when the difference between the header on the first and subsequent pages is only in the header text, using an expression may be the best approach. For instance if you want to print "First page" as the first page's header and "Page x of y" for other pages, the following code may be used:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.PageLayout.PageHeader = New RenderText("[iif(PageNo=1, ""First page"",
""Page "" & PageNo & "" of "" & PageCount)]")
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.PageLayout.PageHeader = new RenderText(
  "[iif(PageNo=1, \"First page\", \"Page \" & PageNo & \" of \" &
PageCount)]");
```

In the string representing the expression in the code above, the whole expression is enclosed in square brackets – they indicate to the document rendering engine that what is inside should be treated as an expression (those are adjustable via **TagOpenParen** and **TagCloseParen** properties on the document).

Whatever is inside those brackets should represent a valid expression in the current **C1PrintDocument**'s script/expression language. By default it is VB.NET (but can be changed to C# – see below), hence you use a VB.NET iif function to adjust your page header text depending on the page number. Here's the expression that is actually seen/executed by the document engine:

```
iif(PageNo=1, "First page", "Page " & PageNo & " of " & PageCount)
```

Because you must specify this expression as a C# or VB.NET string when assigning it to the page header text, you have to escape double quotes. Variables **PageNo** and **PageCount** are provided by the document engine (for a complete list of special variables accessible in different contexts in expressions, see the Expressions, Scripts, Tags topic).

As was mentioned, the default expression/script language used by C1PrintDocument is VB.NET. But C# can also be used as the expression language. For that, the C1PrintDocument's Language property must be set to **C1.C1Preview.Scripting.ScriptLanguageEnum.CSharp**. Using C# as the expression language, our example would look like this:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.ScriptingOptions.Language =
C1.C1Preview.Scripting.ScriptLanguageEnum.CSharp
doc.PageLayout.PageHeader = New RenderText("[PageNo==1 ? ""First page"" :
""Page "" + PageNo + "" of "" + PageCount]")
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.ScriptingOptions.Language =
C1.C1Preview.Scripting.ScriptLanguageEnum.CSharp;
doc.PageLayout.PageHeader = new RenderText(
  "[PageNo==1 ? \"First page\" : \"Page \" + PageNo + \" of \" +
PageCount]");
```

There were two changes:

- Instead of VB.NET iif function, the C# conditional operator (:?) was used.
- Instead of VB.NET's string concatenation operator (&), C#'s (+) was used.

Note that expressions are real .NET language expressions, and all normally accessible features of the corresponding language may be used in expressions. For instance instead of string concatenation you could have used the **string.Format** method as follows:

- Visual Basic

```
doc.PageLayout.PageHeader = New RenderText("[iif(PageNo=1, ""First page"",
" & "string.Format(""Page {0} of {1}"", PageNo, PageCount))]")
```

- C#

```
doc.PageLayout.PageHeader = new RenderText("[iif(PageNo=1, \"First page\",
" + "string.Format(\"Page {0} of {1}\", PageNo, PageCount))]");
```

### Tip 4: Data Binding and Expressions

Databound render objects together with expressions are among the less known but extremely powerful C1PrintDocument features. In this example, you'll build a document with a render table data bound to a list of objects in memory. The list elements will represent **Customer** records with just two (for brevity) fields – **Name** and **Balance**:

- Visual Basic

```
Public Class Customer
    Private _name As String
    Private _balance As Integer
    Public Sub New(ByVal name As String, ByVal balance As Integer)
        _name = name
        _balance = balance
    End Sub
    Public ReadOnly Property Name() As String
        Get
            Return _name
        End Get
    End Property
    Public ReadOnly Property Balance() As Integer
```

```
        Get
            Return _balance
        End Get
    End Property
End Class
```

- C#

```
public class Customer
{
  private string _name;
  private int _balance;
  public Customer(string name, int balance)
  {
    _name = name;
    _balance = balance;
  }
  public string Name { get { return _name; } }
  public int Balance { get { return _balance; } }
}
```

The following code can be used to create a list of customer records and fill it with some sample data:

- Visual Basic

```
' build sample list of customers
Dim customers As New List(Of Customer)()
Dim rnd As New Random(DateTime.Now.Second)
For i As Integer = 0 To 599
    customers.Add(New Customer("Customer_" & (i + 1).ToString(),
rnd.[Next](-1000, 1000)))
Next
```

- C#

```
// build sample list of customers
List<Customer> customers = new List<Customer>();
Random rnd = new Random(DateTime.Now.Second);
  for (int i = 0; i < 600; i++)
    customers.Add(new Customer("Customer_" + (i+1).ToString(), rnd.Next(-
1000, 1000)));
```

Note that the **Balance** field's value ranges from -1000 to 1000 – so the field allows negative values. This will be used to demonstrate a new C1PrintDocument feature, style expressions, below.

The following code prints the list created above as a **RenderTable** in a C1PrintDocument:

- Visual Basic

```
Dim doc As New C1PrintDocument()
Dim rt As New RenderTable()
' Define data binding on table rows:
rt.RowGroups(0, 1).DataBinding.DataSource = customers
' Bind column 0 to Name:
rt.Cells(0, 0).Text = "[Fields!Name.Value]"
' Bind column 1 to Balance:
rt.Cells(0, 1).Text = "[Fields(""Balance"").Value]"
' Add the table to the document:
doc.Body.Children.Add(rt)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderTable rt = new RenderTable();
// Define data binding on table rows:
```

```
rt.RowGroups[0, 1].DataBinding.DataSource = customers;
// Bind column 0 to Name:
rt.Cells[0, 0].Text = "[Fields!Name.Value]";
// Bind column 1 to Balance:
rt.Cells[0, 1].Text = "[Fields(\"Balance\").Value]";
// Add the table to the document:
doc.Body.Children.Add(rt);
```

Databinding is achieved with just 3 lines of code. The first line defines a row group on the table, starting at row 0 and including just that one row:

- Visual Basic

```
' Define data binding on table rows:
rt.RowGroups(0, 1).DataBinding.DataSource = customers)
```

- C#

```
// Define data binding on table rows:
rt.RowGroups[0, 1].DataBinding.DataSource = customers;
```

The other two lines show two syntactically different but equivalent ways of binding a table cell to a data field:

- Visual Basic

```
' Bind column 0 to Name:
rt.Cells(0, 0).Text = "[Fields!Name.Value]"
' Bind column 1 to Balance:
rt.Cells(0, 1).Text = "[Fields(""Balance"").Value]"
```

- C#

```
// Bind column 0 to Name:
rt.Cells[0, 0].Text = "[Fields!Name.Value]";
// Bind column 1 to Balance:
rt.Cells[0, 1].Text = "[Fields(\"Balance\").Value]";
```

As noted, the "Fields!Name" notation is just syntactic sugar for referencing the element called **Name** in the **Fields** array, and allows to avoid the need to use escaped double quotes.

Now, remember that the **Balance** field in the sample data set can be positive or negative. The following line will make all negative **Balance** values appear red colored in the document:

- Visual Basic

```
rt.Cells(0, 1).Style.TextColorExpr = "iif(Fields!Balance.Value < 0,
Color.Red, Color.Blue)"
```

- C#

```
rt.Cells[0, 1].Style.TextColorExpr = "iif(Fields!Balance.Value < 0,
Color.Red, Color.Blue)";
```

This demonstrates a new C1PrintDocument feature – style expressions. Starting with 2009 v3 release, all style properties have matching expression properties (ending in "Expr"), which allow you to define an expression that would be used at run time to calculate the effective corresponding style property. While this feature is independent of data binding, it can be especially useful in data bound documents as shown here.

Style expressions allow the use of predefined C1PrintDocument tags related to pagination. For instance, the following code may be used to print a render object ro on red background if it appears on page with number greater than 10 and on green background otherwise:

- Visual Basic

```
ro.Style.BackColorExpr = "[iif(PageNo > 10, Color.Red, Color.Green)]"
```

- C#

```
ro.Style.BackColorExpr = "[iif(PageNo > 10, Color.Red, Color.Green)]";
```

Finally, it should be noted that while VB.NET is the default expression language in C1PrintDocument, C# can be used instead if the Language property is set on the document:

- Visual Basic

```
doc.ScriptingOptions.Language =
C1.C1Preview.Scripting.ScriptLanguageEnum.CSharp
```

- C#

```
doc.ScriptingOptions.Language =
C1.C1Preview.Scripting.ScriptLanguageEnum.CSharp;
```

With this in mind, our current sample may be rewritten as follows:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.ScriptingOptions.Language =
C1.C1Preview.Scripting.ScriptLanguageEnum.CSharp
Dim rt As New RenderTable()
' Define data binding on table rows:
rt.RowGroups(0, 1).DataBinding.DataSource = customers
' Bind column 0 to Name:
rt.Cells(0, 0).Text = " [Fields[""Name""].Value]"
' Bind column 1 to Balance:
rt.Cells(0, 1).Text = " [Fields[""Balance""].Value]"
rt.Cells(0, 1).Style.TextColorExpr = "(int)(Fields[""Balance""].Value) < 0
? Color.Red : Color.Blue"
' Add the table to the document:
doc.Body.Children.Add(rt)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.ScriptingOptions.Language =
C1.C1Preview.Scripting.ScriptLanguageEnum.CSharp;
RenderTable rt = new RenderTable();
// Define data binding on table rows:
rt.RowGroups[0, 1].DataBinding.DataSource = customers;
// Bind column 0 to Name:
rt.Cells[0, 0].Text = " [Fields[\"Name\"].Value]";
// Bind column 1 to Balance:
rt.Cells[0, 1].Text = " [Fields[\"Balance\"].Value]";
rt.Cells[0, 1].Style.TextColorExpr =
  "(int)(Fields[\"Balance\"].Value) < 0 ? Color.Red : Color.Blue";
// Add the table to the document:
doc.Body.Children.Add(rt);
```

Note the following as compared to code that used VB.NET as expressions/scripting language:

- The use of square brackets as **C1PrintDocument**'s open/close tag parentheses (adjustable via **TagOpenParen** and **TagCloseParen** properties) does not conflict with their C# use for array indexing within the expressions because after seeing the first opening bracket, C1PrintDocument tries to find a matching closing one.
- Because a field's **Value** property is of type object, we need to cast it to an int for the conditional expression to work correctly (VB.NET does that automatically).
- The "Fields!Name" notation cannot be used as it is purely a VB.NET feature.

### Tip 5: Customizing Databound Table Columns

In C1PrintDocument's tables, you can have data bound columns rather than rows. Consider our example from previous section – it only takes a few changes to make the data bound table expand horizontally rather than vertically. Here's the code rewritten to show customer's name in the first row of the table, customer's balance in the second row, with each column corresponding to a customer entry:

- Visual Basic

```
Dim doc As New C1PrintDocument()
Dim rt As New RenderTable()
' Next 3 lines set table up for horizontal expansion:
rt.Width = Unit.Auto
rt.ColumnSizingMode = TableSizingModeEnum.Auto
rt.SplitHorzBehavior = SplitBehaviorEnum.SplitIfNeeded
' Define data binding on table columns:
rt.ColGroups(0, 1).DataBinding.DataSource = customers
' Bind column 0 to Name:
rt.Cells(0, 0).Text = "[Fields!Name.Value]"
' Bind column 1 to Balance:
rt.Cells(1, 1).Text = "[Fields(""Balance"").Value]"
' Print negative values in red, positive in blue:
rt.Cells(0, 1).Style.TextColorExpr = "iif(Fields!Balance.Value < 0,
Color.Red, Color.Blue)"
' Add the table to the document:
doc.Body.Children.Add(rt)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderTable rt = new RenderTable();
// Next 3 lines set table up for horizontal expansion:
rt.Width = Unit.Auto;
rt.ColumnSizingMode = TableSizingModeEnum.Auto;
rt.SplitHorzBehavior = SplitBehaviorEnum.SplitIfNeeded;
// Define data binding on table columns:
rt.ColGroups[0, 1].DataBinding.DataSource = customers;
// Bind column 0 to Name:
rt.Cells[0, 0].Text = "[Fields!Name.Value]";
// Bind column 1 to Balance:
rt.Cells[1, 1].Text = "[Fields(\"Balance\").Value]";
// Print negative values in red, positive in blue:
rt.Cells[0, 1].Style.TextColorExpr =
  "iif(Fields!Balance.Value < 0, Color.Red, Color.Blue)";
// Add the table to the document:
doc.Body.Children.Add(rt);
```

Note the following changes:

- The three lines of code following the comment starting with "next 3 lines" ensure that the table's width is based on the sum of columns' widths, that the columns' width are based on their content, and that the table is allowed to split horizontally if it becomes too wide to fit on a single page.
- Aside from that, all other changes are basically the result of swapping rows and columns.

### Tip 6: Including WinForms controls in a document

It is easy to include a snapshot of a WinForms control from your application in a document that is generated. To do that, use a **RenderImage** object and its **Control** property. For instance if your form contains a button button1, this code will include a snapshot of that button within a document:

- Visual Basic

```
Dim doc As New C1PrintDocument()
Dim ri As New RenderImage()
ri.Control = Me.button1
doc.Body.Children.Add(ri)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderImage ri = new RenderImage();
ri.Control = this.button1;
doc.Body.Children.Add(ri);
```

## C1Report Tips

The following tips relate to the C1Report component.

**Tip 1: Converting Microsoft Access Reports**

One of the most powerful features of the **C1ReportDesigner** application is the ability to import reports created with Microsoft Access. To import reports from an Access file, click the **Application** button and select **Import** from the menu. In the dialog box, select a Microsoft Access file (MDB or ADP) and reports.

Note that you must have Access installed on the computer to convert an Access report. Once the report is imported into the Designer, Access is no longer required. For details and more information, see Importing Microsoft Access Reports.

**Tip 2: Styling Your Reports**

You can use the **Style Gallery** in the **C1ReportDesigner** application to style your report using one of over 30 built-in styles. You can also create your own custom report styles. Built-in styles include standard Microsoft AutoFormat themes, including Vista and Office 2007 themes. You can access the **Style Gallery** from the **C1ReportDesigner** application by selecting the **Arrange** tab and clicking **Styles**. For details see Style Gallery.

**Tip 3: Using Script Expressions**

Expressions are widely used throughout a report definition to retrieve, calculate, display, group, sort, filter, parameterize, and format the contents of a report. Some expressions are created for you automatically (for example, when you drag a field from the Toolbox onto a section of your report, an expression that retrieves the value of that field is displayed in the text box). However, in most cases, you create your own expressions to provide more functionality to your report.

For more information about taking advantage of script expressions, see Creating VBScript Expressions, Working with VBScript, and Expressions, Scripts, Tags.

**Tip 4: Creating Subreports**

A subreport is a report that is inserted in another report. Subreports are useful when you want to combine several reports into one. For example, you may have a main report that integrates several subreports into a single main report. Or you can use the main report to show detailed information and use subreports to show summary data at the beginning of each group. For information about subreports, see the Subreport property in the reference section, and for an example see Creating a Master-Detail Report Using Subreports.

**Tip 5: Understanding Reports and Report Definitions**

Reports are based on a report definition, an XML file that describes data and layout. **Reports for WinForms** creates the report definition for you when you add a report item to a project and define the report layout. When report execution is triggered (for example, you provide a button that the user clicks to view a report), the C1Report control retrieves data using the data bindings you have defined and merges the result set into the report layout. The report is presented in the native output format for the control you are using.

You can load report definitions at design time or run time, from XML files, strings, or you can create and customize report definitions dynamically using code.

**Tip 6: CustomFields and Chart/Reports Version**

You can include a chart in a report by adding the **C1Chart** object to **C1Report**'s custom field. This method is demonstrated in detail in the **CustomFields** sample installed with **Reports for WinForms**. Along with the sample, a pre-built **CustomFields** assembly – just as can be built with that sample – is shipped, together with several other

DLLs (such as the **C1Chart** assembly) as part of the **C1ReportDesigner** application installed by default in the **C1Report/Designer** folder in the **Studio for WinForms** installation directory.

You may be tempted to just add a reference to that prebuilt **C1.C1Report.CustomFields.2.dll** (or **C1.C1Report.CustomFields.4.dll**) file to your own project when including a chart in a report in a **C1Report**-based application. While this may work initially, it may cause unexpected problems later. The binary **CustomFields** assembly shipped with the **C1ReportDesigner** application is built with references to specific versions of **C1Report** and **C1Chart** located in the **C1ReportDesigner** application's subfolder. When you install **Studio for WinForms** you will have the same versions of the reports and chart products as the components you may use in your own application development. But if you later update just one assembly, for example just **C1Chart**, the prebuilt **CustomFields** in your application will suddenly no longer work since it will be referencing an outdated **C1Chart** assembly after upgrading the chart.

For that reason it is much better and a best practice to add the actual **CustomFields** project (together with the source code) from the corresponding sample (available in both VB.NET and C# variants) to your own solution, and reference that project instead of the prebuilt binary **CustomFields** assembly. Then upgrading any of the involved ComponentOne products will not break your application.

### Tip 7: Adding a Custom Outline Entry for Each Data Record

Outline entries are automatically generated for group headers in **C1Report**; the AddOutlineEntry event allows customizing the text of those entries. To associate a customized outline entry with each record of a report that otherwise is not using groups, follow the following steps:

1. Create a group that will always contain exactly one record. The easiest way to do that is to group records by a key field.
2. On that group, create a group header with the minimal height of 1 twip so that it won't show in the report. The AddOutlineEntry event will be fired for each instance of that group.
3. Attach a handler to the AddOutlineEntry event to customize the entry's text by modifying the **Text** property on the ReportEventArgs passed to the handler.

This will associate a customized outline entry with each record of a report that otherwise is not using groups.

### Tip 8: Printing Two Subreports Side by Side

While it is possible to create a report definition with two subreports arranged side by side on a page, generally speaking the **C1Report** component cannot properly render such subreports if they take up more than one page and page breaks are involved. Sometimes though, it is possible to render such reports correctly by importing the report definition into a **C1PrintDocument** component and rendering that instead. For example:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.ImportC1Report("myReportFile.xml", "myReportName")
doc.Generate()
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.ImportC1Report("myReportFile.xml", "myReportName");
doc.Generate();
```

See Generating Reports (C1Report vs. C1PrintDocument) and Deciding on Report Generation Method for details concerning advantages and limitations of importing reports into a **C1PrintDocument** instead of using **C1Report** directly.

### Tip 9: Passing Parameters from the Main Report into a Subreport

To pass parameters from the main report to a subreport set the ExposeScriptObjects property to **True** in the main report. When you do so, all fields of the main report will become visible in the subreport.

So, for instance, if you took the CommonTasks.xml report shipped with the **Reports for WinForms** samples, and in report "14: Page Headers in Subreports", set ExposeScriptObjects to **True**, you can use the CategoryName (defined in the main report but not in the subreport) in the subreport's fields.

### Tip 10: Linking a Subreport to a Report Using Multiple Fields

Normally when a subreport is used to print part of a report the linkage between the report and the subreport is maintained via the **Text** property of the field containing the subreport, somewhat like the following example:

```
"SubReportLookupField = \" & ReportKeyField & \""
```

At run time that expression is calculated, yielding something similar to the following:

```
SubReportLookupField = "1"
```

The result is used as a filter for the subreport's data source (RowFilter property).

The interesting thing here is that the initial expression may be more complex and may references to several fields. For instance, suppose the main report has **DateTime** type fields **StartTime** and **EndTime**, while the subreport has an **OpenTime** field, also of the **DateTime** type. Suppose the subreport must show only those records for which the following condition is true:

```
OpenTime >= StartTime and OpenTime <= EndTime
```

In that case the following string may be used as the subreport filter (all on a single line):

```
"OpenTime >= #" & Format(StartTime, \"MM/dd/yyyy\") & "# and OpenTime <= #" &
Format(EndTime, \"MM/dd/yyyy\") & "#"
```

The **Format** function is used here to format **DateTime** values using **InvariantCulture**, as that is the format that should be used for the RowFilter.

## Visual Previewing Control Tips

The following tips relate to the visual previewing controls included in **ComponentOne Reports for WinForms** (such as C1PrintPreviewControl, C1PrintPreviewDialog, C1PreviewPane, C1PreviewThumbnailView, C1PreviewOutlineView, and C1PreviewTextSearchPanel).

### Tip 1: Adding a Custom Toolbar Button to C1PrintPreviewControl

You may want to customize a toolbar in the C1PrintPreviewControl control by adding a custom button. To add your own toolbar button to any of the built-in toolbars on the C1PrintPreviewControl, add a C1PrintPreviewControl to the form, and complete the following steps:

1. Click once on the C1PrintPreview control on the form to select it, and navigate to the Properties window.
2. In the Properties window, locate and expand the **Toolbars** top-level property by clicking on the plus sign (+) to the left of the item.

   You should see the list of predefined preview toolbars: **File**, **Navigation**, **Page**, **Text**, and **Zoom**.
3. Select and expand the toolbar, for instance **File**, that will contain the new item. The expanded list should show the predefined **File** buttons: **Open**, **PageSetup**, **Print**, **Reflow**, **Save**, and a **ToolStrip** expandable item.
4. Select and expand the **ToolStrip** item. The expanded list should contain a single **Items** collection node.
5. Select the **Items** node and click on the **ellipses** (…) button to the right of the item. The items collection editor dialog box containing the predefined items will be displayed.
6. Add a new button using the collection dialog box's commands, adjust its properties as needed, note the name of the button, and press OK to close the dialog box and save the changes. Now you should see the newly added button in the designer view.
7. Select the newly added button in the Properties window (you can do this by either clicking on the button in the designer view, or by selecting the button by its name from the Properties window's drop-down list).

Once you have added the toolbar button, you can customize its actions. You can now add a **Click** event handler to the button or further customize its properties.

### Tip 2: Change the Default Toolbar Button Processing

To change the default processing of a built-in toolbar button on a C1PrintPreviewControl, you need to handle the preview pane's UserAction event.

Add a C1PrintPreviewControl to the form, and complete the following steps:

1. Click on the preview pane within that control (the main area showing the pages of the viewed document) in the Visual Studio designer.

   This will select the C1PreviewPane within the preview control into the properties window. For example, if your preview control has the name **c1PrintPreviewControl1**, the selected object's name should become **c1PrintPreviewControl1.PreviewPane**.

2. Click the lightning bolt icon in the Properties window to view **Events**, scroll to the UserAction event item, and double click the item to create an event handler.

3. You can customize the default processing of a built-in toolbar button in the UserAction event handler.

The handler receives an instance of UserActionEventArgs which contains two interesting properties: PreviewAction and **Cancel**. PreviewAction is an enum listing all supported user events, such as file open, print, and so on. Testing that property you may find the action you're interested in. The other important property of UserActionEventArgs is **Cancel** – if you add your own processing to an action and want to omit the standard processing, set **UserActionEventArgs.Cancel** to **True**, otherwise standard processing will take place after your event handler returns.

### Tip 3: Using Preview Pane's PropertyChanged Event

To monitor interesting preview related properties, use the PropertyChanged event on the C1PreviewPane object within the C1PrintPreviewControl. All of the preview pane's own properties (not inherited from its base control) fire the PropertyChanged event when their values change.

For instance, the following code will add a handler to the PropertyChanged event provided that your form includes a C1PrintPreviewControl named c1PreviewControl1:

- Visual Basic

```
AddHandler Me.C1PrintPreviewControl1.PreviewPane.PropertyChanged,
AddressOf PreviewPane_PropertyChanged
```

- C#

```
this.c1PrintPreviewControl1.PreviewPane.PropertyChanged += new
PropertyChangedEventHandler(PreviewPane_PropertyChanged);
```

The following property changed event handler will print a debug line each time the current page changes in the preview for whatever reason:

- Visual Basic

```
Private Sub PreviewPane_PropertyChanged(ByVal sender As Object, ByVal e As
PropertyChangedEventArgs)
    If e.PropertyName = "StartPageIdx" Then
        Debug.WriteLine("Current page changed to " &
Me.c1PrintPreviewControl1.PreviewPane.StartPageIdx.ToString())
    End If
End Sub
```

- C#

```
void PreviewPane_PropertyChanged(object sender, PropertyChangedEventArgs
e)
{
  if (e.PropertyName == "StartPageIdx")
```

```
    Debug.WriteLine("Current page changed to " +
this.c1PrintPreviewControl1.PreviewPane.StartPageIdx.ToString());
}
```

# Design-Time Support

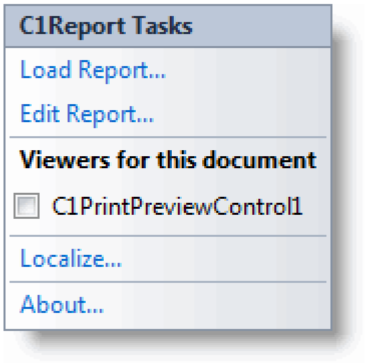
You can easily configure **ComponentOne Reports for WinForms** at design time using the property grid, menus, and designers in Visual Studio. The following sections describe how to use **Reports for WinForms**' design-time environment to configure the **Reports for WinForms** controls.

## C1Report Tasks Menu

In Visual Studio, the C1Report component includes a smart tag. A smart tag represents a short-cut tasks menu that provides the most commonly used properties.

The C1Report component provides quick and easy access to the **C1Report Wizard** (for reports definitions that have not been created) or the **C1ReportDesigner** (for report definitions that already exist in the project), as well as loads reports through its smart tag.

To access the **C1Report Tasks** menu, click the smart tag (▸) in the upper-right corner of the C1Report component.



The **C1Report Tasks** menu operates as follows:

- **Load Report**

  Clicking **Load Report** opens the **Select a report** dialog box. See the Loading a Report Definition from a File topic for more information about the **Select a report** dialog box and loading a report.

- **Edit Report**

  Clicking **Edit Report** opens the **C1Report Wizard** if you have not already created a report definition or the **C1ReportDesigner** if you have already created a report.

  For more information on using the **C1Report Wizard**, see Creating a Basic Report Definition. For details on using the **C1ReportDesigner**, see Working with C1ReportDesigner.

  > **Note:** If the **Edit Report** command doesn't appear on the Tasks menu and Properties window, it is probably because the control could not find the **C1ReportDesigner** application. To fix this, simply run the **C1ReportDesigner** application once in stand-alone mode. The designer will save its location to the registry, and the C1Report component should be able to find it afterwards.

- **Viewers for this document**

  If any visual previewing controls – such as C1PrintPreviewControl control – are included in the application, the **C1Report Tasks Menu** will display them here. When the check box next to the name of a previewing control is checked, the C1Report control is associated with the previewing control and reports will appear in the previewing control.

- **Localize**

  Clicking **Localize** opens the **Localize** dialog box. In the **Localize** dialog box, you can customize your localization settings. For more information on the **Localize** dialog box, see Localization.

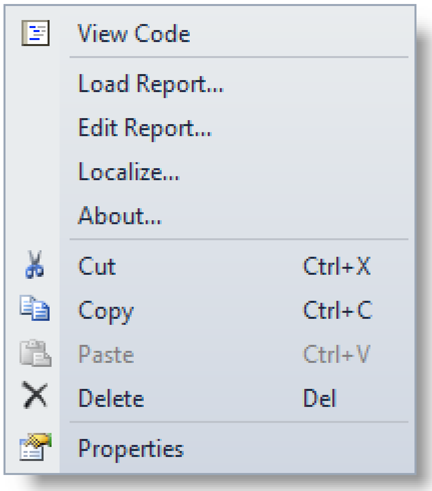
- **About**

  Clicking **About** displays the **About** dialog box, which is helpful in finding the version number of **Reports for WinForms**, as well as online resources.

## C1Report Context Menu

You can also access the **C1Report Wizard** or the **C1ReportDesigner**, as well as load a report by using the C1Report component's context menu.

To access C1Report control's context menu, right-click the C1Report component. The context menu appears:



The C1Report context menu operates as follows:

- **Load Report**

  Clicking **Load Report** opens the **Select a report** dialog box. See the Loading a Report Definition from a File topic for more information about the **Select a report** dialog box and loading a report.

- **Edit Report**

  Clicking **Edit Report** opens the **C1Report Wizard** if you have not already created a report definition or the **C1ReportDesigner** if you have already created a report.

  For more information on using the **C1Report Wizard**, see Creating a Basic Report Definition. For details on using the **C1ReportDesigner**, see Working with C1ReportDesigner.

  > **Note:** If the **Edit Report** command doesn't appear on the context menu and Properties window, it is probably because the control could not find the **C1ReportDesigner** application. To fix this, simply run the **C1ReportDesigner** application once in stand-alone mode. The designer will save its location to the registry, and the C1Report component should be able to find it afterwards.

- **Localize**

  Clicking **Localize** opens the **Localize** dialog box. In the **Localize** dialog box, you can customize your localization settings. For more information on the **Localize** dialog box, see Localization.

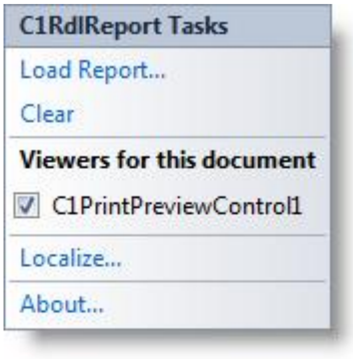
- **About**

  Clicking **About** displays the **About** dialog box, which is helpful in finding the version number of **Reports for WinForms**, as well as online resources.

## C1RdlReport Tasks Menu

In Visual Studio, the C1RdlReport component includes a smart tag. A smart tag represents a short-cut tasks menu that provides the most commonly used properties.

To access the **C1RdlReport Tasks** menu, click the smart tag (▸) in the upper-right corner of the C1RdlReport component.



The **C1RdlReport Tasks** menu operates as follows:

- **Load Report**

  Clicking **Load Report** opens the **Open** dialog box which allows you to select an RDL file to load into the C1RdlReport component.

- **Clear**

  Clicking **Clear** clears any report definition that is currently loaded. After clicking this option, you will see a dialog box asking you to confirm clearing the loaded report definition. In the dialog box, click **Yes** to continue and **No** to not clear the report definition.

- **Viewers for this document**

  If any visual previewing controls – such as C1PrintPreviewControl control – are included in the application, the **C1RdlReport Tasks Menu** will display them here. When the check box next to the name of a previewing control is checked, the C1RdlReport control is associated with the previewing control and reports will appear in the previewing control.

- **Localize**

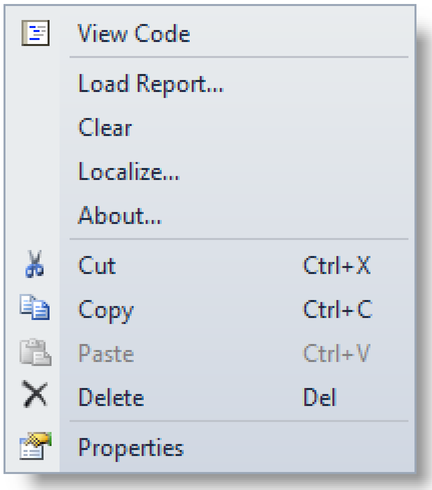
  Clicking **Localize** opens the **Localize** dialog box. In the **Localize** dialog box, you can customize your localization settings. For more information on the **Localize** dialog box, see Localization.

- **About**

  Clicking **About** displays the **About** dialog box, which is helpful in finding the version number of **Reports for WinForms**, as well as online resources.

## C1RdlReport Context Menu

To access the C1RdlReport control's context menu, right-click the C1RdlReport component. The context menu will appear.



The **C1RdlReport** context menu operates as follows:

- **Load Report**

  Clicking **Load Report** opens the **Open** dialog box which allows you to select an RDL file to load into the C1RdlReport component.

- **Clear**

  Clicking **Clear** clears any report definition that is currently loaded. After clicking this option, you will see a dialog box asking you to confirm clearing the loaded report definition. In the dialog box, click **Yes** to continue and **No** to cancel clearing the report definition.

- **Localize**

  Clicking **Localize** opens the **Localize** dialog box. In the **Localize** dialog box, you can customize your localization settings. For more information on the **Localize** dialog box, see Localization.

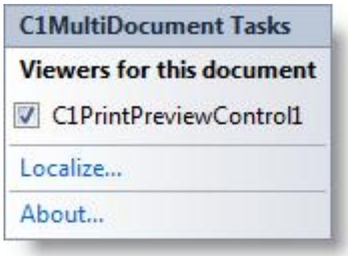
- **About**

  Clicking **About** displays the **About** dialog box, which is helpful in finding the version number of **Reports for WinForms**, as well as online resources.

## C1MultiDocument Tasks Menu

In Visual Studio, the C1MultiDocument component includes a smart tag. A smart tag represents a short-cut tasks menu that provides the most commonly used properties.

To access the **C1MultiDocument Tasks** menu, click the smart tag (▸) in the upper-right corner of the C1MultiDocument component.



The **C1MultiDocument Tasks** menu operates as follows:

- **Viewers for this document**

  If any visual previewing controls – such as C1PrintPreviewControl control – are included in the application, the **C1MultiDocument Tasks Menu** will display them here. When the check box next to the name of a previewing control is checked, the C1MultiDocument component is associated with the previewing control and reports will appear in the previewing control.

- **Localize**

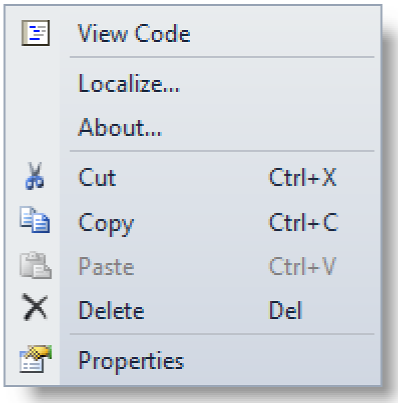
  Clicking **Localize** opens the **Localize** dialog box. In the **Localize** dialog box, you can customize your localization settings. For more information on the **Localize** dialog box, see Localization.

- **About**

  Clicking **About** displays the **About** dialog box, which is helpful in finding the version number of **Reports for WinForms**, as well as online resources.

## C1MultiDocument Context Menu

To access the C1MultiDocument component's context menu, right-click the C1MultiDocument component. The context menu will appear.



The C1MultiDocument context menu operates as follows:

- **Localize**

  Clicking **Localize** opens the **Localize** dialog box. In the **Localize** dialog box, you can customize your localization settings. For more information on the **Localize** dialog box, see Localization.

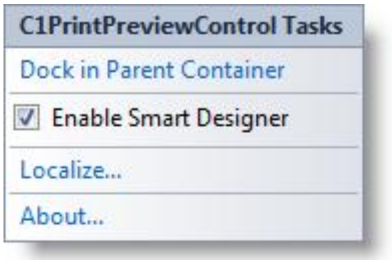
- **About**

  Clicking **About** displays the **About** dialog box, which is helpful in finding the version number of **Reports for WinForms**, as well as online resources.

## C1PrintPreviewControl Tasks Menu

In the **C1PrintPreviewControl Tasks** menu, you can quickly and easily dock the C1PrintPreviewControl in the parent container, enable the **Smart Designer**, and access the **Localize** dialog box.

To access the **C1PrintPreviewControl Tasks** menu, click the smart tag (▶) in the upper right corner of the control. This will open the **C1PrintPreviewControl Tasks** menu.



C1PrintPreviewControl Tasks

Dock in Parent Container

☑ Enable Smart Designer

Localize...

About...

The **C1PrintPreviewControl Tasks** menu operates as follows:

- **Dock in parent container/Undock in parent container**

  Clicking **Dock in parent container** sets the **Dock** property for C1PrintPreviewControl to **Fill**.

  If **C1PrintPreviewControl** is docked in the parent container, the option to undock C1PrintPreviewControl from the parent container will be available. **Clicking Undock in parent container** sets the **Dock** property for **C1PrintPreviewControl** to **None**.

- **Enable Smart Designer**

  Selecting the **Enable Smart Designer** check box activates the **Smart Designer** on the **C1PrintPreviewControl** for greater design time interaction. By default the **Enable Smart Designer** check box is checked and the smart designer is enabled. For more information on the **Smart Designer**, see Smart Designers.

- **Localize**

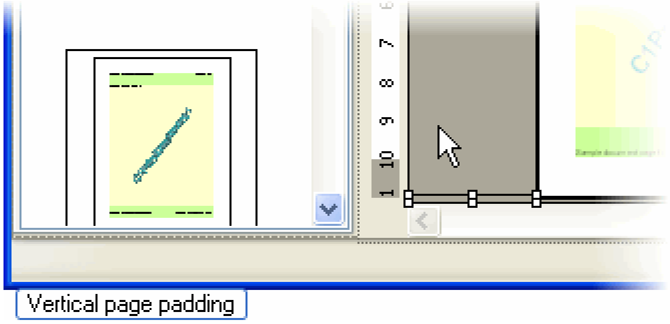
  Clicking **Localize** opens the **Localize** dialog box. In the **Localize** dialog box, you can customize your localization settings. For more information on the **Localize** dialog box, see Localization.

- **About**

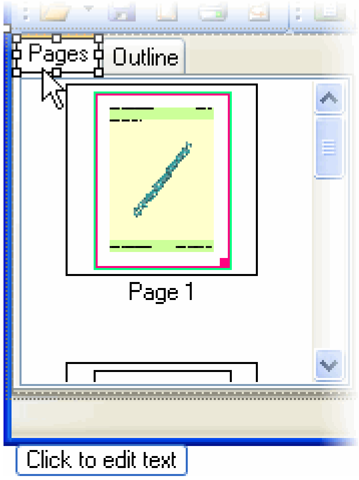
  Clicking **About** displays the control's **About** dialog box, which is helpful in finding the build number of the control.

## Smart Designers

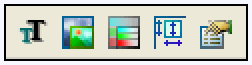
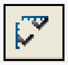
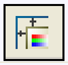
**Reports for WinForms** features a Smart Designer to enhance design-time interaction. Using the Smart Designer, you can set properties directly on the form. When the mouse is over an item in the form, a floating toolbar will appear, as well as a tab at the lower left side of the form indicating what item the mouse is over.



Items that do not have a floating toolbar associated with them will provide directions on how to customize that item directly on the form.

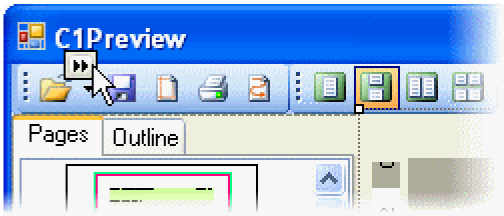


The Smart Designer consists of the following floating toolbars:

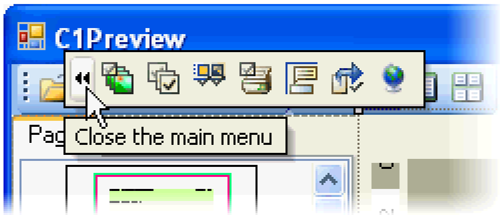
| Toolbar | Description |
|---|---|
| | **Main Menu**: The Main Menu floating toolbar allows you to select preset toolbar images, hide or show preview control panels, hide or show toolbars, set printing options, set preview pane tooltips, set export options, and localize preview. |
|  | **ToolStrip**: The ToolStrip floating toolbar allows you to edit text-related properties, images and background images, item colors, item layout, and other properties. |
| | **Thumbnails**: The Thumbnails floating toolbar allows you to control the appearance and behavior of the Thumbnail view in the Navigation panel. |
| | **Outline**: The Outline floating toolbar allows you to control the appearance and behavior of the Outline view in the Navigation panel. |
|  | **Rulers**: The Rulers floating toolbar allows you to enable or disable the horizontal and vertical rulers. |
|  | **Preview Pane Appearance**: The Preview Pane Appearance floating toolbar allows you to set the padding, colors, and border style of the preview pane. |
| | **Margins**: The Margins floating toolbar allow you to hide or show the document page margins. |
| | **Preview Pane**: The Preview Pane floating toolbar allows you to control the preview pane layout, zoom, and behavior settings. |
| | **Text Search Panel**: The Text Search Panel floating toolbar allows you to control the appearance and behavior of the Text Search Panel. |

### Main Menu Floating Toolbar

The Main Menu is the only floating toolbar that will appear on the form regardless of the item the mouse is positioned over. The main menu can be displayed by clicking the ⏩ button that appears in the upper left corner of the form.



To close the main menu, click the ◀◀ button.



The main menu consists of the following toolbar buttons:

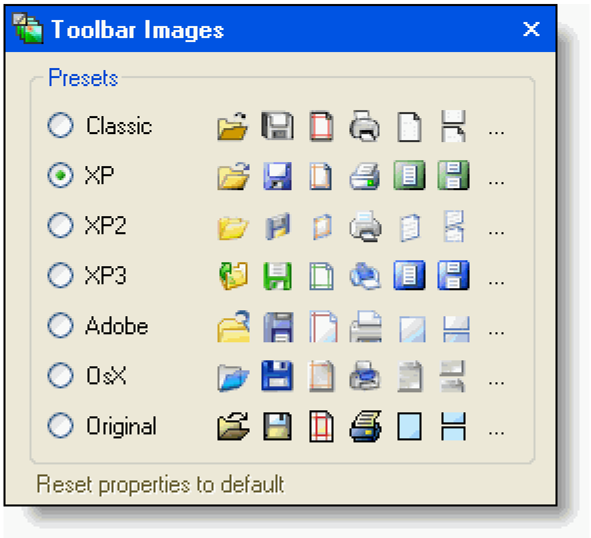
| Button | Description |
|---|---|
| | **Toolbar Images**: Select from preset toolbar images. |
| | **Layout**: Hide or show preview control panels. |
| | **Toolbar Buttons**: Hide or show toolbars. |
| | **Printing**: Set printing options. |
| | **Tooltips**: Set preview pane tooltips. |
| | **Export**: Set export options. |
| 🌐 | **Localize**: Localize preview. |

Each button will open a dialog box where you can customize the settings on the form.

**Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

### Toolbar Images

Clicking the **Toolbar Images** button opens the **Toolbar Images** dialog box where you can select from preset toolbar images. The default preset is **XP**.


Toolbar Images

Presets

○ Classic

● XP

○ XP2

○ XP3

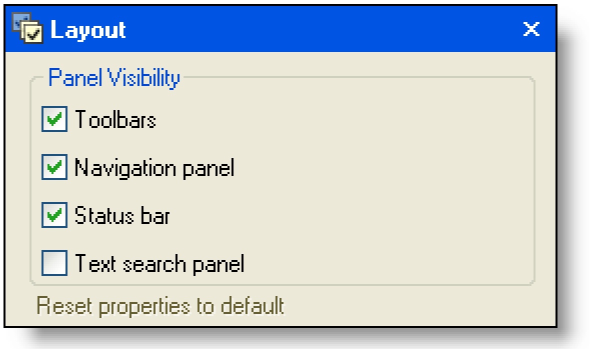
○ Adobe

○ OsX

○ Original

### Layout

Clicking the **Layout** button opens the **Layout** dialog box where you can hide or show the preview control panels. The default is checked for Toolbars, Navigation panel, and Status bar.


Layout

Panel Visibility

☑ Toolbars

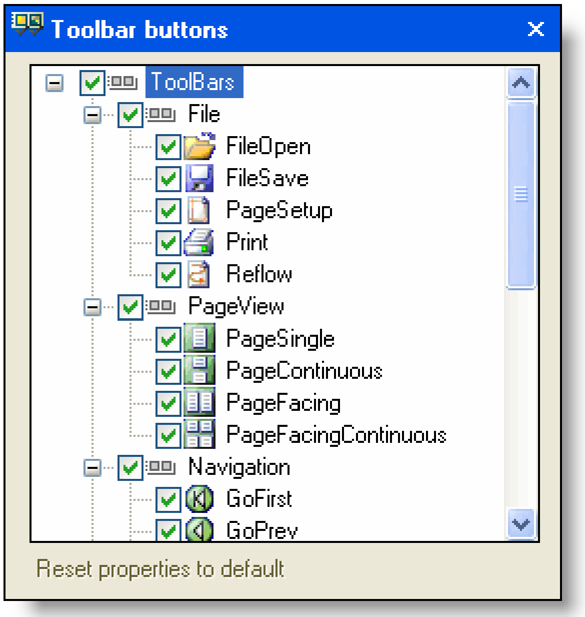
☑ Navigation panel

☑ Status bar

☐ Text search panel

### Toolbar Buttons

Clicking the **Toolbar Buttons** button opens the **Toolbar Buttons** dialog box where you can hide or show toolbars and buttons. The default is checked for all toolbars and buttons.



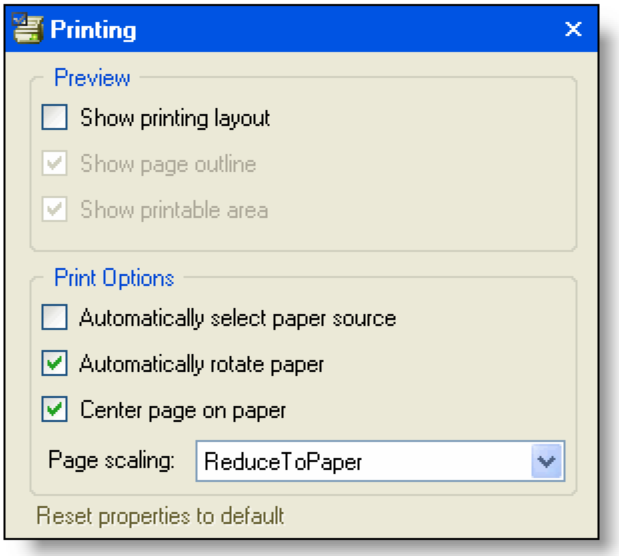
Toolbar buttons

- ☑ ToolBars
  - ☑ File
    - ☑ FileOpen
    - ☑ FileSave
    - ☑ PageSetup
    - ☑ Print
    - ☑ Reflow
  - ☑ PageView
    - ☑ PageSingle
    - ☑ PageContinuous
    - ☑ PageFacing
    - ☑ PageFacingContinuous
  - ☑ Navigation
    - ☑ GoFirst
    - ☑ GoPrev

Reset properties to default

### Printing

Clicking the **Printing** button opens the **Printing** dialog box where you can set printing options. Below are the default settings.



Printing

Preview

- □ Show printing layout
- ☑ Show page outline
- ☑ Show printable area

Print Options

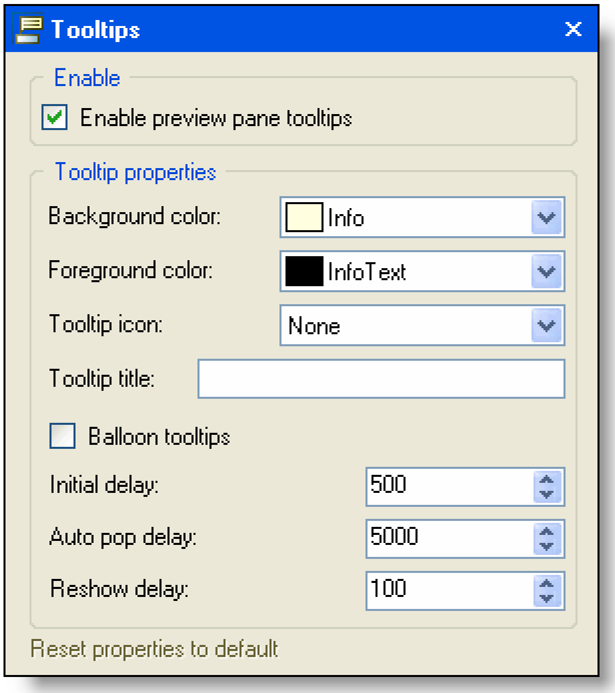
- □ Automatically select paper source
- ☑ Automatically rotate paper
- ☑ Center page on paper

Page scaling: ReduceToPaper

Reset properties to default

### Tooltips

Clicking the **Tooltips** button opens the **Tooltips** dialog box where you can customize the settings for the preview pane tooltips. Below are the default settings.



Tooltips

Enable

☑ Enable preview pane tooltips

Tooltip properties

Background color: Info

Foreground color: InfoText

Tooltip icon: None

Tooltip title:

☐ Balloon tooltips

Initial delay: 500

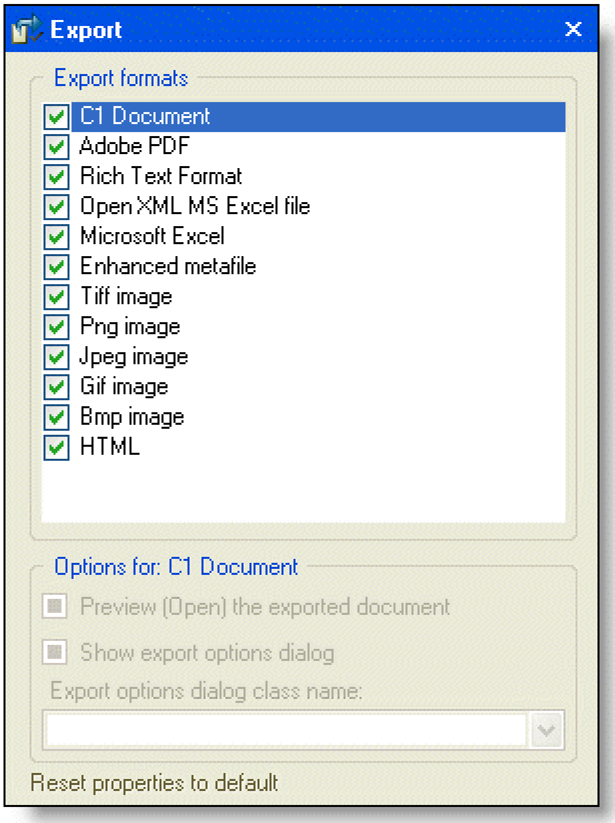
Auto pop delay: 5000

Reshow delay: 100

Reset properties to default

### Export

Clicking the **Export** button opens the **Export** dialog box where you can set export options. The default is checked for all formats.



Export

Export formats

☑ C1 Document
☑ Adobe PDF
☑ Rich Text Format
☑ Open XML MS Excel file
☑ Microsoft Excel
☑ Enhanced metafile
☑ Tiff image
☑ Png image
☑ Jpeg image
☑ Gif image
☑ Bmp image
☑ HTML

Options for: C1 Document

☐ Preview (Open) the exported document

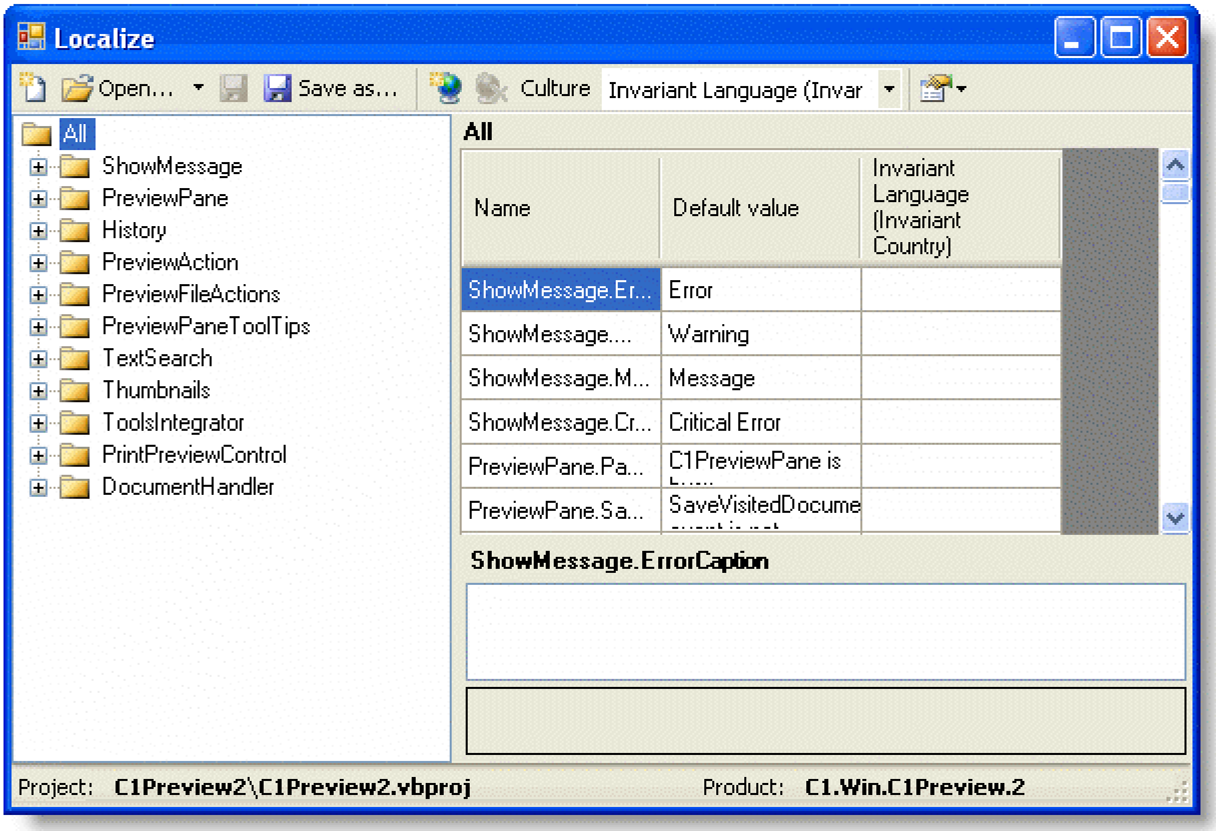
☐ Show export options dialog

Export options dialog class name:

Reset properties to default

### Localize

Clicking the **Localize** button opens the **Localize** dialog box. In the **Localize** dialog box, you can customize your localization settings.



- All
  - ShowMessage
  - PreviewPane
  - History
  - PreviewAction
  - PreviewFileActions
  - PreviewPaneToolTips
  - TextSearch
  - Thumbnails
  - ToolsIntegrator
  - PrintPreviewControl
  - DocumentHandler

**All**

| Name | Default value | Invariant Language (Invariant Country) |
|---|---|---|
| ShowMessage.Er... | Error | |
| ShowMessage.... | Warning | |
| ShowMessage.M... | Message | |
| ShowMessage.Cr... | Critical Error | |
| PreviewPane.Pa... | C1PreviewPane is ... | |
| PreviewPane.Sa... | SaveVisitedDocume... | |

**ShowMessage.ErrorCaption**

Project: **C1Preview2\C1Preview2.vbproj** Product: **C1.Win.C1Preview.2**

For more information on the **Localize** dialog box, see Localization.

## *ToolStrip Floating Toolbar*

The ToolStrip floating toolbar will appear for each item on the toolstrip and consists of the following buttons:

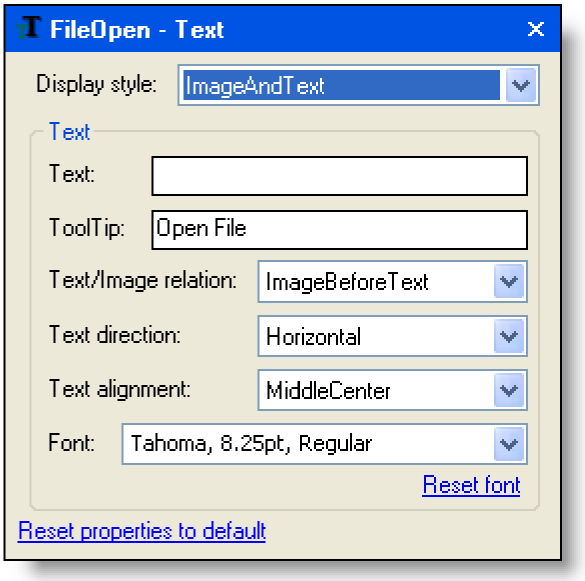
| Button | Description |
|---|---|
| | **Text**: Edit text-related properties. |
| | **Image**: Edit image and background image. |
| | **Color**: Edit item colors. |
| | **Layout**: Edit item layout. |
| | **Properties**: Edit other properties. |

Each button will open a dialog box where you can customize the settings on the form.

> **Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

### Text

Clicking the **Text** button opens the **Text** dialog box where you can edit text-related properties for that toolbar item.



FileOpen - Text

Display style: ImageAndText

Text

Text:

ToolTip: Open File

Text/Image relation: ImageBeforeText

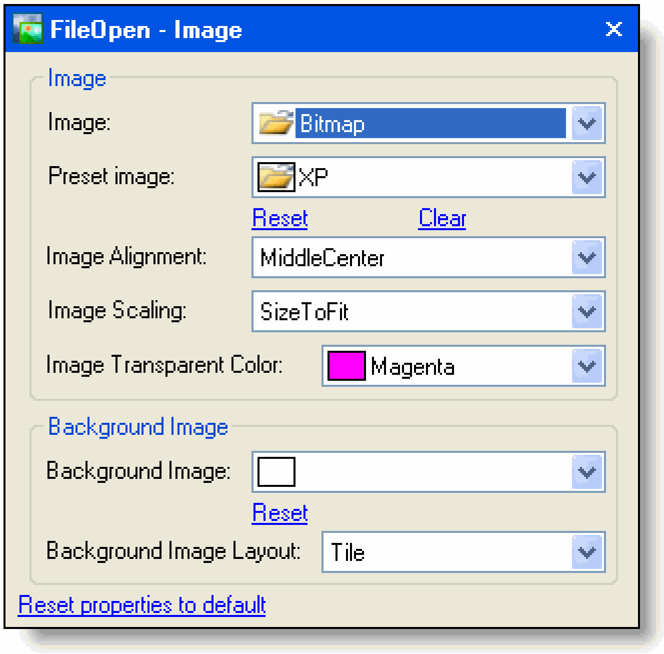
Text direction: Horizontal

Text alignment: MiddleCenter

Font: Tahoma, 8.25pt, Regular

### Image

Clicking the **Image** button opens the **Image** dialog box where you can edit the image and background image for that toolbar item.



FileOpen - Image

Image

Image: Bitmap

Preset image: XP

Image Alignment: MiddleCenter

Image Scaling: SizeToFit

Image Transparent Color: Magenta

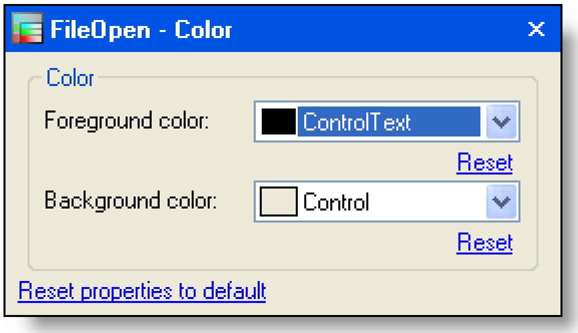
Background Image

Background Image:

Background Image Layout: Tile

### Color

Clicking the **Color** button opens the **Color** dialog box where you can edit the colors for that toolbar item.

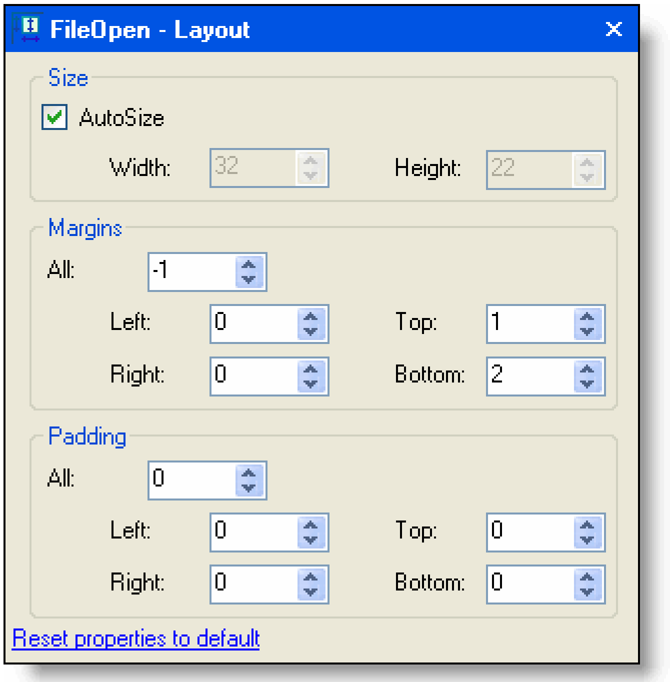


Foreground color: ControlText

Background color: Control

### Layout

Clicking the **Layout** button opens the **Layout** dialog box where you can edit the layout for that toolbar item.



☑ AutoSize

Width: 32 Height: 22

Margins

All: -1

Left: 0 Top: 1

Right: 0 Bottom: 2

Padding

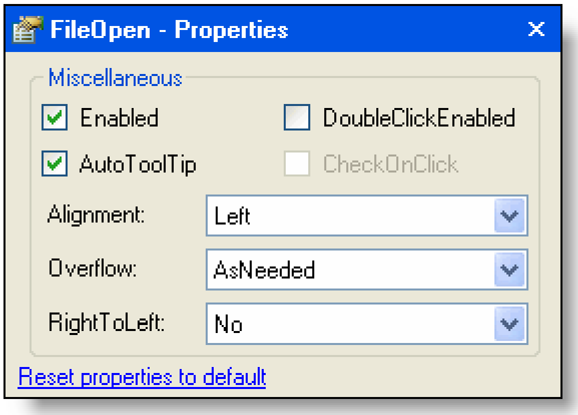
All: 0

Left: 0 Top: 0

Right: 0 Bottom: 0

### Properties

Clicking the **Properties** button opens the **Properties** dialog box where you can edit other properties for that toolbar item.



FileOpen - Properties

Miscellaneous

☑ Enabled ☐ DoubleClickEnabled

☑ AutoToolTip ☐ CheckOnClick

Alignment: Left

Overflow: AsNeeded

RightToLeft: No

Reset properties to default

### *Thumbnails Floating Toolbar*

The Thumbnails floating toolbar will appear when the mouse is over the Thumbnail view in the Navigation panel.

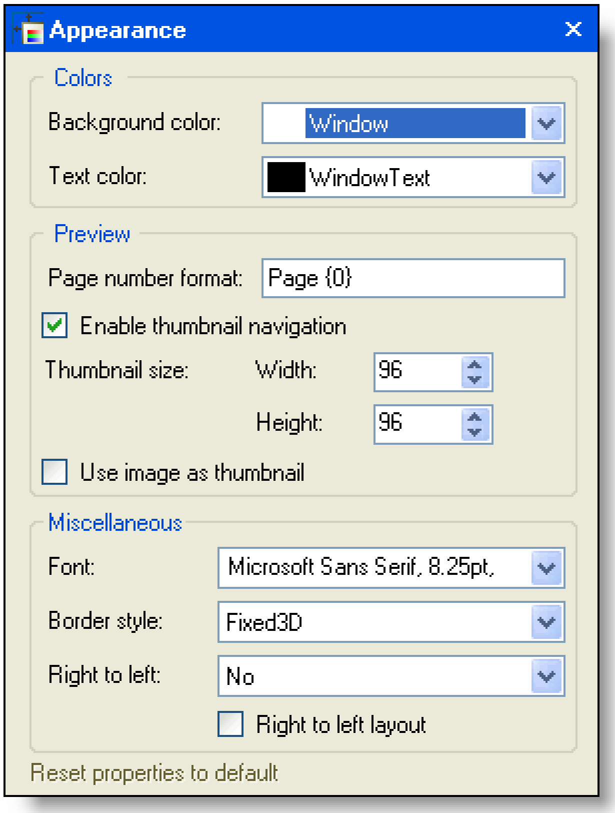
| Button | Description |
|---|---|
| | **Appearance**: Edit the Thumbnail view appearance. |
| | **Behavior**: Edit the Thumbnail view behavior. |

Each button will open a dialog box where you can customize the settings on the form.

> **Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

#### Appearance

Clicking the **Appearance** button opens the **Appearance** dialog box where you can set the colors, thumbnail settings, and font of the Thumbnail view.


Appearance

Colors

Background color: Window

Text color: WindowText

Preview

Page number format: Page {0}

☑ Enable thumbnail navigation

Thumbnail size: Width: 96

Height: 96

□ Use image as thumbnail

Miscellaneous

Font: Microsoft Sans Serif, 8.25pt,

Border style: Fixed3D

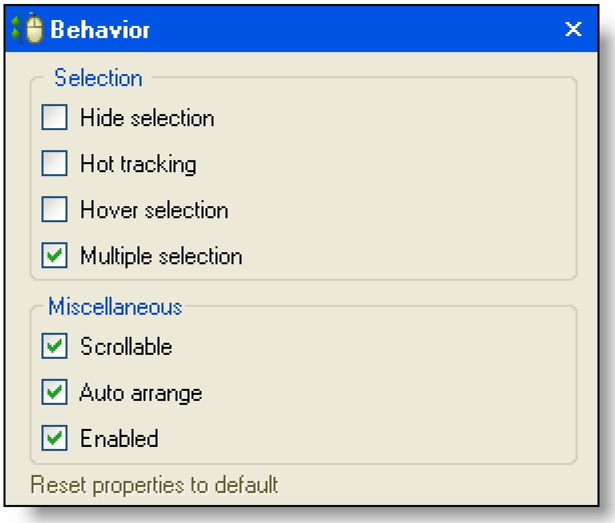
Right to left: No

□ Right to left layout

Reset properties to default

### Behavior

Clicking the **Behavior** button opens the **Behavior** dialog box where where you can edit the behavior settings of the Thumbnail view.


Behavior

Selection

□ Hide selection

□ Hot tracking

□ Hover selection

☑ Multiple selection

Miscellaneous

☑ Scrollable

☑ Auto arrange

☑ Enabled

Reset properties to default

## *Outline Floating Toolbar*

The Outline floating toolbar will appear when the mouse is over the Outline view in the Navigation panel.

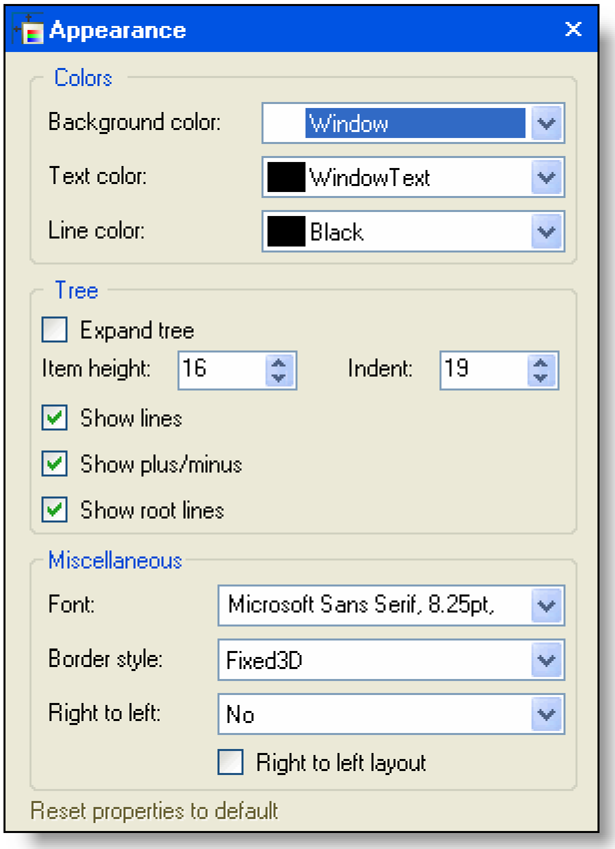
| Button | Description |
|---|---|
|  | **Appearance**: Edit the Outline view appearance. |
|  | **Behavior**: Edit the Outline view behavior. |

Each button will open a dialog box where you can customize the settings on the form.

> **Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

### Appearance

Clicking the **Appearance** button opens the **Appearance** dialog box where you can set the colors, tree settings, and font of the Outline view.



Appearance

Colors

Background color: Window

Text color: WindowText

Line color: Black

Tree

□ Expand tree

Item height: 16 Indent: 19

☑ Show lines

☑ Show plus/minus

☑ Show root lines

Miscellaneous

Font: Microsoft Sans Serif, 8.25pt,

Border style: Fixed3D

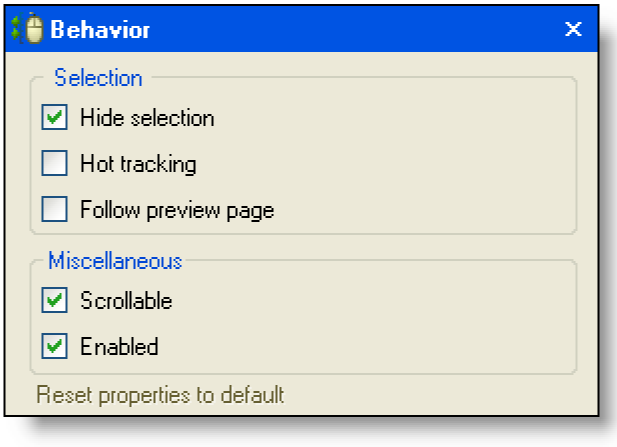
Right to left: No

□ Right to left layout

Reset properties to default

### Behavior

Clicking the **Behavior** button opens the **Behavior** dialog box where where you can edit the behavior settings of the Outline view.



Behavior

Selection

☑ Hide selection

☐ Hot tracking

☐ Follow preview page

Miscellaneous

☑ Scrollable

☑ Enabled

Reset properties to default

### Rulers Floating Toolbar

The Rulers floating toolbar will appear when the mouse is over either the horizontal or vertical ruler:

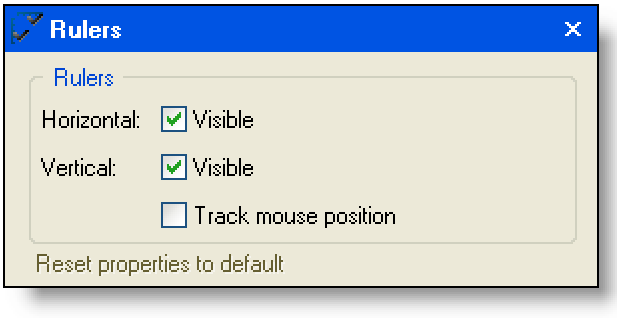
| Button | Description |
|---|---|
|  | **Rulers**: Enable or disable rulers. |

This button will open a dialog box where you can customize the settings on the form.

> **Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

#### Rulers

Clicking the **Rulers** button opens the **Rulers** dialog box where you can enable or disable the rulers and set the mouse tracking.



Rulers

Rulers

Horizontal: ☑ Visible

Vertical: ☑ Visible

☐ Track mouse position

Reset properties to default

> **Note:** At least one ruler must be present to use the **Rulers** dialog box. If no rulers are visible check the **Horizontal** and **Vertical** check boxes under **Rulers** on the **Behavior** dialog box in the preview pane menu.

### Preview Pane Appearance Floating Toolbar

The Preview Pane Appearance floating toolbar will appear when the mouse is over either the horizontal or vertical page padding:

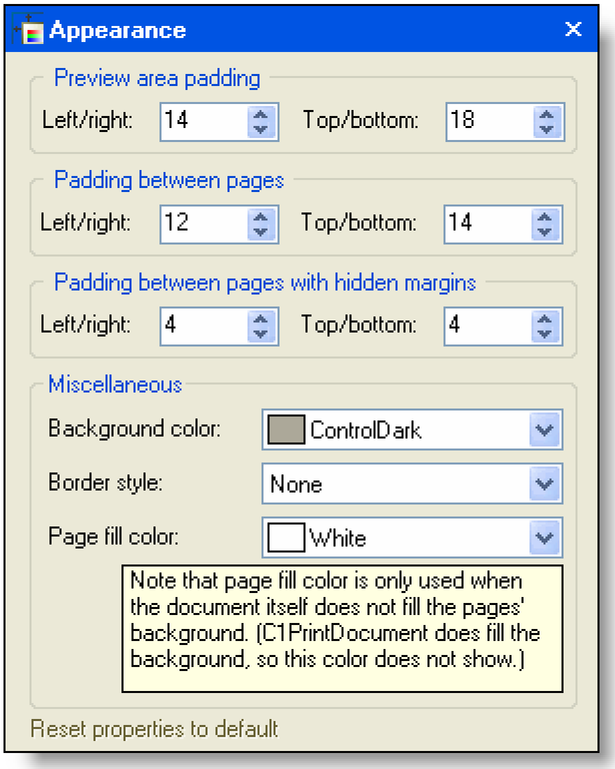
| Button | Description |
|---|---|
|  | **Appearance**: Edit the preview pane appearance. |

This button will open a dialog box where you can customize the settings on the form.

> **Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

### Appearance

Clicking the **Appearance** button opens the **Appearance** dialog box where you can set the padding, colors, and border style of the preview pane.



Appearance

Preview area padding

Left/right: 14 Top/bottom: 18

Padding between pages

Left/right: 12 Top/bottom: 14

Padding between pages with hidden margins

Left/right: 4 Top/bottom: 4

Miscellaneous

Background color: ControlDark

Border style: None

Page fill color: White

Note that page fill color is only used when the document itself does not fill the pages' background. (C1PrintDocument does fill the background, so this color does not show.)

Reset properties to default

## *Margins Floating Toolbar*

The Margins floating toolbar will appear when the mouse is over either the horizontal or vertical margins:

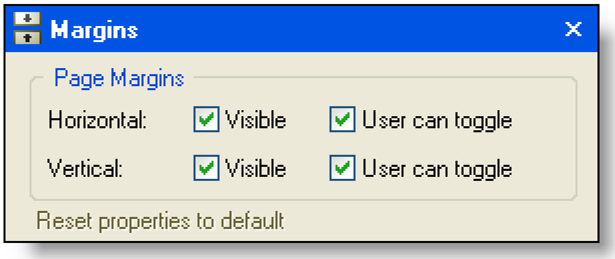
| Button | Description |
|---|---|
|  | **Margins**: Hide or show document page margins. |

This button will open a dialog box where you can customize the settings on the form.

> **Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

### Margins

Clicking the **Margins** button opens the **Margins** dialog box where you can set the visibility of the document page margins.



Margins

Page Margins

Horizontal: ☑ Visible ☑ User can toggle

Vertical: ☑ Visible ☑ User can toggle

Reset properties to default

## *Preview Pane Floating Toolbar*

The Preview Pane floating toolbar will appear when the mouse is over the page body:

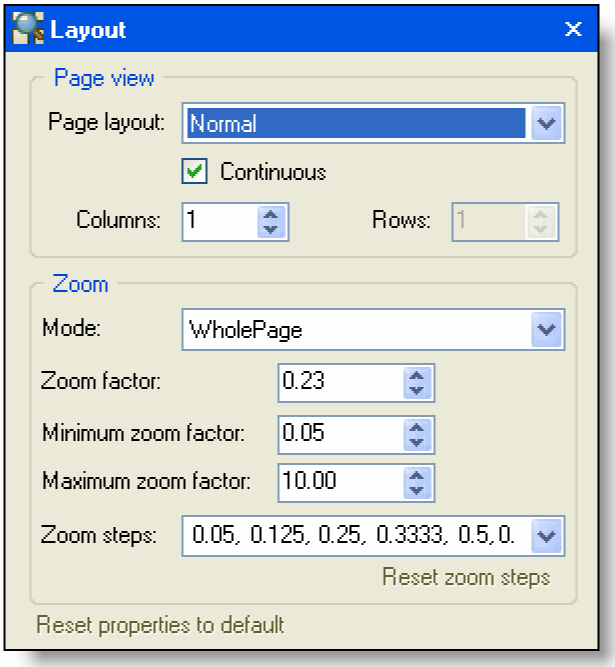
| Button | Description |
|---|---|
| 🔍 | **Layout**: Edit preview pane layout and zoom settings. |
| 🖱 | **Behavior**: Edit preview pane behavior. |

Each button will open a dialog box where you can customize the settings on the form.

> **Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

### Layout

Clicking the **Layout** button opens the **Layout** dialog box where you can edit preview pane layout and zoom settings.



Layout

Page view

Page layout: Normal

☑ Continuous

Columns: 1 Rows: 1

Zoom

Mode: WholePage

Zoom factor: 0.23

Minimum zoom factor: 0.05

Maximum zoom factor: 10.00

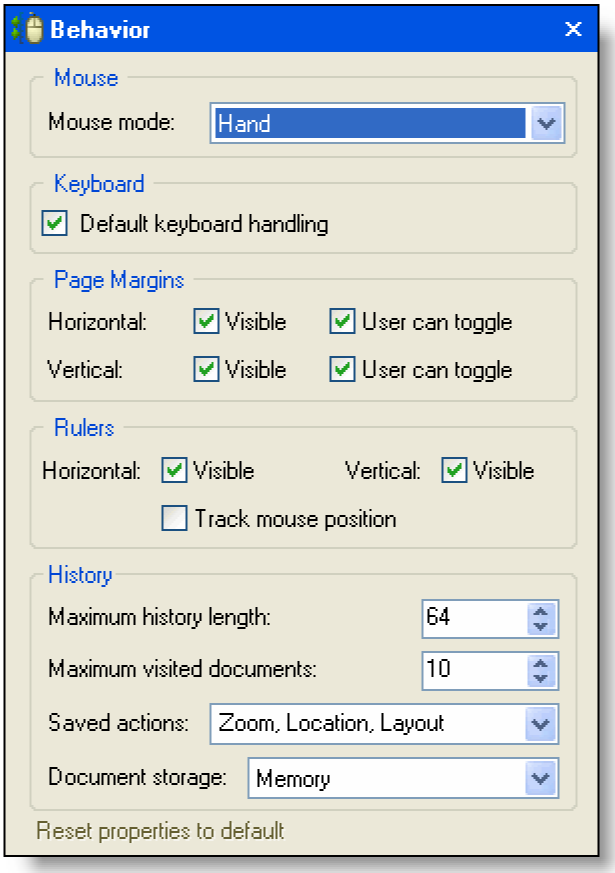
Zoom steps: 0.05, 0.125, 0.25, 0.3333, 0.5, 0.

Reset zoom steps

Reset properties to default

### Behavior

Clicking the **Behavior** button opens the **Behavior** dialog box where you can edit preview pane behavior settings.



Behavior

Mouse

Mouse mode: Hand

Keyboard

☑ Default keyboard handling

Page Margins

Horizontal: ☑ Visible ☑ User can toggle

Vertical: ☑ Visible ☑ User can toggle

Rulers

Horizontal: ☑ Visible Vertical: ☑ Visible

☐ Track mouse position

History

Maximum history length: 64

Maximum visited documents: 10

Saved actions: Zoom, Location, Layout

Document storage: Memory

Reset properties to default

### *Text Seach Panel Floating Toolbar*

The Text Search Panel floating toolbar will appear when the mouse is over the Text Search panel.

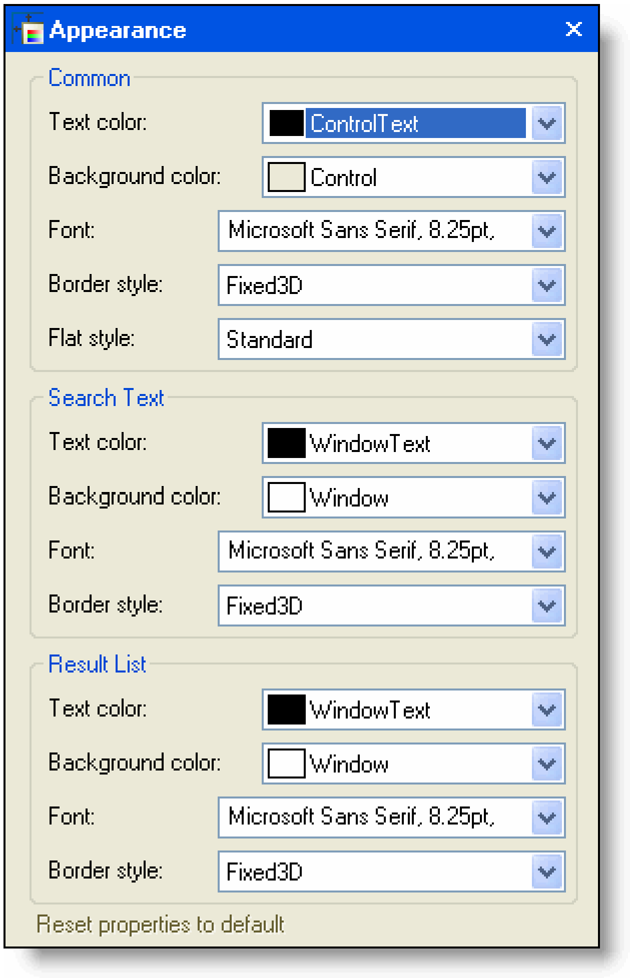
| Button | Description |
|---|---|
|  | **Appearance**: Edit the Text Search panel appearance. |
|  | **Behavior**: Edit the Text Search panel behavior. |

Each button will open a dialog box where you can customize the settings on the form.

> **Note:** If you change the settings in the dialog box the **Reset properties to default** link will appear. Clicking this link will reset all of the properties to the default setting.

#### Appearance

Clicking the **Appearance** button opens the **Appearance** dialog box where you can set the colors, tree settings, and font of the Text Search panel.



Common

Text color: ControlText

Background color: Control

Font: Microsoft Sans Serif, 8.25pt,

Border style: Fixed3D

Flat style: Standard

Search Text

Text color: WindowText

Background color: Window

Font: Microsoft Sans Serif, 8.25pt,

Border style: Fixed3D

Result List

Text color: WindowText

Background color: Window

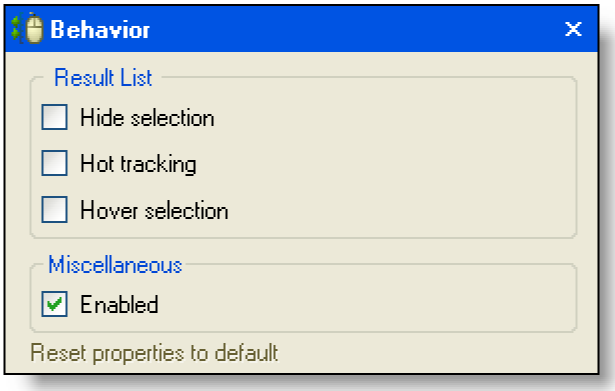
Font: Microsoft Sans Serif, 8.25pt,

Border style: Fixed3D

Reset properties to default

### Behavior

Clicking the **Behavior** button opens the **Behavior** dialog box where you can edit the behavior settings of the Text Search panel.



□ Hide selection

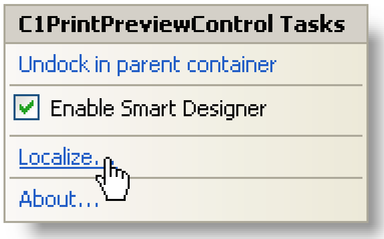
□ Hot tracking

□ Hover selection

☑ Enabled

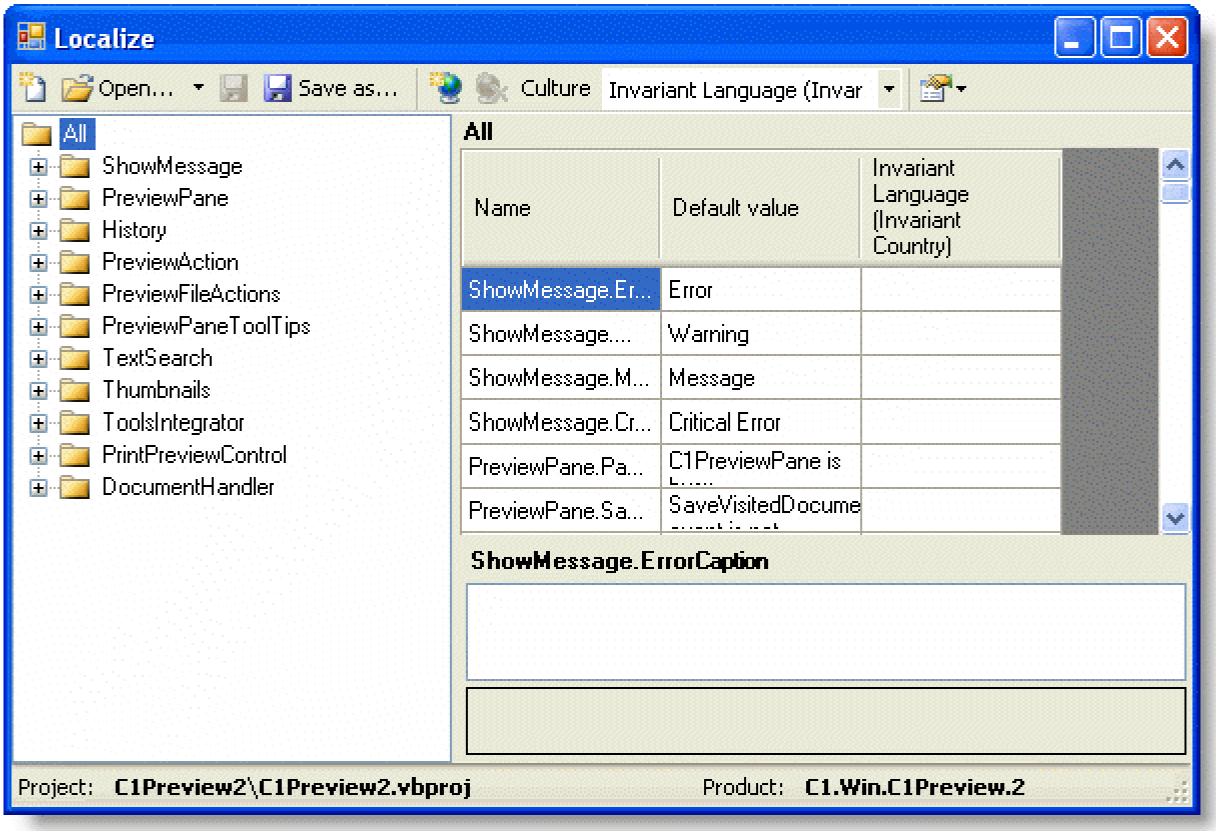
## Localization

To localize **Reports for WinForms** components in your application, select **Localize** in the C1PrintPreviewControl Tasks Menu, in the Main Menu Floating Toolbar, or from the context menu of a **Reports for WinForms** component.



☑ Enable Smart Designer

Clicking **Localize** opens the **Localize** dialog box:



Culture: Invariant Language (Invar

All

| Name | Default value | Invariant Language (Invariant Country) |
|---|---|---|
| ShowMessage.Er... | Error | |
| ShowMessage.... | Warning | |
| ShowMessage.M... | Message | |
| ShowMessage.Cr... | Critical Error | |
| PreviewPane.Pa... | C1PreviewPane is | |
| PreviewPane.Sa... | SaveVisitedDocume | |

ShowMessage.ErrorCaption

Project: C1Preview2\C1Preview2.vbproj Product: C1.Win.C1Preview.2

The **Localize** dialog box allows you to localize any of the **Reports for WinForms** assemblies (C1.C1Report.2.dll or C1.Win.C1Report.2.dll) and save the localized resources in any project of your solution.

On the left of the **Localize** dialog box, there is a tree listing the localizable strings' IDs, and on the right are the strings themselves. The structure of the tree reflects the hierarchy of sub-classes in the Strings class. The right panel can show either all strings, or just the strings belonging to the selected tree node.

The strings' list contains the following columns:

| Column | Description |
|---|---|
| **Name** | The string's name (ID); this column repeats the selected tree node, and can be optionally turned off. |
| **Default value** | The default (English) value of the string. |
| **Value** | The string value for the currently selected culture (the column header displays that culture). |

Below the list, the currently selected string's value is shown, and an optional description.

Status bar displays the project which will contain the localized resources, and the name of the C1 assembly which is currently being localized.

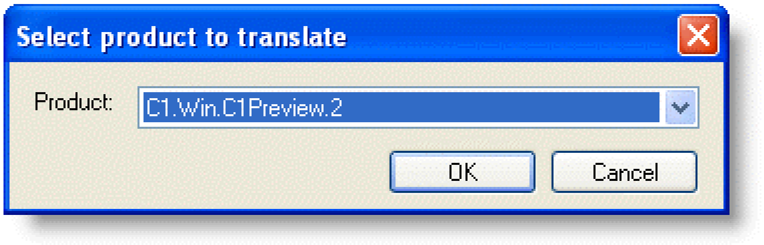
## Localization Toolbar

The **Localize** dialog box contains the following toolbar menu buttons:

| Button | Description |
|---|---|
| 🗋 | **Create new translation** begins a new localization for a ComponentOne assembly. |
| 📂 Open... ▾ | **Open** opens an existing translation for a particular assembly. |
| 💾 | **Save** saves the current translation. |
| 💾 Save as... | **Save as** saves the current translation and allows you to select the project in which to save the translation. |
| 🌐 | **Add culture** adds a new culture. |
| 🌐✗ | **Delete culture** removes a culture from the translations. |
| Culture Invariant Language (Invar ▾ | **Select culture** selects the culture to display and edit. |
| 📋▾ | **Options** customizes the appearance and behavior of the localization window. |

## Create New Translation

Clicking the **Create new translation** button begins a new localization for a ComponentOne assembly. A dialog box opens for you to select the ComponentOne assembly to localize.


Select product to translate

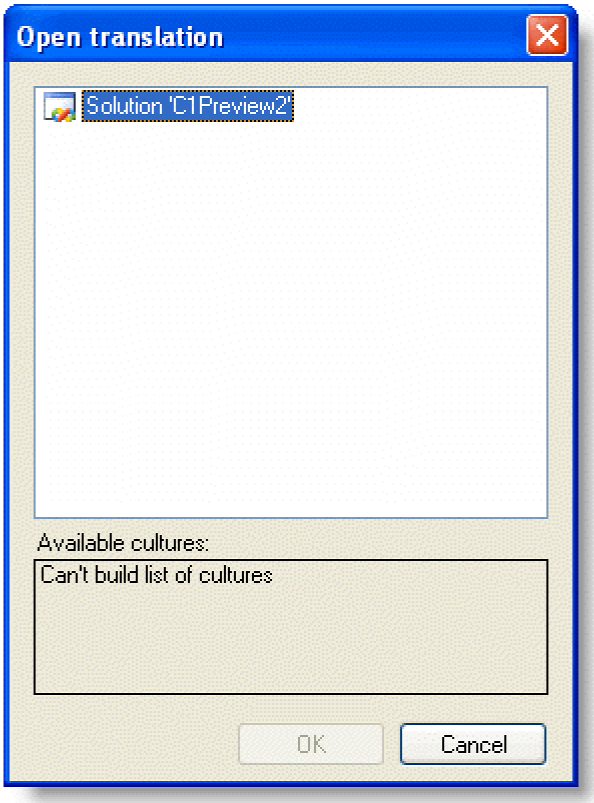
Product: C1.Win.C1Preview.2

**Note:** The assembly must be referenced in the currently open solution.

## Open

Clicking the **Open** button opens an existing translation for a particular assembly. All translations that you create are stored as .resx files, and are automatically added to the project that you select while saving the translation. Clicking this item shows a dialog box where a previously saved translation can be selected.

When you create a new solution it does not contain any translations, so initially that window would look similar to the following image:



After you have created and saved a translation, the **Available cultures** panel shows the list of cultures for which translations were created for the selected assembly.

### Save

Clicking the **Save** button saves the current translation.

The translation is saved in the project shown in the status bar. When the translation is saved, a folder with the name **C1LocalizedResources** is created in the selected project (if it does not already exist), and the .resx files with translations are saved in that folder, and added to the project. For each culture, a separate .resx file is created. These files are visible in the Solution Explorer window.

**Note:** If your translation is only for the invariant culture, the .resx file does not contain a culture suffix.

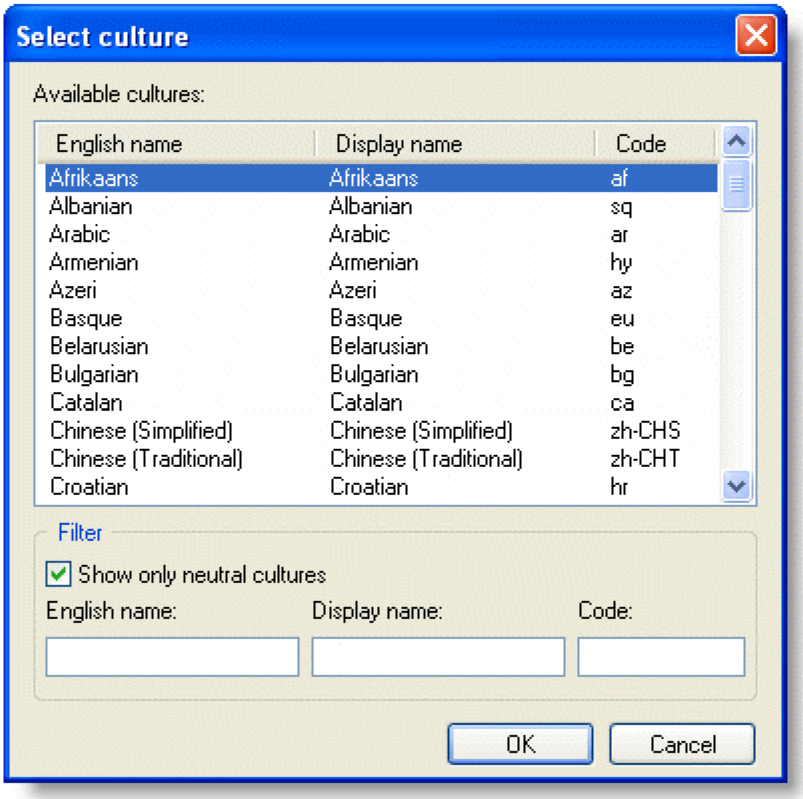
### Save As

Clicking the **Save as** button saves the current translation and allows you to select the project in which to save the translation.

### Add Culture

Clicking the **Add culture** button adds a new culture.

You can make translations for several cultures, and dynamically switch between them at run time. For each culture, a separate .resx file is created in the **C1LocalizedResources** folder. Clicking the **Add culture** button opens the **Select culture** dialog box that provides a list of available cultures:



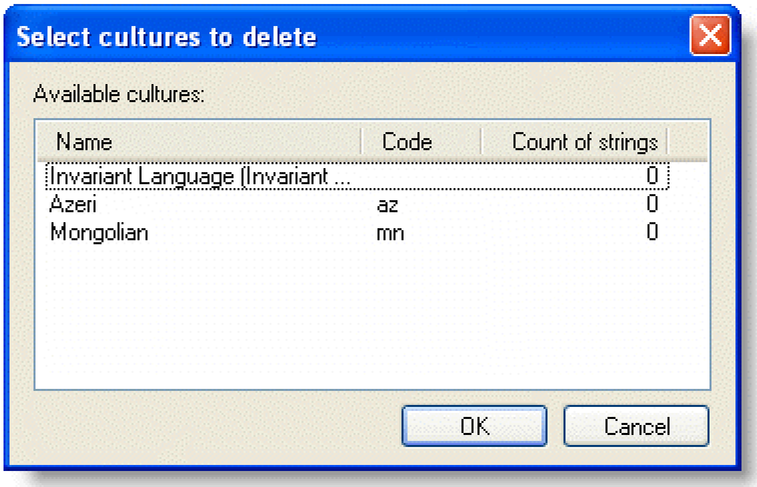
| English name | Display name | Code |
|---|---|---|
| Afrikaans | Afrikaans | af |
| Albanian | Albanian | sq |
| Arabic | Arabic | ar |
| Armenian | Armenian | hy |
| Azeri | Azeri | az |
| Basque | Basque | eu |
| Belarusian | Belarusian | be |
| Bulgarian | Bulgarian | bg |
| Catalan | Catalan | ca |
| Chinese (Simplified) | Chinese (Simplified) | zh-CHS |
| Chinese (Traditional) | Chinese (Traditional) | zh-CHT |
| Croatian | Croatian | hr |

☑ Show only neutral cultures

Initially the list contains neutral cultures only. To show all cultures, uncheck the **Show only neutral cultures** checkbox. You can use the **English name**, **Display name**, and **Code** boxes to filter the list of shown cultures. After you have selected a culture, press the **OK** button to add it to the translations. The newly added culture will appear in the **Culture** drop-down in the toolbar, and will become current in the window.

### Delete Culture

Clicking the **Delete culture** button removes a culture from the translations. The **Select cultures to delete** dialog box provides the list of cultures existing in the translations:



| Name | Code | Count of strings |
|---|---|---|
| Invariant Language (Invariant ... | | 0 |
| Azeri | az | 0 |
| Mongolian | mn | 0 |

Selecting a culture and clicking **OK** removes it from the translations.

### Select Culture

The **Culture** drop-down allows you to select the culture to display and edit.

### Options

Clicking the **Options** button allows to customize the appearance and behavior of the localization window. The available Localization options include:

| Option | Description |
|---|---|
| **Sync tree with grid** | When this item is checked, selecting a string in the right panel list also selects that string in the tree on the left. By default this item is unchecked. |
| **Show names in grid** | When this item is checked, the *Name* column is show in the right hand panel, otherwise that column is hidden. By default this item is checked. |
| **Show selected group only** | When this item is checked, the list of strings on the right contains only the strings from the group currently selected in the tree on the left. By default this item is unchecked. |

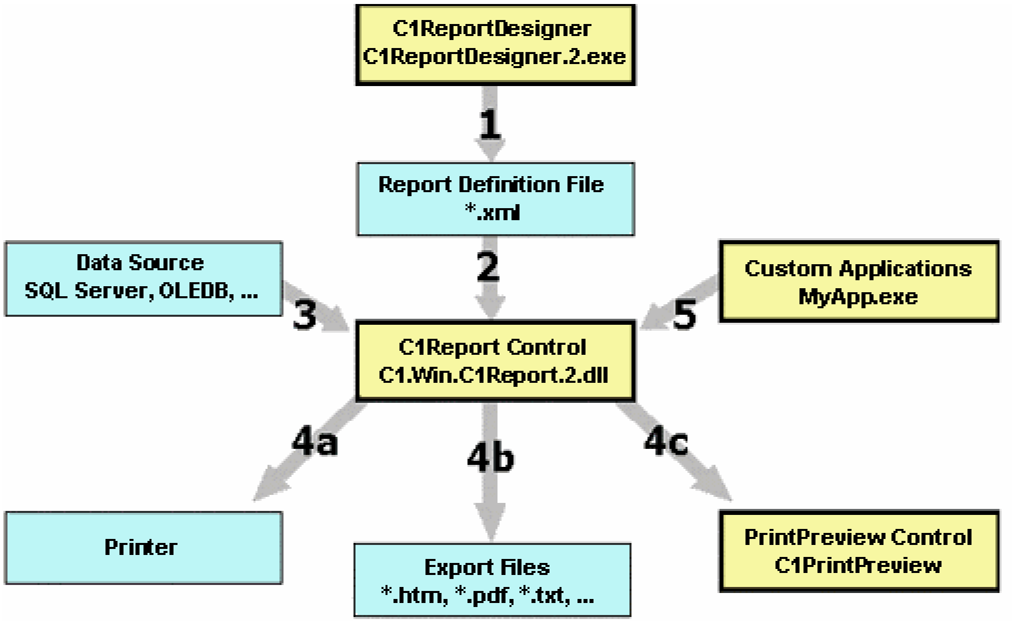
# Working with C1Report

You can use C1Report in many different scenarios, on the desktop and on the Web. The main sequence of steps is always the same:

1. You start by creating a report using the **C1ReportDesigner** application to create report definitions; report definitions are saved in XML files, and can be designed from scratch or imported from existing Microsoft Access reports. You can then modify the basic report using **C1ReportDesigner**.
2. The C1Report component reads the report definitions and renders the reports using data from any standard .NET data source.
3. The report definitions can be loaded at design time, and embedded in your application, or they can be read and modified at run time. (You can even create report definitions from scratch, using the C1Report object model.)
4. Reports can be rendered directly to a printer, into a **C1PrintPreview** control, or into HTML and PDF files that can be published on the Web.

The following diagram shows the relationship between the components in the **ComponentOne Reports for WinForms** package:

> **Note:** Boxes with a bold border represent code components (controls and applications). Boxes with a thin border represent files containing information (report definitions, data, and finished reports).



The following numbers refer to the numbered arrows in the image, indicating relationships between the components:

1. Use the **C1ReportDesigner** application to create, edit, and save XML report definition files.
2. The C1Report component loads report definitions from the XML files created with the Designer. This can be done at design time (in this case the XML file is persisted with the control and not needed at run time) or at run time using the Load method.

3. The C1Report component loads data from the data source specified in the report definition file. Alternatively, you can provide your own custom data source.
4. The C1Report component formats the data according to the report definition and renders reports to a (a) printer, (b) to one of several file formats, or (c) to a print preview control.
5. Custom applications can communicate with the C1Report component using a rich object model, so you can easily customize your reports or generate entirely new ones. **C1ReportDesigner** is a good example of such an application.

## Object Model Summary

The object model for the C1Report component is largely based on the Microsoft Access model, except that Access has different types of controls (label control, textbox control, line control, and so on), while C1Report has a single Field object with properties that can make it look like a label, textbox, line, picture, subreport, and so on.

The following table lists all objects, along with their main properties and methods (note that C1Report uses *twips* for all measurements. One *twip* is 1/20 point, so 72 points = 1440 *twips* = 1 inch):

**C1Report Object:** the main component

ReportName, GetReportInfo, Load, Save, Clear, Render, RenderToFile, RenderToStream, PageImages, Document, DoEvents, IsBusy, Cancel, Page, MaxPages, Font, OnOpen, OnClose, OnNoData, OnPage, OnError, Evaluate, Execute

**Layout Object:** determines how the report will be rendered on the page

Width, MarginLeft, MarginTop, MarginRight, MarginBottom, PaperSize, Orientation, Columns, ColumnLayout, PageHeader, PageFooter, Picture, PictureAlign, PictureShow

**DataSource Object:** manages the data source

ConnectionString, RecordSource, Filter, MaxRecords, Recordset

**Groups Collection:** a report may have many groups

**Group Object:** controls data sorting and grouping

Name, GroupBy, Sort, KeepTogether, SectionHeader, SectionFooter, Move

**Sections Collection:** all reports have at least 5 sections

**Section Object:** contains Field objects (also known as "report band")

Name, Type, Visible, BackColor, OnFormat, OnPrint, Height, CanGrow, CanShrink, Repeat, KeepTogether, ForcePageBreak

**Fields Collection:** a report usually has many Fields

**Field Object:** a rectangular area within a section where information is displayed

Name, Section, Text, TextDirection, Calculated, Value, Format, Align, WordWrap, Visible, Left, Top, Width, Height, CanGrow, CanShrink, Font, BackColor, ForeColor, BorderColor, BorderStyle, LineSlant, LineWidth, MarginLeft, MarginRight, MarginTop, MarginBottom, LineSpacing, ForcePageBreak, HideDuplicates, RunningSum, Picture, PictureAlign, Subreport, CheckBox, RTF

## Sections of a Report

Every report has at least the following five sections:

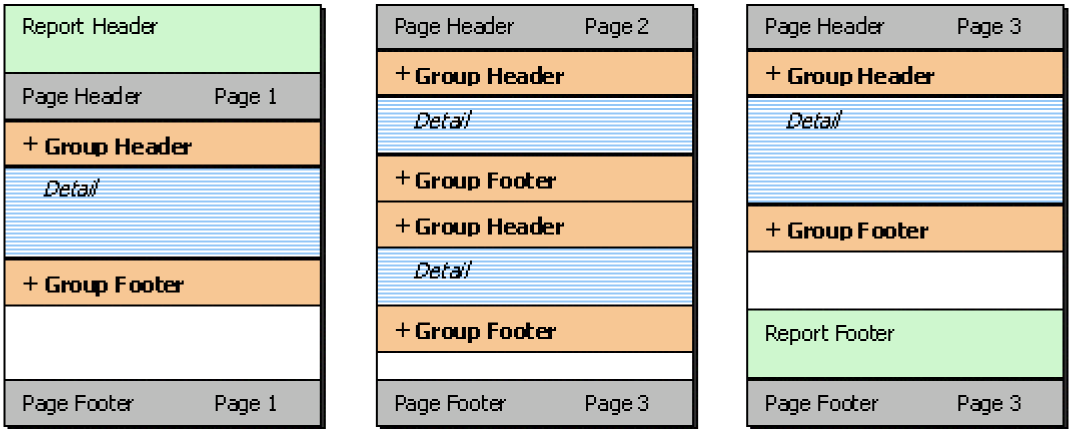
| Section | Description |
|---|---|

| | |
|---|---|
| Detail | The Detail section contains fields that are rendered once for each record in the source recordset. |
| Header | The Report Header section is rendered at the beginning of the report. |
| Footer | The Report Footer section is rendered at the end of the report. |
| Page Header | The Page Header section is rendered at the top of every page (except optionally for pages that contain the Report Header). |
| Page Footer | The Page Footer section is rendered at the bottom of every page (except optionally for pages that contain the Report Footer). |

In addition to these five sections, there are two additional sections for each group: a Group Header and a Group Footer Section. For example, a report with 3 grouping levels will have 11 sections.

Note that sections can be made invisible, but they cannot be added or removed, except by adding or removing groups.

The following diagram shows how each section is rendered on a typical report:



### Report Header

The first section rendered is the Report Header. This section usually contains information that identifies the report.

### Page Header

After the Report Header comes the Page Header. If the report has no groups, this section usually contains labels that describe the fields in the Detail Section.

### Group Headers and Group Footers

The next sections are the Group Headers, Detail, and Group Footers. These are the sections that contain the actual report data. Group Headers and Footers often contain aggregate functions such as group totals, percentages, maximum and minimum values, and so on. Group Headers and Footers are inserted whenever the value of the expression specified by the GroupBy property changes from one record to the next.

### Detail

The Detail section contains data for each record. It is possible to hide this section by setting its Visible property to **False**, and display only Group Headers and Footers. This is a good way to create summary reports.

### Page Footer

At the bottom of each page is the Page Footer Section. This section usually contains information such as the page number, total number of pages in the report, and/or the date the report was printed.

### Report Footer

Finally, the Report Footer section is printed before the last page footer. This section is often used to display summary information about the entire report.

### Customized sections

You can determine whether or not a section is visible by setting its Visible property to **True** or **False**. Group Headers can be repeated at the top of every page (whether or not it is the beginning of a group) by setting their Repeat property to **True**. Page Headers and Footers can be removed from pages that contain the Report Header and Footer sections by setting the PageHeader and PageFooter properties on the Layout object.

# Developing Reports for Desktop Scenarios

In typical desktop scenarios, C1Report runs on the same computer where the reports will be generated and viewed (the report data itself may still come from a remote server). The following scenarios assume that C1Report will be hosted in a Visual Studio.NET environment.

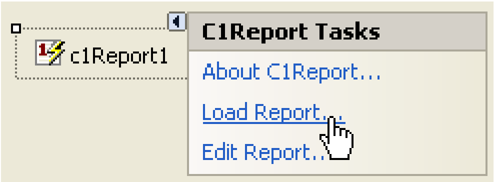
## *Embedded Reports (Loaded at Design Time)*

Under this scenario, an application generates reports using a fixed set of report definitions that are built into the application. This type of application does not rely on any external report definition files, and end-users have no way to modify the reports.

The main advantage of this type of application is that you don't need to distribute the report definition file, and you can be sure that no one will modify the report format. The disadvantage is that to make any modifications to the report, you must recompile the application.

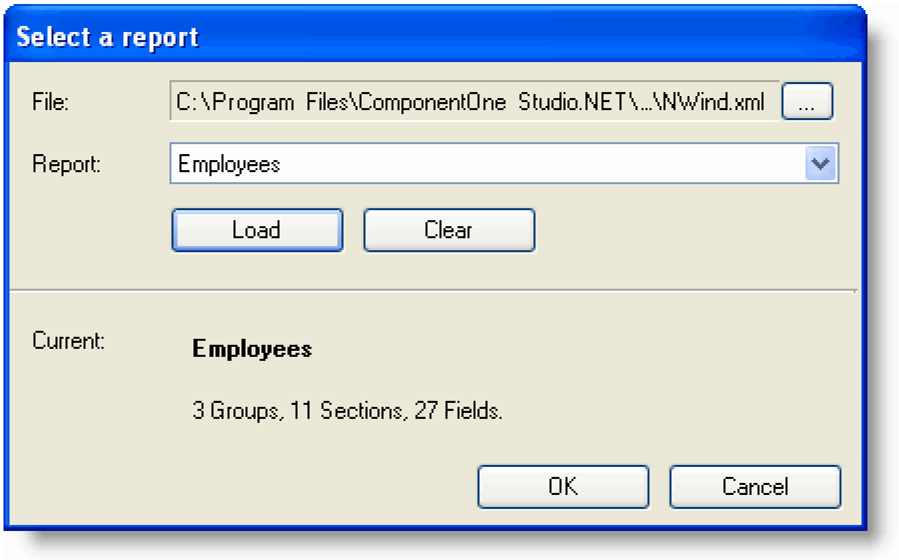
If you want to use a report definition that you already have, without any modifications, follow these steps (we will later describe how you can edit an embedded report or create one from scratch):

1. Add one C1Report component for each report definition you want to distribute. You may want to name each control after the report it will render (this will make your code easier to maintain).
2. Right-click each C1Report component and select the **Load Report** menu option to load report definitions into each control. (You can also click the smart tag (▸) above the component to open the **C1Report Tasks** menu.)



The **Select a report** dialog box appears, which allows you to select a report definition file and then a report within that file.

To load a report, click the **ellipsis** button to select the report definition file you created in step 1, then select the report from the drop-down list and click the **Load** button. The property page shows the name of the report you selected and a count of groups, sections, and fields. This is what the dialog box looks like:



File: C:\Program Files\ComponentOne Studio.NET\...\NWind.xml

Report: Employees

Current: **Employees**

3 Groups, 11 Sections, 27 Fields.

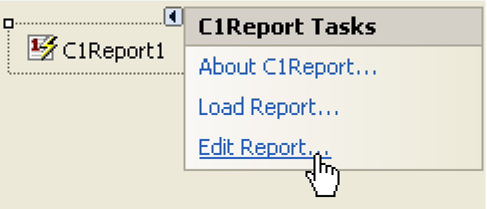
3. Add a single **C1PrintPreview** control to the form (or a **Microsoft PrintPreviewControl**) and also add a control that will allow the user to pick a report (this could be a menu, a list box, or a group of buttons).
4. Add code to render the report selected by the user. For example, if you added a button in the previous step with the name **btnProductsReport**, the code would look like this:
   - Visual Basic
     ```
     Private Sub btnProductsReport_Click(ByVal sender As System.Object,
     ByVal e As System.EventArgs) Handles btnProductsReport.Click
       ppv.Document = rptProducts.Document
     End Sub
     ```
   - C#
     ```
     private void btnProductsReport_Click(object sender, System.EventArgs e)
     {
       ppv.Document = rptProducts.Document;
     }
     ```

   Note that **rptProducts** is the name of the C1Report component that contains the report selected by the user and **ppv** is the name of the **C1PrintPreview** control.

### *Embedded Reports (Created at Design Time)*

The **Load Report** command, described in Embedded Reports (Loaded at Design Time), makes it easy to embed reports you already have into your application. In some cases, however, you may want to customize the report, or use data source objects that are defined in your Visual Studio application rather than use connection strings and record sources. In these situations, use the **Edit Report** command instead.

To create or edit reports at design time, right-click the C1Report component and select the **Edit Report** menu option to invoke the **C1ReportDesigner** application (you can also click the smart tag (▸) above the component to open the **C1Report Tasks** menu).



> **Note:** If the **Edit Report** command doesn't appear on the context menu and Properties window, it is probably because the control could not find the **C1ReportDesigner** application. To fix this, simply run the **C1ReportDesigner** application once in stand-alone mode. The designer will save its location to the registry, and the C1Report component should be able to find it afterwards.

The **C1ReportDesigner** application will show the report currently loaded in the C1Report component. If the C1Report component is empty, the Designer will show the **C1Report Wizard** so you can create a new report.

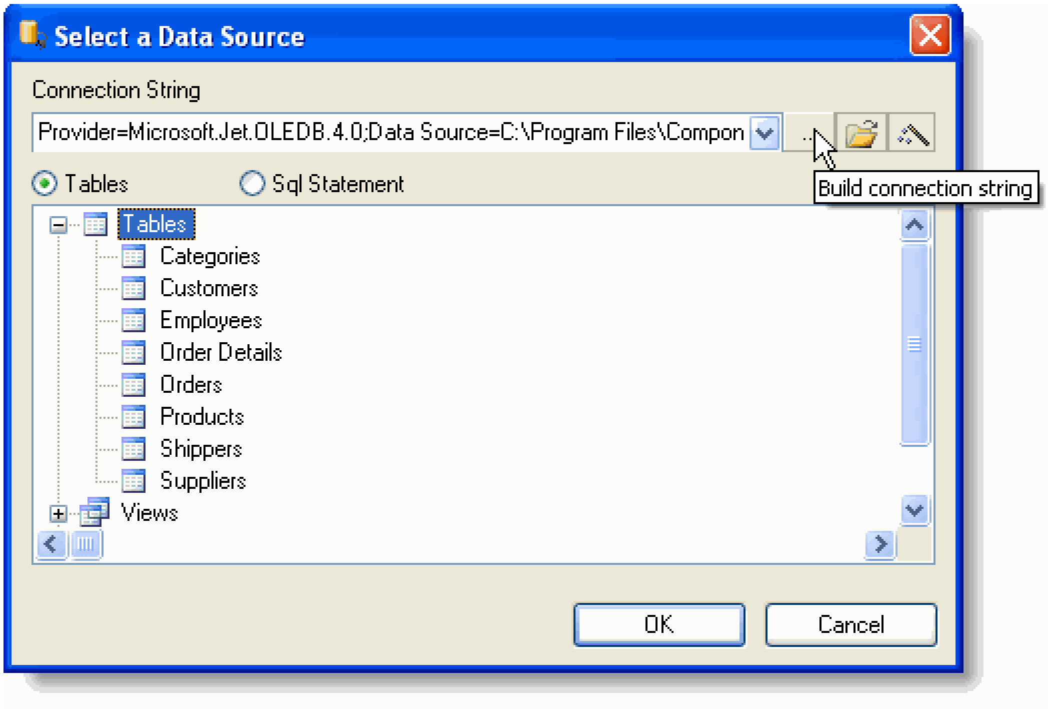
This is the same **C1ReportDesigner** application used in stand-alone mode. The only differences you will notice when you start the **C1ReportDesigner** application in this mode are:

- You can use the data source objects defined in your application as data sources for your new reports.
- When you close the Designer, any changes you made will be saved back into the C1Report component on the form (unless you choose to discard the changes by selecting **File | Exit** from the Designer's menu, and select **No** to saving the changes).

To use data source objects defined in your application, click the **Data Source** button in the Designer, then select the **Tables** option in the **Select a Data Source** dialog box.

The **Tables** page shows a list of data objects currently defined on the form (the page will not be visible if there aren't any valid data sources on the form). Alternatively, you can use the **Build connection string** button to build and select a connection string and record source as usual.

For example, if the main form contains a DataSet object with several DataTables attached to it, the data source picker dialog box might look like this:



Select a Data Source

Connection String: Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\Compon

● Tables ○ Sql Statement

- Tables
  - Categories
  - Customers
  - Employees
  - Order Details
  - Orders
  - Products
  - Shippers
  - Suppliers
- Views

Once you are done creating or editing the report, you can close the Designer by selecting **File | Save** and **File | Exit** from the menu. This will save the report definition directly into the component (as if you had loaded it from a file using the **Load Report** command).

If you change your mind and decide to cancel the changes, quit the Designer selecting **File | Exit** from the menu and choose **No** to saving the changes.

### *Reports Loaded at Run Time*

Loading reports at run time requires a report definition file and works like a viewer. The main advantage of this type of application is that if you modify the report format, there's no need to update the application. Simply send the new report definition file to the users and you are done.

To create an application with reports loaded at run time, follow these steps:

1. Use the **C1ReportDesigner** application to create all the reports you will need. (See Working with C1ReportDesigner for details on how to do this.)
2. Add the following controls to the application:
   - **C1Report** component named **c1r**
   - **C1PrintPreview** component named **ppv**
   - **ComboList** control named **cmbReport**
   - **StatusBar** control named **status**
3. Add the following Import statements to the top of the file:
   - Visual Basic
     ```
     Imports C1.C1Report
     Imports System.IO
     ```
   - C#
     ```
     using C1.C1Report;
     using System.IO;
     ```

   This allows you to reference the C1Report and System.IO classes and objects without having to specify the full namespaces.
4. Add code to read the report definition file and build a list of all reports in it. This can be done as follows:
   - Visual Basic
     ```
     ' get application path
     Dim appPath As String
     appPath = Path.GetDirectoryName(Application.ExecutablePath).ToLower()
     Dim i As Integer = appPath.IndexOf("/bin")
     If (i < 0) Then i = appPath.IndexOf("\bin")
     If (i > 0) Then appPath = appPath.Remove(i, appPath.Length - i)

     ' get names of reports in the report definition file
     m_ReportDefinitionFile = appPath & "\Data\Nwind.xml"
     Dim reports As String() = c1r.GetReportInfo(m_ReportDefinitionFile)

     ' populate combo box
     cmbReport.Items.Clear()
     Dim report As String
     For Each report In reports
       cmbReport.Items.Add(report)
     Next
     ```
   - C#
     ```
     // get application path
      string appPath;
     appPath = Path.GetDirectoryName(Application.ExecutablePath).ToLower();
      int i = appPath.IndexOf("/bin");
     ```

```
if ((i < 0) ) { i = appPath.IndexOf("\bin"); }
if ((i > 0) ) { appPath = appPath.Remove(i, appPath.Length - i); }

// get names of reports in the report definition file
m_ReportDefinitionFile = appPath + "\Data\Nwind.xml";
 string ( reports) = c1r.GetReportInfo(m_ReportDefinitionFile);

// populate combo box
cmbReport.Items.Clear();
 string report;
foreach report In reports
  cmbReport.Items.Add(report);
}
```

The code starts by getting the location of the file that contains the report definitions. This is done using static methods in the system-defined **Path** and **Application** classes. You may have to adjust the code to reflect the location and name of your report definition file.

Then it uses the GetReportInfo method to retrieve an array containing the names of all reports in the report definition file (created in step 1), and populates the combo box that will allow users to select the report.

5. Add code to render the report selected by the user. For example:

- Visual Basic

```
Private Sub cmbReport_SelectedIndexChanged(ByVal sender As Object,
ByVal e As EventArgs) Handles cmbReport.SelectedIndexChanged
  Try
    Cursor = Cursors.WaitCursor

    ' load report
    status.Text = "Loading " & cmbReport.Text
    c1r.Load(m_ReportDefinitionFile, cmbReport.Text)

    ' render into print preview control
    status.Text = "Rendering " & cmbReport.Text
    ppv.Document = c1r.Document

    ' give focus to print preview control
    ppv.StartPage = 0
    ppv.Focus()

    Finally
      Cursor = Cursors.Default
    End Try
End Sub
```

- C#

```
private void cmbReport_SelectedIndexChanged(object sender,
System.EventArgs e)
{
  try {
    Cursor = Cursors.WaitCursor;

    // load report
    status.Text = "Loading " + cmbReport.Text;
    c1r.Load(m_ReportDefinitionFile, cmbReport.Text);

    // render into print preview control
    status.Text = "Rendering " + cmbReport.Text;
    ppv.Document = c1r.Document;

    // give focus to print preview control
    ppv.StartPage = 0;
    ppv.Focus();

    } finally {
      Cursor = Cursors.Default;
    }
}
```

### Customizable Reports

Customizable reports are a variation on Reports Loaded at Run Time. This scenario consists of loading the basic report definitions from a file, then writing code to customize the reports according to user selections.

For example, the following code changes the font used in the Detail section:

- Visual Basic

```
Imports C1.C1Report
…
Dim s As Section = c1r.Sections(SectionTypeEnum.Detail)
Dim f As Field
For Each f In s.Fields
  f.Font.Name = "Arial Narrow"
Next
```

- C#

```
using C1.C1Report;
…
Section s = c1r.Sections[SectionTypeEnum.Detail];
foreach (Field f in s.Fields)
  f.Font.Name = "Arial Narrow";
```

The following code toggles the display of a group by turning its Sort property on or off and setting the Visible property of the Group's Header and Footer sections:

- Visual Basic

```
Dim bShowGroup As Boolean
bShowGroup = True
With c1r.Groups(0)
  If bShowGroup Then
    .SectionHeader.Visible = True
    .SectionFooter.Visible = True
    .Sort = SortEnum.Ascending
  Else
    .SectionHeader.Visible = False
    .SectionFooter.Visible = False
    .Sort = SortEnum.NoSort
    End If
End With
```

- C#

```
bool bShowGroup;
bShowGroup = true;
  if (bShowGroup)
  {
    c1r.Groups[0].SectionHeader.Visible = true;
    c1r.Groups[0].SectionFooter.Visible = true;
    c1r.Groups[0].Sort = SortEnum.Ascending;
  }
  else
  {
    c1r.Groups[0].SectionHeader.Visible = false;
    c1r.Groups[0].SectionFooter.Visible = false;
    c1r.Groups[0].Sort = SortEnum.NoSort;
  }
```

These samples illustrate just a few ways that you can customize reports. There are infinite possibilities, because the object model offers access to every aspect of the report. (In fact, you can create whole reports entirely with code).

## Developing Reports for Web Scenarios

If you are developing reports for the Web (ASP.NET), you can use the **C1WebReport** control included with the **ComponentOne Studio Enterprise** package. This control encapsulates the C1Report component and provides methods and properties that make it very easy to add reports to your Web pages. The **C1WebReport** control is 100% compatible with C1Report, and provides advanced caching and rendering options designed specifically for Web scenarios, as well as the usual design time editing options provided by ASP.NET server controls.

You can still use the C1Report component in your Web applications if you like, but you will have to write some code to create HTML or PDF versions of the reports. The following sections describe how to do this.

In typical Web scenarios, C1Report runs on the server machine and creates reports either in batch mode or on demand. The user can select the reports and preview or print them on the client machine, using a Web browser.

### *Static Web Reports*

Static Web reports are based on a server application that runs periodically and creates a predefined set of reports, saving them to HTML or PDF files. These files are referenced by Web pages on your site, and they are downloaded to the client machine like any other Web page.

To implement this type of application, follow these steps:

1. Use the **C1ReportDesigner** application to create all the reports you will need. (See Working with C1ReportDesigner for details on how to do this.)

2. Create an application on the server that contains a C1Report component. If you don't want to use forms and windows, create the control using the **CreateObject** function.

3. Add a routine that runs periodically and updates all the reports you want to make available to your users. The loop would look like this:

   - Visual Basic

   ```
   ' this runs every 6 hours:

   ' get a list of all reports in the definition file
   sFile = "c:\inetpub\wwwroot\Reports\MyReports.xml"
   sList = c1r.GetReportInfo(sFile)

   ' refresh the reports on the server
   For i = 0 To sList.Length - 1
     c1r.Load(sFile, sList(i))
     sFile = "Reports\Auto\" & sList(i) & ".htm"
     c1r.RenderToFile(sFile, FileFormatEnum.HTMLPaged)
   Next
   ```

   - C#

   ```
   // this runs every 6 hours:
   ```

```
// get a list of all reports in the definition file
sFile = "c:\inetpub\wwwroot\Reports\MyReports.xml";
sList = c1r.GetReportInfo(sFile);

// refresh the reports on the server
for ( i = 0 ; GAIS <= sList.Length – 1
  c1r.Load(sFile, sList(i));
  sFile = "Reports\Auto\" + sList(i) + ".htm";
  c1r.RenderToFile(sFile, FileFormatEnum.HTMLPaged);
}
```

The code uses the GetReportInfo method to retrieve a list of all reports contained in the MyReports.xml report definition file (created in step 1), then renders each report into a paged HTML file. (Paged HTML files contain one HTML page for each page in the original report, with a navigation bar that allows browsing.)

4. Edit the home HTML page by adding links to the reports that were saved.

You are not restricted to HTML. C1Report can also export to PDF files, which can be viewed on any browser with freely available plug-ins. The PDF format is actually superior to HTML in many ways, especially when it comes to producing hard copies of your Web reports.

### *Dynamic Web Reports*

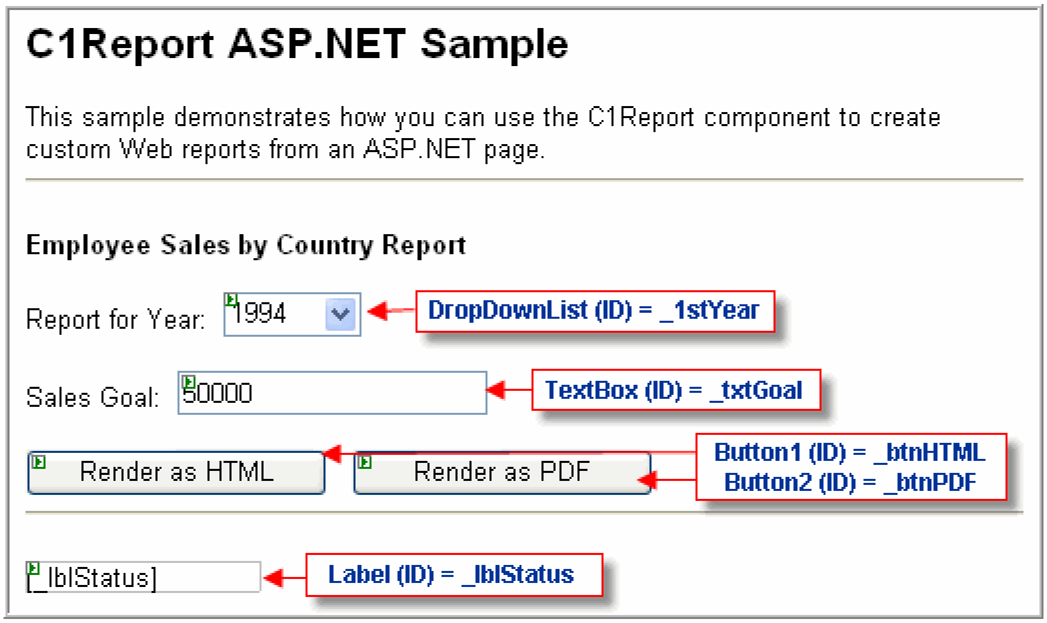
Dynamic Web reports are created on-demand, possibly based on data supplied by the user. This type of solution typically involves using an ASP.NET page that presents a form to the user and collects the information needed to create the report, then creates a C1Report component to render the report into a temporary file, and returns a reference to that file.

The example that follows is a simple ASP.NET page that allows users to enter some information and to select the type of report they want. Based on this, the ASP code creates a custom version of the NorthWind "Employee Sales by Country" report and presents it to the user in the selected format.

The sample uses a temporary file on the server to store the report. In a real application, you would have to generate unique file names and delete them after a certain time, to avoid overwriting reports before the users get a chance to see them. Despite this, the sample illustrates the main techniques involved in delivering reports over the Web with C1Report.

To implement this type of application, follow these steps:

1. Start by creating a new Web application with a Web page that looks like this:



C1Report ASP.NET Sample

This sample demonstrates how you can use the C1Report component to create custom Web reports from an ASP.NET page.

Employee Sales by Country Report

The page has five server controls:

- **_1stYear:** Contains a list of valid years for which there is data (1994, 1995, and 1996). Note that you can add the items by clicking the smart tag (▶) and selecting **Edit Items** from the menu. From the **ListItem Collection Editor**, add three new items.
- **_txtGoal:** Contains the yearly sales goal for each employee.
- **_btnHTML**, **_btnPDF:** Buttons used to render the report into HTML or PDF, and show the result.
- **_lblStatus:** Displays error information if something goes wrong.

> **Note:** If you run this application with a demo or beta version of the C1Report component, there will be errors when the control tries to display its **About** dialog box on the server. If that happens, simply reload the page and the problem should go away.

2. After the page has been set up, you need to add a reference to the C1Report component to the project. Just right-click the project in the **Solution Explorer** window, select **Add Reference** and choose the C1Report component.
3. Add the Nwind.xml to the project in the Data folder. Just right-click the project in the **Solution Explorer** window, select **New Folder** and rename the folder **Data**. Then right-click the folder, select **Add Existing Item** and browse for the **Nwind.xml** definition file, which is installed by default in the ComponentOne Samples\C1Report\C1Report\VB\NorthWind\Data directory in the **Documents** or **My Documents** folder.
4. Add a **Temp** folder to the project. Just right-click the project in the **Solution Explorer** window, select **New Folder** and rename the folder **Temp**.
5. If you have used traditional ASP, this is where things start to become interesting. Double-clicking the controls will take you to a code window where you can write full-fledged code to handle the events, using the same editor and environment you use to write Windows Forms projects.

   Add the following code:

   - Visual Basic

```
Imports C1.C1Report
…
```

```
' handle user clicks
Private Sub _btnHTML_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles _btnHTML.Click
     RenderReport(FileFormatEnum.HTMLDrillDown)
End Sub

Private Sub _btnPDF_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles _btnPDF.Click
     RenderReport(FileFormatEnum.PDF)
End Sub
```

- C#

```
using C1.C1Report;
…
// handle user clicks
private void _btnHTML_Click(object sender, System.EventArgs e)
{
   RenderReport(FileFormatEnum.HTMLDrillDown);
}
private void _btnPDF_Click(object sender, System.EventArgs e)
{
   RenderReport(FileFormatEnum.PDF);
}
```

This is the code that gets executed on the server, when the user clicks either button.

6. The following code delegates the work to the main routine, **RenderReport**:

- Visual Basic

```
Private Sub RenderReport(ByVal fmt As FileFormatEnum)

    ' build file names
    Dim rpt As String = "Employee Sales by Country"
    Dim fileIn As String = GetDataPath() & "NWind.xml"
    Dim ext As String = Iif(fmt = FileFormatEnum.PDF, ".pdf", ".htm")
    Dim fileOut As String = GetOutputPath() & rpt & ext

    Try
        ' create C1Report component
        Dim c1r As New C1Report()

        ' load the report
        c1r.Load(fileIn, rpt)

        ' get user parameters
        Dim year As String = _lstYear.SelectedItem.Text
        Dim goal As String = _txtGoal.Text

        ' customize the report data source
        Dim sSQL As String = "SELECT DISTINCTROW " & _
            "Employees.Country, Employees.LastName, " & _
            "Employees.FirstName, Orders.ShippedDate, Orders.OrderID, "
& _
            "  [Order Subtotals].Subtotal AS SaleAmount " & _
            "FROM Employees INNER JOIN (Orders INNER JOIN " & _
            "  [Order Subtotals] ON Orders.OrderID = " & _
            "  [Order Subtotals].OrderID) " & _
            "  ON Employees.EmployeeID = Orders.EmployeeID " & _
            "WHERE Year(Orders.ShippedDate) = " & year & ";"
        c1r.DataSource.RecordSource = sSQL

        ' customize the report's event handlers
        Dim sScript As String = _
            "If SalespersonTotal > " & goal & " Then" & vbCrLf & _
            "  ExceededGoalLabel.Visible = True" & vbCrLf & _
            "  SalespersonLine.Visible = True" & vbCrLf & _
            "Else" & vbCrLf & _
            "  ExceededGoalLabel.Visible = False" & vbCrLf & _
            "  SalespersonLine.Visible = False" & vbCrLf & _
            "End If"
        c1r.Sections(SectionTypeEnum.GroupHeader2).OnPrint = sScript

        ' render the report into a temporary file
        c1r.RenderToFile(fileOut, fmt)

        ' redirect user to report file
        Response.Redirect("Temp/" + rpt + ext)

    Catch x As Exception

        _lblStatus.Text = "*** " & x.Message

    End Try
End Sub
```

- C#

```
// render the report
private void RenderReport(FileFormatEnum fmt)
{
   // build file names
   string rpt  = "Employee Sales by Country";
   string fileIn = GetDataPath() + "NWind.xml";
   string ext = (fmt == FileFormatEnum.PDF)? ".pdf": ".htm";
   string fileOut = GetOutputPath() + rpt + ext;

   try
   {
   // create C1Report component
   C1Report c1r = new C1Report();

   // load the report
   c1r.Load(fileIn, rpt);

   // get user parameters
   string year = _lstYear.SelectedItem.Text;
   string goal = _txtGoal.Text;

   // customize the report data source
   string sSQL = "SELECT DISTINCTROW " +
   "Employees.Country, Employees.LastName, " +
   "Employees.FirstName, Orders.ShippedDate, Orders.OrderID, " +
   "  [Order Subtotals].Subtotal AS SaleAmount " +
   "FROM Employees INNER JOIN (Orders INNER JOIN " +
   "  [Order Subtotals] ON Orders.OrderID = " +
   "  [Order Subtotals].OrderID) " +
   "  ON Employees.EmployeeID = Orders.EmployeeID " +
   "WHERE Year(Orders.ShippedDate) = " + year + ";";
   c1r.DataSource.RecordSource = sSQL;

   // customize the report's event handlers
   string sScript =
   "If SalespersonTotal > " + goal + " Then \n" +
   "  ExceededGoalLabel.Visible = True\n" +
   "  SalespersonLine.Visible = True\n" +
   "Else\n" +
   "  ExceededGoalLabel.Visible = False\n" +
   "  SalespersonLine.Visible = False\n" +
   "End If";
   c1r.Sections[SectionTypeEnum.GroupHeader2].OnPrint = sScript;

   // render the report into a temporary file
   c1r.RenderToFile(fileOut, fmt);
   // redirect user to report file
   Response.Redirect("Temp/" + rpt + ext);
   }
   catch (Exception x)
   {
   _lblStatus.Text = "*** " + x.Message;
   }
}
```

The **RenderReport** routine is long, but pretty simple. It begins working out the names of the input and output files. All file names are built relative to the current application directory.

Next, the routine creates a C1Report component and loads the "Employee Sales by Country" report. This is the raw report, which you will customize in the next step.

The parameters entered by the user are available in the **_lstYear** and **_txtGoal** server-side controls. The code reads these values and uses them to customize the report's **RecordSource** property and to build a VBScript handler for the OnPrint property. These techniques were discussed in previous sections.

Once the report definition is ready, the code calls the RenderToFile method, which causes the C1Report component to write HTML or PDF files to the output directory. When the method returns, the report is ready to be displayed to the user.

The last step is the call to **Response.Redirect**, which displays the report you just created on the user's browser.

Note that the whole code is enclosed in a **try/catch** block. If anything goes wrong while the report is being generated, the user gets to see a message explaining the problem.

7. Finally, there's a couple of simple helper routines that need to be added:

- Visual Basic

```
' get directories to use for loading and saving files
Private Function GetDataPath() As String
   Return Request.PhysicalApplicationPath + "Data\"
End Function

Private Function GetOutputPath() As String
   Return Request.PhysicalApplicationPath + "Temp\"
End Function
```

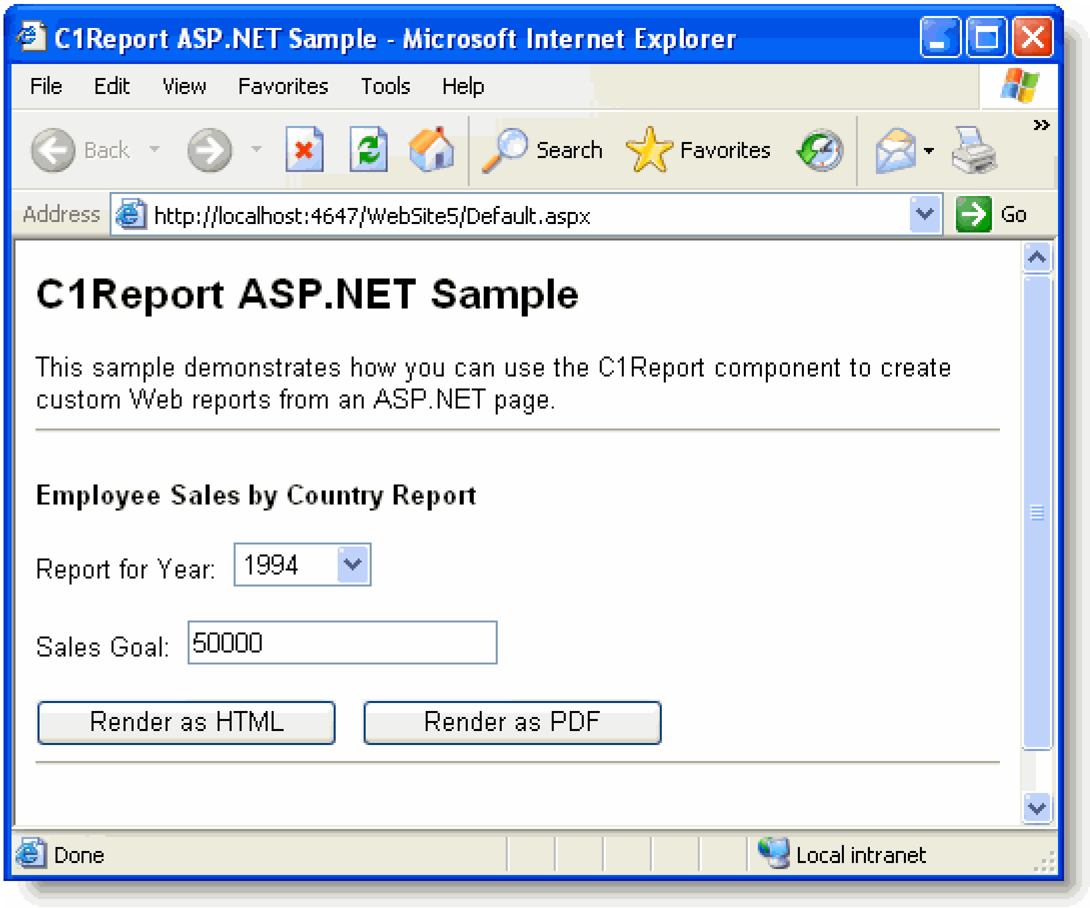
- C#

```
// get directories to use for loading and saving files
private string GetDataPath()
```

```
{
   return Request.PhysicalApplicationPath + @"Data\";
}
private string GetOutputPath()
{
   return Request.PhysicalApplicationPath + @"Temp\";
}
```

8. After you enter this code, the application is ready. You can press F5 and trace its execution within Visual Studio.

The following screen shot shows what the result looks like in the browser:



C1Report ASP.NET Sample - Microsoft Internet Explorer

Address: http://localhost:4647/WebSite5/Default.aspx

**C1Report ASP.NET Sample**

This sample demonstrates how you can use the C1Report component to create custom Web reports from an ASP.NET page.

**Employee Sales by Country Report**

Report for Year: 1994

Sales Goal: 50000

Done

Local intranet

## Creating, Loading, and Rendering the Report

Although you can use C1Report in many different scenarios, on the desktop and on the Web, the main sequence of steps is always the same:

1. **Create a report definition**

   This can be done directly with the **C1Report Designer** application or using the report designer in Microsoft Access and then importing it into **C1Report Designer**. You can also do it using code, either using the object model to add groups and fields or by writing a custom XML file.

2. **Load the report into the C1Report component**

This can be done at design time, using the **Load Report** context menu, or programmatically using the Load method. If you load the report at design time, it will be persisted (saved) with the control and you won't need to distribute the report definition file.

3. **Render the report (desktop applications)**

   If you are writing a desktop application, you can render the report into a **C1PrintPreview** controleasily. The preview control will display the report on the screen, and users will be able to preview it with full zooming, panning, and so on. For example:

   - Visual Basic
     ```
     C1PrintPreview1.Document = c1r
     ```
   - C#
     ```
     c1printPreview1.Document = c1r;
     ```

4. **Render the report (Web applications)**

   If you are writing a Web application, you can render reports into HTML or PDF files using the RenderToFile method, and your users will be able to view them using any browser.

## *Creating a Report Definition*

The following topics show how you can create a report definition using the **C1ReportDesigner** application or using code. Note that creating a report definition is not the same as rendering a report. To render a report, you can simply load an existing definition and call the Render method.

### Creating a Report Definition Using C1ReportDesigner

The easiest way to create a report definition is to use the **C1ReportDesigner** application, which is a stand alone application similar to the report designer in Microsoft Access. The **C1Report Wizard** walks you through the steps of creating a new report from start to finish.

For information on the steps for creating a new report, see Creating a Basic Report Definition.

### Creating a Report Definition Using Code

You can also create reports from scratch, using code. This approach requires some extra work, but it gives you complete flexibility. You can even write your own report designer or ad-hoc report generator.

The following example uses code to create a simple tabular report definition based on the NorthWind database. The code is commented and illustrates the most important elements of the C1Report object model. Complete the following steps:

1. First, add a button control, C1Report component, and **C1PrintPreview** control to your form. Set the following properties:

   Button.Name = **btnEmployees**

   C1Report.Name = **c1r**

   C1PrintPreview.Name = **ppv**

2. Initialize the control, named **c1r**, using the Clear method to clear its contents and set the control font (this is the font that will be assigned to new fields):

   - Visual Basic
     ```
     Private Sub RenderEmployees()
     With c1r
       ' clear any existing fields
       .Clear()
       ' set default font for all controls
       .Font.Name = "Tahoma"
       .Font.Size = 8
     End With
     ```

- C#

```
private void RenderEmployees()
{
  // clear any existing fields
  c1r.Clear();
  // set default font for all controls
  c1r.Font.Name = "Tahoma";
  c1r.Font.Size = 8;
}
```

3. Next, set up the DataSource object to retrieve the data that you want from the NorthWind database. This is done using the ConnectionString and RecordSource properties (similar to the Microsoft ADO DataControl):

   - Visual Basic

```
' initialize DataSource
With c1r.DataSource
  .ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
                      "Data Source=C:\…\ComponentOne
Samples\Common\Nwind.mdb;" & _
                      "Persist Security Info=False"
  .RecordSource = "Employees"
End With
```

   - C#

```
// initialize DataSource
DataSource ds = c1r.DataSource;
ds.ConnectionString = @"Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\…\ComponentOne Samples\Common\Nwind.mdb;";
ds.RecordSource = "Employees";
```

4. Next, initialize the Layout object that defines how the report will be laid out on the page. In this case, render the report in Portrait mode and set its Width to 6.5 inches (8.5 page width minus one inch for margins on either side):

   - Visual Basic

```
' initialize Layout
With c1r.Layout
  .Orientation = OrientationEnum.Portrait
  .Width = 6.5 * 1440 ' 8.5 - margins, in twips
End With
```

   - C#

```
// initialize Layout
Layout l = c1r.Layout;
l.Orientation = OrientationEnum.Portrait;
l.Width = 6.5 * 1440; // 8.5 - margins, in twips
```

5. Now comes the interesting part. Every report has five basic sections: Detail, Report Header, Report Footer, Page Header, and Page Footer. Use the following code to set up the report header by setting a couple of properties and adding a title field to it:

   - Visual Basic

```
' variable used to create and format report fields
Dim f As Field

' create a report header
 With c1r.Sections(SectionTypeEnum.Header)
  .Height = 1440
  .Visible = True
  .BackColor = Color.FromArgb(200, 200, 200)
  f = .Fields.Add("FldTitle", "Employees Report", 0, 0, 8000, 1440)
  f.Font.Size = 24
  f.Font.Bold = True
  f.ForeColor = Color.FromArgb(0, 0, 100)
End With
```

- C#

```
// variable used to create and format report fields
 Field f;

// create a report header
Section s = c1r.Sections[SectionTypeEnum.Header];
  s.Height = 1440;
  s.Visible = true;
  s.BackColor = Color.FromArgb(200, 200, 200);
  f = s.Fields.Add("FldTitle", "Employees Report", 0, 0, 8000, 1440);
  f.Font.Size = 24;
  f.Font.Bold = true;
  f.ForeColor = Color.FromArgb(0, 0, 100);
```

The section object has a Fields collection. The collection's Add method creates a new field and assigns it to the Section. The parameters specify the new field's Name, Text, Left, Top, Width, and Height properties. By default, the field has the same font as the control. Since this is a title, it makes sense to change the font and make it larger. Note that the field should be tall enough to accommodate the font size, or nothing will appear in it.

6. Next, set up the Page Footer Section. This section is more interesting because it contains calculated fields. Calculated fields contain VBScript expressions in their Text property, which are evaluated when the report is rendered. To make a field calculated, set its Calculated property to **True**. To create a page footer, add the following code:

- Visual Basic

```
' create a page footer
With c1r.Sections(SectionTypeEnum.PageFooter)
  .Height = 500
  .Visible = True
  f = .Fields.Add("FldFtrLeft", """Employees: Printed on "" & Now",_
                  0, 0, 4000, 300)
  f.Calculated = True
  f = .Fields.Add("FldFtrRight", """Page "" & Page & "" of "" &
Pages",_
                  4000, 0, 4000, 300)
  f.Calculated = True
  f.Align = FieldAlignEnum.RightTop
  f.Width = c1r.Layout.Width - f.Left
  f = .Fields.Add("FldLine", "", 0, 0, c1r.Layout.Width, 20)
  f.LineSlant = LineSlantEnum.NoSlant
  f.BorderStyle = BorderStyleEnum.Solid
  f.BorderColor = Color.FromArgb(0, 0, 100)
End With
```

- C#

```
// create a page footer
s = c1r.Sections[SectionTypeEnum.PageFooter];
  s.Height = 500;
  s.Visible = true;
  f = s.Fields.Add("FldFtrLeft", @"""Employees: Printed on "" & Now",_
                  0, 0, 4000, 300);
  f.Calculated = true;
  f = .Fields.Add("FldFtrRight", @"""Page "" + Page + "" of "" &
Pages",_
                  4000, 0, 4000, 300);
  f.Calculated = true;
  f.Align = FieldAlignEnum.RightTop;
  f.Width = c1r.Layout.Width - f.Left;
  f = s.Fields.Add("FldLine", "", 0, 0, c1r.Layout.Width, 20);
  f.LineSlant = LineSlantEnum.NoSlant;
  f.BorderStyle = BorderStyleEnum.Solid;
  f.BorderColor = Color.FromArgb(0, 0, 100);
```

The Page Footer section uses expressions with variables that are not intrinsic to VBScript, but are defined by C1Report. **Page** and **Pages** are variables that contain the current page number and the total page count. The section also uses a field configured to look like a line. This is done using the BorderStyle and LineSlant properties.

7. Next, set up the Page Header Section. This section gets rendered at the top of every page and will display the field labels. Using a Page Header section to display field labels is a common technique in tabular reports. The code is simple, but looks a bit messy because of all the field measurements. In a real application, these values would not be hard-wired into the program. To create a page header with field labels, add the following code:

- Visual Basic

```
' create a page header with field labels
With c1r.Sections(SectionTypeEnum.PageHeader)
  .Height = 500
  .Visible = True
  c1r.Font.Bold = True
  f = .Fields.Add("LblID", "ID", 0, 50, 400, 300)
  f.Align = FieldAlignEnum.RightTop
  f = .Fields.Add("LblFirstName", "First", 500, 50, 900, 300)
  f = .Fields.Add("LblLastName", "Last", 1500, 50, 900, 300)
  f = .Fields.Add("LblTitle", "Title", 2500, 50, 2400, 300)
  f = .Fields.Add("LblTitle", "Notes", 5000, 50, 8000, 300)
  c1r.Font.Bold = False
  f = .Fields.Add("FldLine", "", 0, 400, c1r.Layout.Width, 20)
  f.LineSlant = LineSlantEnum.NoSlant
  f.LineWidth = 50

  f.BorderColor = Color.FromArgb(100, 100, 100)
End With
```

- C#

```
// create a page header with field labels
s = c1r.Sections[SectionTypeEnum.PageHeader];
  s.Height = 500;
  s.Visible = true;
  c1r.Font.Bold = true;
  f = s.Fields.Add("LblID", "ID", 0, 50, 400, 300);
  f.Align = FieldAlignEnum.RightTop;
  f = s.Fields.Add("LblFirstName", "First", 500, 50, 900, 300);
  f = s.Fields.Add("LblLastName", "Last", 1500, 50, 900, 300);
  f = s.Fields.Add("LblTitle", "Title", 2500, 50, 2400, 300);
  f = s.Fields.Add("LblTitle", "Notes", 5000, 50, 8000, 300);
  c1r.Font.Bold = false;
  f = s.Fields.Add("FldLine", "", 0, 400, c1r.Layout.Width, 20);
  f.LineSlant = LineSlantEnum.NoSlant;
  f.LineWidth = 50;
  f.BorderColor = Color.FromArgb(100, 100, 100);
```

This code illustrates a powerful technique for handling fonts. Since every field inherits the control font when it is created, set the control's Font.**Bold** property to **True** before creating the fields, and set it back to **False** afterwards. As a result, all controls in the Page Header section have a bold font.

8. To finalize the report, add the Detail Section. This is the section that shows the actual data. It has one calculated field below each label in the Page Header Section. To create the Detail section, add the following code:

    - Visual Basic

    ```
    ' create the Detail section
    With c1r.Sections(SectionTypeEnum.Detail)
      .Height = 330
      .Visible = True
      f = .Fields.Add("FldID", "EmployeeID", 0, 0, 400, 300)
      f.Calculated = True
      f = .Fields.Add("FldFirstName", "FirstName", 500, 0, 900, 300)
      f.Calculated = True
      f = .Fields.Add("FldLastName", "LastName", 1500, 0, 900, 300)
      f.Calculated = True
      f = .Fields.Add("FldTitle", "Title", 2500, 0, 2400, 300)
      f.Calculated = True
      f = .Fields.Add("FldNotes", "Notes", 5000, 0, 8000, 300)
      f.Width = c1r.Layout.Width - f.Left
      f.Calculated = True
      f.CanGrow = True
      f.Font.Size = 6
      f.Align = FieldAlignEnum.JustTop
      f = .Fields.Add("FldLine", "", 0, 310, c1r.Layout.Width, 20)
      f.LineSlant = LineSlantEnum.NoSlant
      f.BorderStyle = BorderStyleEnum.Solid
      f.BorderColor = Color.FromArgb(100, 100, 100)
    End With
    ```

    - C#

    ```
    // create the Detail section
    s = c1r.Sections[SectionTypeEnum.Detail];
      s.Height = 330;
      s.Visible = true;
      f = s.Fields.Add("FldID", "EmployeeID", 0, 0, 400, 300);
      f.Calculated = true;
      f = s.Fields.Add("FldFirstName", "FirstName", 500, 0, 900, 300);
      f.Calculated = true;
      f = s.Fields.Add("FldLastName", "LastName", 1500, 0, 900, 300);
      f.Calculated = true;
      f = s.Fields.Add("FldTitle", "Title", 2500, 0, 2400, 300);
      f.Calculated = true;
      f = s.Fields.Add("FldNotes", "Notes", 5000, 0, 8000, 300);
      f.Width = c1r.Layout.Width - f.Left;
      f.Calculated = true;
      f.CanGrow = true;
      f.Font.Size = 6;
      f.Align = FieldAlignEnum.JustTop;
      f = s.Fields.Add("FldLine", "", 0, 310, c1r.Layout.Width, 20);
      f.LineSlant = LineSlantEnum.NoSlant;
      f.BorderStyle = BorderStyleEnum.Solid;
    ```

```
  f.BorderColor = Color.FromArgb(100, 100, 100);
```

Note that all fields are calculated, and their Text property corresponds to the names of fields in the source recordsetsource. Setting the Calculated property to **True** ensures that the Text property is interpreted as a database field name, as opposed to being rendered literally. It is important to adopt a naming convention for report fields that makes them unique, different from recordset field names. If you had two fields named "LastName", an expression such as "Left(LastName,1)" would be ambiguous. This example has adopted the convention of beginning all report field names with "Fld".

Note also that the "FldNotes" field has its CanGrow property set to **True**, and a smaller font than the others. This was done to make sure that the "Notes" field in the database, which contains a lot of text, will appear in the report. Rather than make the field very tall and waste space, setting the CanGrow property to **True** tells the control to expand the field as needed to fit its contents; it also sets the containing section's CanGrow property to **True**, so the field doesn't spill off the Section.

9. The report definition is done. To render the report into the **C1PrintPreview** control, double-click the **Employees** button to add an event handler for the **btnEmployees_Click** event. The Code Editor will open with the insertion point placed within the event handler. Enter the following code:

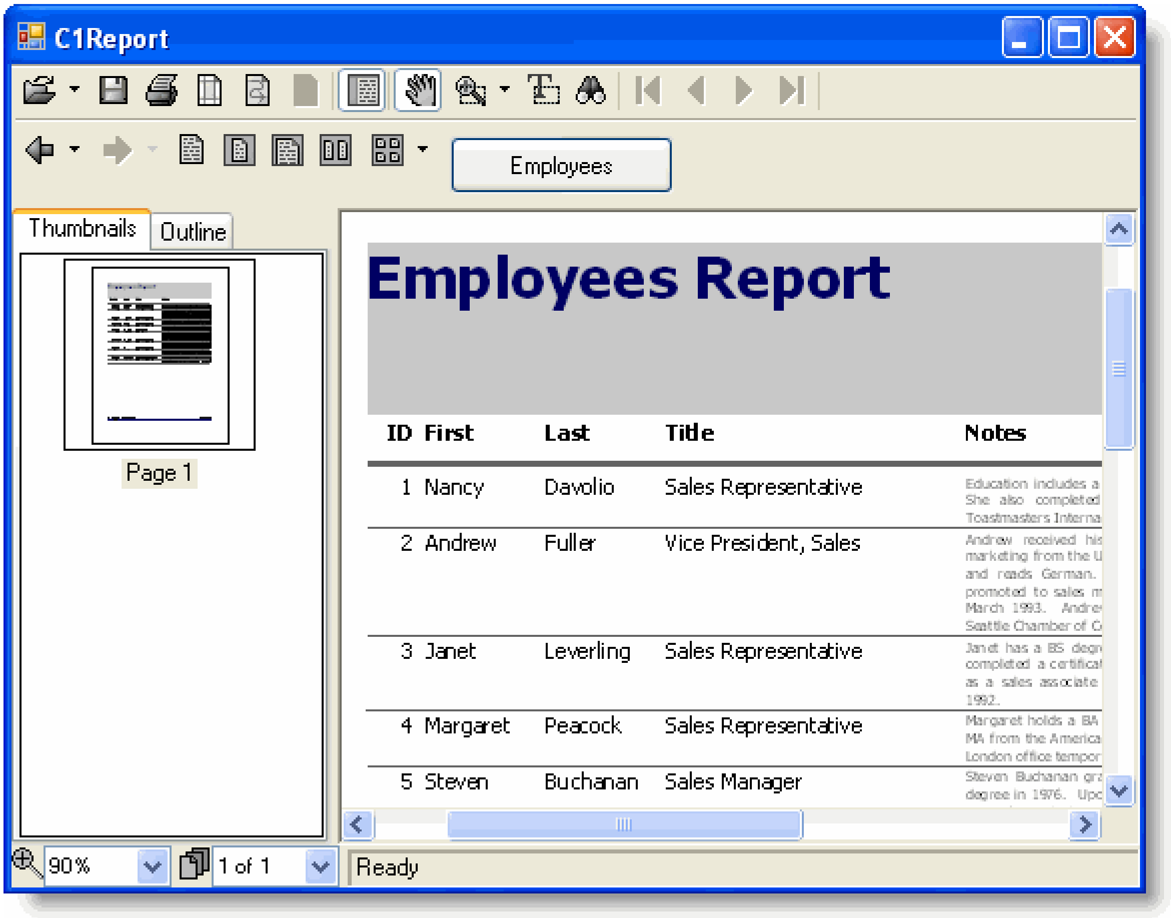
- Visual Basic

```
RenderEmployees()
' render the report into the C1PrintPreview control
ppv.Document = c1r.Document
```

- C#

```
RenderEmployees();
// render the report into the C1PrintPreview control
ppv.Document = c1r.Document;
```

Here's what the basic report looks like:



Employees Report

| ID | First | Last | Title | Notes |
|---|---|---|---|---|
| 1 | Nancy | Davolio | Sales Representative | Education includes a She also completed Toastmasters Interna |
| 2 | Andrew | Fuller | Vice President, Sales | Andrew received his marketing from the U and reads German. promoted to sales m March 1993. Andre Seattle Chamber of C |
| 3 | Janet | Leverling | Sales Representative | Janet has a BS degr completed a certificat as a sales associate 1992. |
| 4 | Margaret | Peacock | Sales Representative | Margaret holds a BA MA from the America London office tempor |
| 5 | Steven | Buchanan | Sales Manager | Steven Buchanan gra degree in 1976. Upo |

### *Loading Report Data*

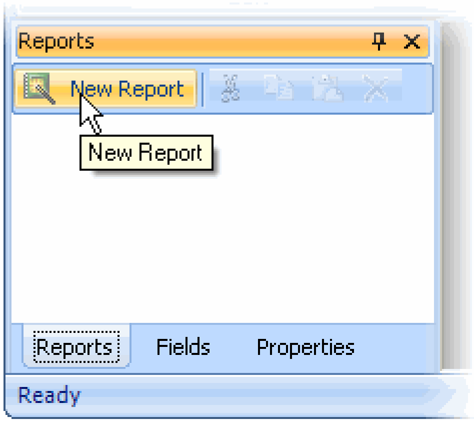
In addition to a report definition, C1Report needs the actual data to create the report. In most cases, the data comes from a database, but there are other options.

## Loading Data from a Database

The simplest option for loading the report data is to set the C1Report control's DataSource properties: ConnectionString and RecordSource. If these properties are set, C1Report uses them to load the data from the database automatically and no additional work is needed.

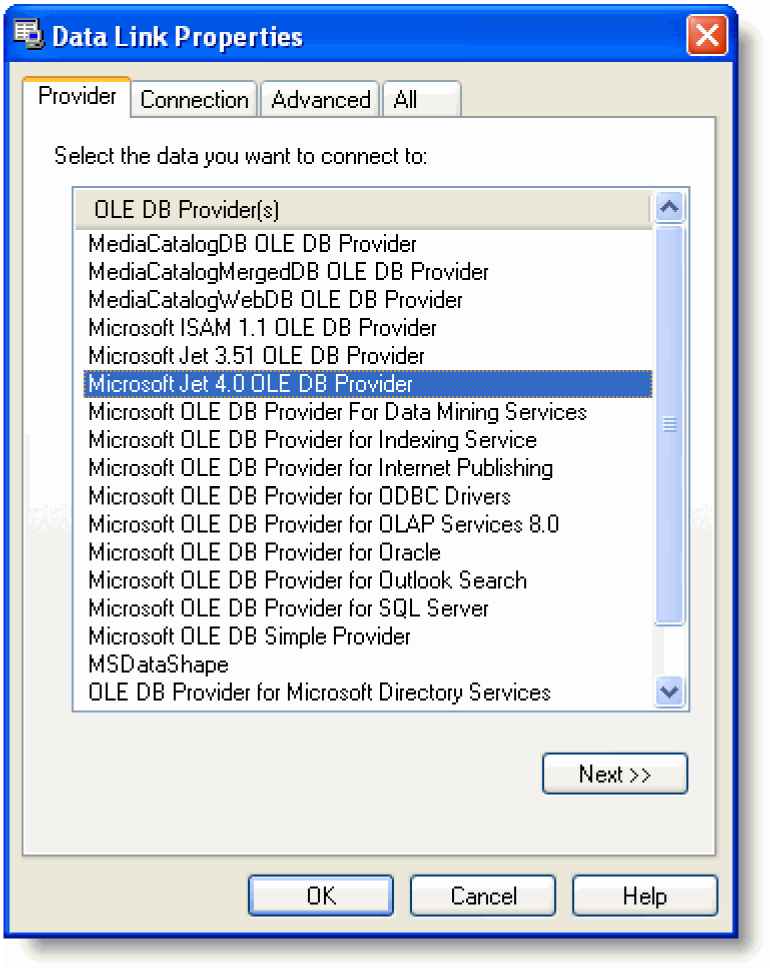
**To select a data source for a new report:**

1. In the **C1ReportDesigner** application, select the **Add New Report** button:



The **C1Report Wizard** appears and walks you through the steps of creating a new report from start to finish.

2. In the first step of the wizard you must select a data source. Click the **Build connection string** button ··· . The **Data Link Properties** dialog box appears.
3. You must first select a data provider. Select the **Provider** tab and select a data provider from the list. For this example, select **Microsoft Jet 4.0 OLE DB Provider**.



Select the data you want to connect to:

OLE DB Provider(s)

MediaCatalogDB OLE DB Provider
MediaCatalogMergedDB OLE DB Provider
MediaCatalogWebDB OLE DB Provider
Microsoft ISAM 1.1 OLE DB Provider
Microsoft Jet 3.51 OLE DB Provider
Microsoft Jet 4.0 OLE DB Provider
Microsoft OLE DB Provider For Data Mining Services
Microsoft OLE DB Provider for Indexing Service
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for OLAP Services 8.0
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for Outlook Search
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Simple Provider
MSDataShape
OLE DB Provider for Microsoft Directory Services

4. Click the **Next** button or select the **Connection** tab. Now you must choose a data source.

5. To select a database, click the **ellipsis** button. The **Select Access Database** dialog box appears. For this example, select the **Nwind.mdb** located in the **ComponentOne Samples\Common** directory in the **Documents** or **My Documents** folder. Note that this directory reflects the default installation path and its path may be different if you made changes to the installation path.

6. Click **Open**. Test the connection, if you choose, and click **OK**.

7. Click **OK** to close the **Select Access Database** dialog box.

8. Once you have selected your data source, you can select a table, view, or stored procedure to provide the actual data. You can specify the RecordSource string in two ways:

   - Click the **Table** option and select a table from the list.
   - Click the **SQL** option and type (or paste) an SQL statement into the editor.

     For example:

     ```
     select * from products inner join categories on categories.categoryid =
     products.categoryid
     ```

9. Click **Next**. The wizard will walk you through the remaining steps.

For more details on the **C1Report Wizard**, see Creating a Basic Report Definition.

## Loading Data from a Stored Procedure

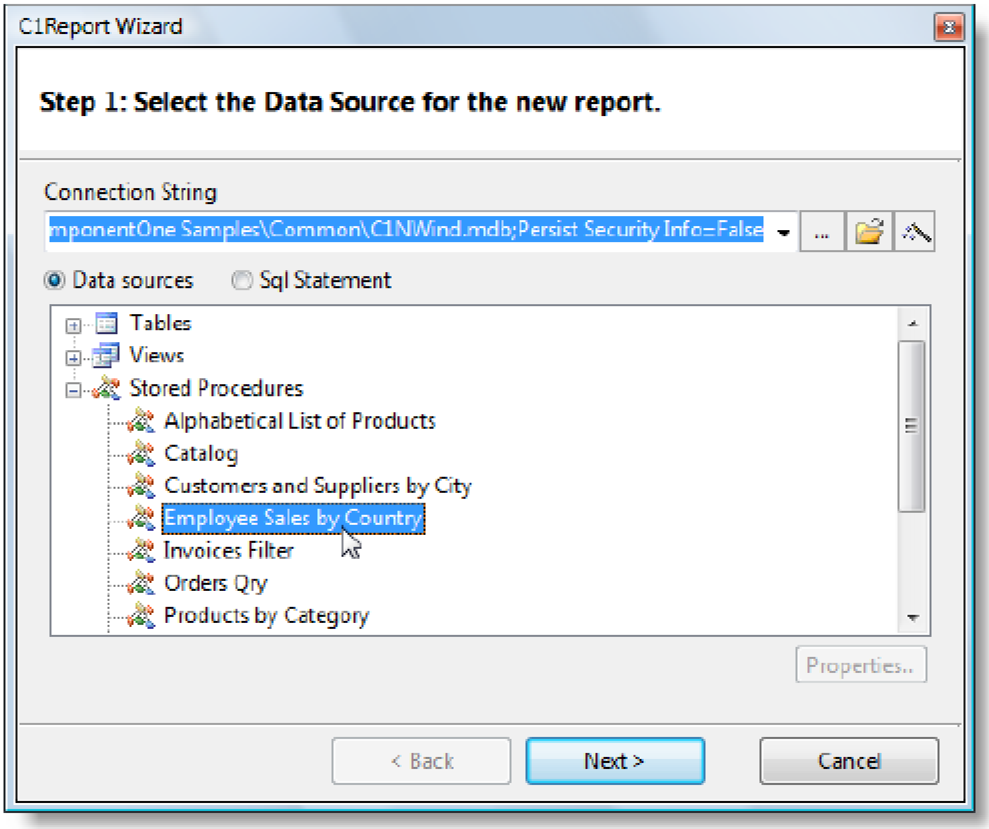
Stored procedures (or sprocs) can assist you in achieving a consistent implementation of logic across applications, improve performance, and shield users from needing to know the details of the tables in the database. One of the major advantages to stored procedures is you can pass in parameters to have the database filter the recordset. This returns a smaller set of data, which is quicker and easier for the report to manipulate.

You can populate a report from a stored procedure in the **C1Report Wizard**. To open the **C1Report Wizard** complete one of the following:

- From the **C1ReportDesigner** application, click the **New Report** button from the **Reports** tab
- In Visual Studio by select **Edit Report** from the **C1Report** context menu
- In Visual Studio by select **Edit Report** from the **C1Report Tasks** menu

For more information accessing the **Edit Report** link, see C1Report Tasks Menu or C1Report Context Menu.

Populating a report from a stored procedure is no different than using SQL statements or straight tables. In the first screen of the **C1ReportWizard**, click the **ellipses** button to choose a datasource. Then choose a **Stored Procedure** from the list of available **Data sources**:



C1Report Wizard

**Step 1: Select the Data Source for the new report.**

Connection String

mponentOne Samples\Common\C1NWind.mdb;Persist Security Info=False

● Data sources ○ Sql Statement

- Tables
- Views
- Stored Procedures
  - Alphabetical List of Products
  - Catalog
  - Customers and Suppliers by City
  - Employee Sales by Country
  - Invoices Filter
  - Orders Qry
  - Products by Category

Select **Next** and continue through the wizard.

As with loading other forms of data, you have two options:

- You can use the DataSource's ConnectionString and RecordSource properties to select the datasource:

  In the Designer, use the DataSource dialog box to select the connection string (by clicking the ellipses button "..."), then pick the table or sproc you want to use from the list. For example:

connectionstring = "Provider=SQLOLEDB.1;Integrated Security=SSPI;" + "Persist Security Info=False;Initial Catalog=Northwind;Data Source=YOURSQLSERVER;" recordsource = "[Employee Sales by Country]('1/1/1990', '1/1/2010')"

(In this case the stored procedure name has spaces, so it's enclosed in square brackets).

- You can create the data source using whatever method you want, then assign it to the DataSource's Recordset property:

  This method requires you to write code, and is useful when you have your data cached somewhere and want to use it to produce several reports. It overrides the previous method (it you specify ConnectionString, RecordSource, and Recordset, C1Report will use the Recordset).

  The syntax will be different depending on the type of connection/adapter you want to use (OleDb, SQL, Oracle, and so on). The easiest way to get the syntax right is to drag tables or sprocs from Visual Studio's Server Explorer onto a form. That will add all the cryptic elements required, and then you can go over the code and pick up the pieces you want.

You can specify stored procedures as data sources by their name. If the sproc has parameters, you pass them as parameters, for example:

## Loading Data from an XSD File

XSD files are XML Schema Documents. Reports for WinForms is compatible with XSD files, allowing you to load report data from an XSD file. To load the report data, you can set the ConnectionString property to the name of an XSD file. XSD files contain XML representations of ADO.NET DataSet objects. So in this case, you would set the RecordSource property to the name of a table in the DataSet.

> **Note:** To create an XSD file, take an existing DataSet object and call the **WriteXml** method using the XmlWriteMode.WriteSchema parameter.

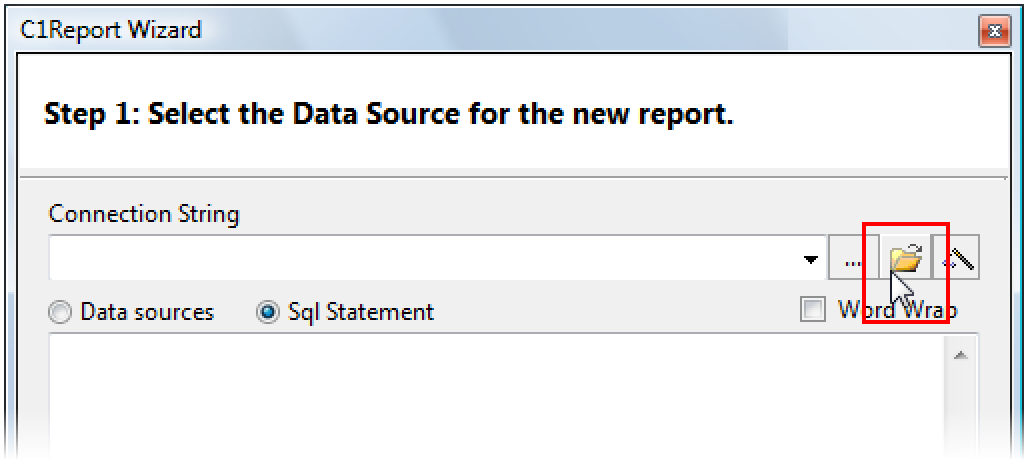
Loading an XSD file in the **C1ReportDesigner** application is quite simple using the **C1Report Wizard**. Complete the following steps to open an XSD file:

1. Open the **C1Report Wizard** by completing one of the following:

   - From the **C1ReportDesigner** application, click the **New Report** button from the **Reports** tab
   - In Visual Studio by select **Edit Report** from the **C1Report** context menu
   - In Visual Studio by select **Edit Report** from the **C1Report Tasks** menu

   This will open the **C1Report Wizard**. You can select an XSD file in the first screen of the **C1ReportWizard**. For more information accessing the **Edit Report** link, see C1Report Tasks Menu or C1Report Context Menu.

2. In the **C1Report Wizard**, click the **Select XSD file** button next to the **Connection String** text box. The **Select XSD file** button appears as an open folder icon:


C1Report Wizard

Step 1: Select the Data Source for the new report.

Connection String

○ Data sources ● Sql Statement □ Word Wrap

This button will open the **Select XML Schema Definition File** dialog box.

3. Navigate to an XSD file, select it, and click the **Open** button to open the XSD file.

   The selected file will open in the **C1ReportDesigner** application. You can edit and format the file. See Working with C1ReportDesigner for details.

## Using a DataTable Object as a Data Source

Many applications need to work on the data outside of C1Report and load it into DataTable objects. In these cases, you may use these objects as report data sources, avoiding the need to load them again when rendering the report.

This approach is also useful in applications where:

- Security restrictions dictate that connection strings must be kept private and only the data itself may be exposed (not its source).
- The database is not supported by OleDb (the provider used internally by C1Report).
- The data does not come from a database at all. Instead, the DataTable is created and populated using custom code.

To use a DataTable object as a C1Report data source, simply load the report definition and then assign the DataTable to the C1Report Recordset property. For example:

- Visual Basic

```
' load DataTable from cache or from a secure/custom provider
Dim dt As DataTable = GetMyDataTable()

' load report definition (before setting the data source)
c1r.Load(reportFile, reportName)

' use DataTable as the data source for the C1Report component
c1r.DataSource.Recordset = dt
```

- C#

```
// load DataTable from cache or from a secure/custom provider
DataTable dt = GetMyDataTable();

// load report definition (before setting the data source)
c1r.Load(reportFile, reportName);
```

```
// use DataTable as the data source for the C1Report component
c1r.DataSource.Recordset = dt;
```

## Using Custom Data Source Objects

You can use custom objects as data sources. The only requirement is that the custom object must implement the IC1ReportRecordset interface.

**IC1ReportRecordset** is a simple and easy-to-implement interface that can be added easily to virtually any collection of data. This is often more efficient than creating a **DataTable** object and copying all the data into it. For example, you could use custom data source objects to wrap a file system or custom XML files.

To use a custom data source objects, load the report definition and then assign the object to the C1Report.Recordset property. For example:

- Visual Basic

```
' get custom data source object
Dim rs As IC1ReportRecordset = CType(GetMyCustomDataSource(),
IC1ReportRecordset)

' load report definition (before setting the data source)
c1r.Load(reportFile, reportName)

' use custom data source object in C1Report component
c1r.DataSource.Recordset = rs
```

- C#

```
// get custom data source object
IC1ReportRecordset rs = (IC1ReportRecordset)GetMyCustomDataSource();

// load report definition (before setting the data source)
c1r.Load(reportFile, reportName);

// use custom data source object in C1Report component
c1r.DataSource.Recordset = rs;
```

## Grouping and Sorting Data

This section shows how you can organize the data in your report by grouping and sorting data, using running sums, and creating aggregate expressions.

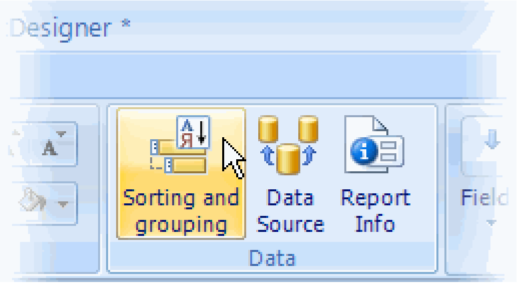
### *Grouping Data*

After designing the basic layout, you may decide that grouping the records by certain fields or other criteria would make the report easier to read. Grouping allows you to separate groups of records visually and display introductory and summary data for each group. The group break is based on a grouping expression. This expression is usually based on one or more recordset fields but it can be as complex as you like.

You can group the data in your reports using the **C1ReportDesigner** application or using code:

- Adding grouping and sorting using C1ReportDesigner

  Groups are also used for sorting the data, even if you don't plan to show the Group Header and Footer sections. You can add groups to your report using the **C1ReportDesigner** application.



To add or edit the groups in the report, complete the following steps:

a. Open the **C1ReportDesigner** application. For details, see [Accessing C1ReportDesigner from Visual Studio]().

b. Click the **Sorting and Grouping** button on the **Design** tab in the **Data** group.

   The **Sorting and Grouping** dialog box appears. You can use this dialog box to create, edit, reorder, and delete groups.

c. To create a group, click the **Add** button and set the properties for the new group.

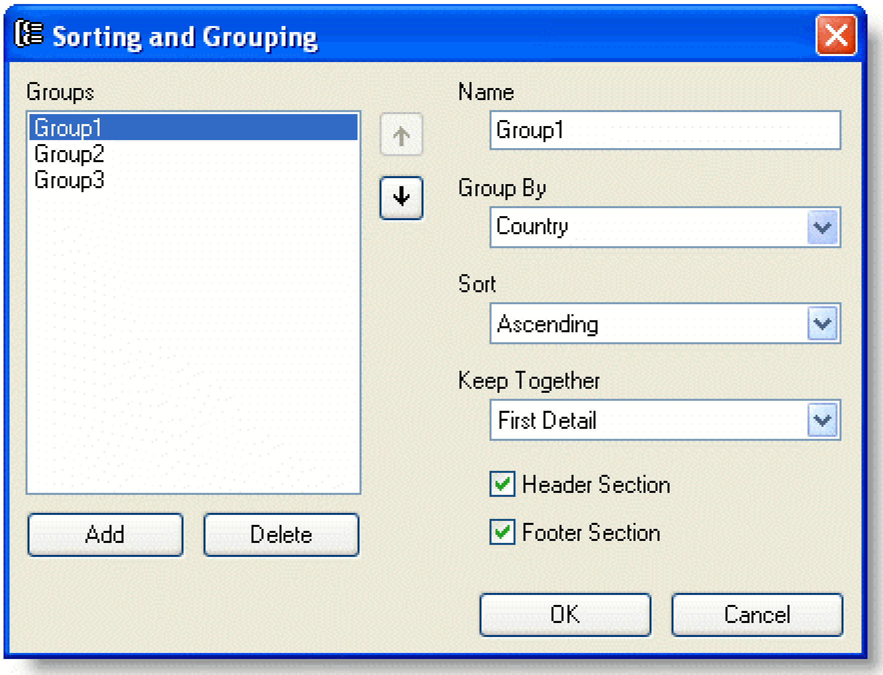
   The **Group By** field defines how the records will be grouped in the report. For simple grouping, you can select fields directly from the drop-down list. For more complex grouping, you can type grouping expressions. For example, you could use **Country** to group by country or **Left(Country, 1)** to group by country initial.

d. To follow along with this report, select **Country** for the **Group By** expression.

e. Next, select the type of sorting you want (**Ascending** in this example). You can also specify whether the new group will have visible Header and Footer sections, and whether the group should be rendered together on a page.

**Note:** You cannot use memo or binary (object) fields for grouping and sorting. This is a limitation imposed by OLEDB.

Here's what the **Sorting and Grouping** dialog box should look like at this point:



Groups
Group1
Group2
Group3

Name: Group1

Group By: Country

Sort: Ascending

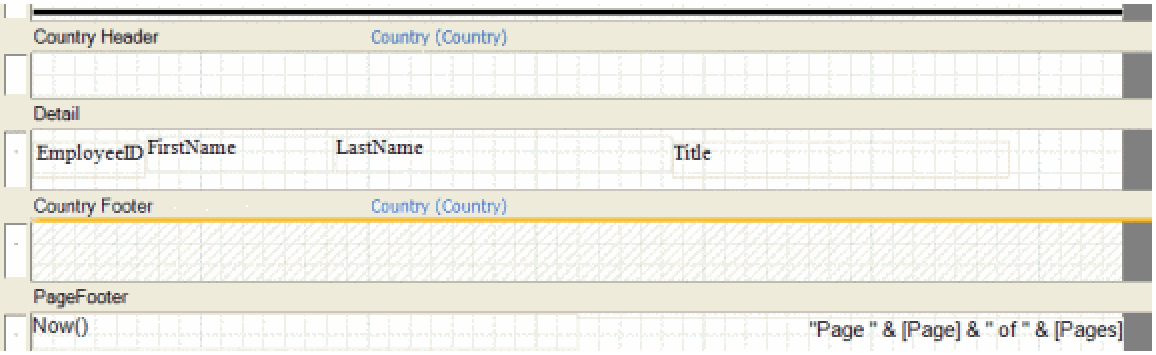
Keep Together: First Detail

☑ Header Section

☑ Footer Section

If you add more fields, you can change their order using the arrow buttons on the right of the **Groups** list. This automatically adjusts the position of the Group Header and Footer sections in the report. To delete a field, use the **Delete** button.

Once you are done arranging the fields, click **OK** to dismiss the dialog box and see the changes in the Designer. There are two new sections, a Header and Footer for the new group. Both have height zero at this point, you can expand them by dragging the edges with the mouse. Notice that the Group Header section is visible, and the footer is not. This is because the **Group Header** button in the dialog box has been checked, and the **Group Footer** button was left unchecked. Invisible sections are displayed with a hatch pattern to indicate they are invisible.



On the tan bars that mark the top of the new sections there are labels that contain the section name and the value of the group's GroupBy property.

To see how groups work, click the **Add Data Field** button , select **Country** from the menu and mark an area in the newly created Group Header Section. Click the new field to select it and change its Font property to make the new field stand out a little.

- Adding grouping and sorting using code

  Useful reports don't simply show data, they show it in an organized manner. C1Report uses *groups* to group and sort data. To demonstrate how this works, return to the Creating a Report Definition topic's code and group the employees by country.

  The following code creates a group object that sorts and groups records by country:

  - Visual Basic

```
If chkGroup.Checked Then

  ' group employees by country, in ascending order
  Dim grp As Group
  grp = c1r.Groups.Add("GrpCountry", "Country", SortEnum.Ascending)

  ' format the Header section for the new group
  With grp.SectionHeader
    .Height = 500
    .Visible = True
    f = .Fields.Add("CtlCountry", "Country", 0, 0, c1r.Layout.Width,
500)
    f.Calculated = True
    f.Align = FieldAlignEnum.LeftMiddle
    f.Font.Bold = True
    f.Font.Size = 12
    f.BorderStyle = BorderStyleEnum.Solid
    f.BorderColor = Color.FromArgb(0, 0, 150)
    f.BackStyle = BackStyleEnum.Opaque
    f.BackColor = Color.FromArgb(150, 150, 220)
    f.MarginLeft = 100
  End With

  ' sort employees by first name within each country
  c1r.Groups.Add("GrpName", "FirstName", SortEnum.Ascending)
End If
```

  - C#

```
if (chkGroup.Checked)
{
  // group employees by country, in ascending order
   Group grp = c1r.Groups.Add("GrpCountry", "Country",
SortEnum.Ascending);

  // format the Header section for the new group
  s = grp.SectionHeader;
    s.Height = 500;
    s.Visible = true;
    f = s.Fields.Add("CtlCountry", "Country", 0, 0, c1r.Layout.Width,
500);
    f.Calculated = true;
    f.Align = FieldAlignEnum.LeftMiddle;
    f.Font.Bold = true;
    f.Font.Size = 12;
    f.BorderStyle = BorderStyleEnum.Solid;
    f.BorderColor = Color.FromArgb(0, 0, 150);
    //f.BackStyle = BackStyleEnum.Opaque;
    f.BackColor = Color.Transparent;
    f.BackColor = Color.FromArgb(150, 150, 220);
    f.MarginLeft = 100;


  // sort employees by first name within each country
  c1r.Groups.Add("GrpName", "FirstName", SortEnum.Ascending);
}
```

Every group has Header and Footer sections. These are invisible by default, but the code above makes the Header section visible to show which country defines the group. Then it adds a field with the country. The new field has a solid background color.

Finally, the code adds a second group to sort the employees within each country by their first name. This group is only used for sorting, so the Header and Footer sections remain invisible.

The changes are now complete. To render the new report, you need to finish the routine with a call to the Render method. Enter the following code in the **btnEmployees_Click** event handler:

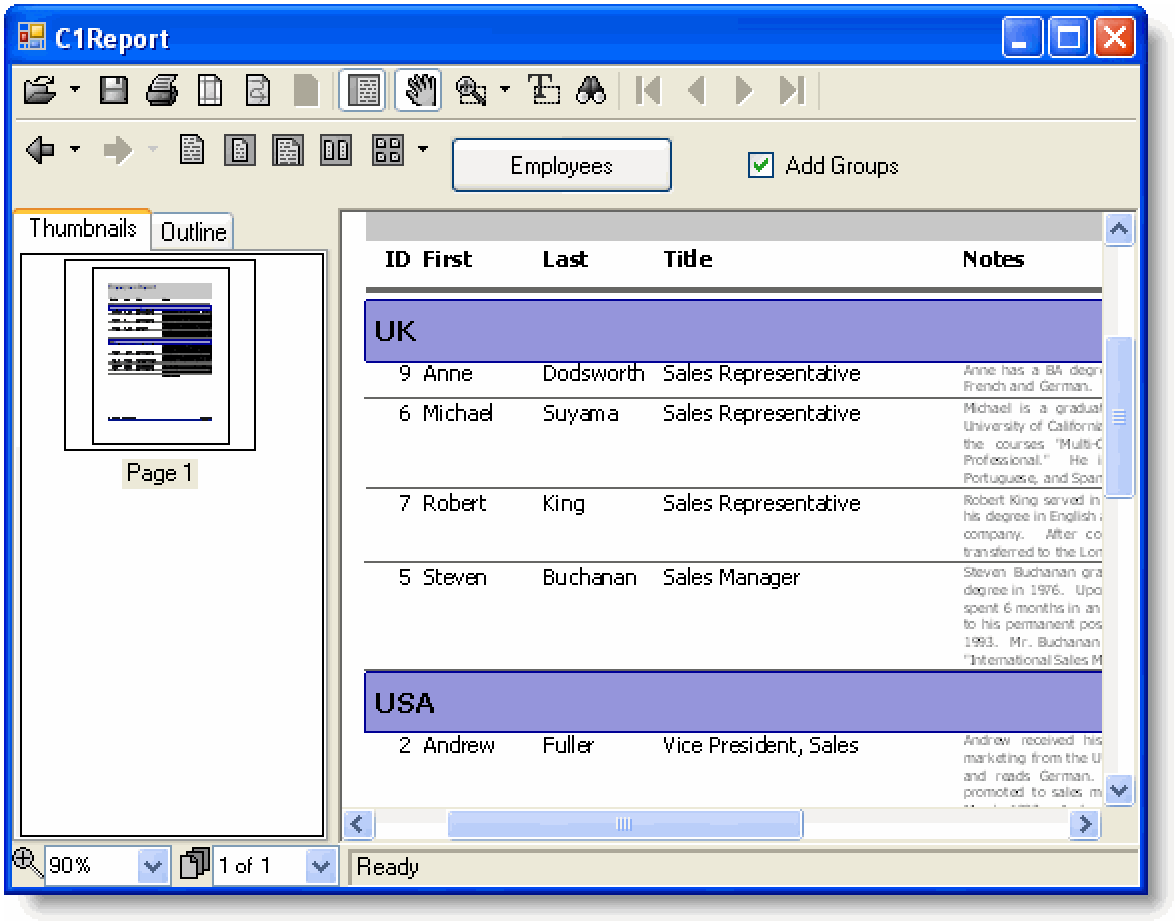
- Visual Basic

```
' render the report into the PrintPreviewControl
ppv.Document = c1r.Document
```

- C#

```
// render the report into the PrintPreviewControl
ppv.Document = c1r.Document;
```

Here's an example of a report using a group object:



| ID | First | Last | Title | Notes |
|---|---|---|---|---|
| UK | | | | |
| 9 | Anne | Dodsworth | Sales Representative | Anne has a BA degr French and German. |
| 6 | Michael | Suyama | Sales Representative | Michael is a graduat University of Californi the courses "Multi-C Professional." He i Portuguese, and Span |
| 7 | Robert | King | Sales Representative | Robert King served in his degree in English a company. After co transferred to the Lon |
| 5 | Steven | Buchanan | Sales Manager | Steven Buchanan gra degree in 1976. Upo spent 6 months in an to his permanent pos 1993. Mr. Buchanan "International Sales M |
| USA | | | | |
| 2 | Andrew | Fuller | Vice President, Sales | Andrew received his marketing from the U and reads German. promoted to sales m |

## Sorting Data

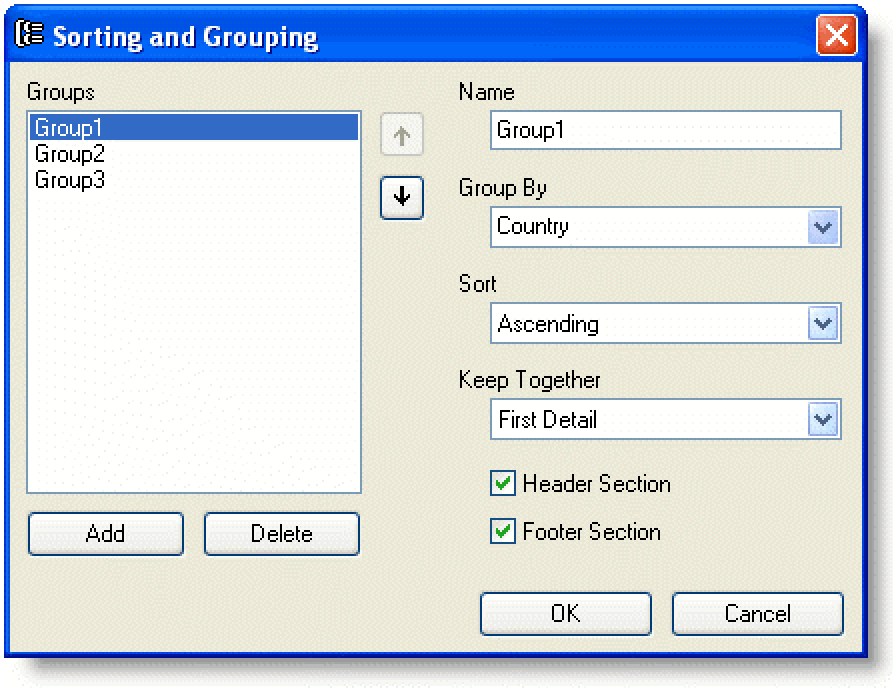
You can sort data in reports the following two ways:

- Sort the data source object itself (for example, using a SQL statement with an ORDER BY clause).
- Add groups to the report and specify how each group should be sorted using the group's GroupBy and Sort properties.

Group sorting is done using the **DataView.Sort** property, which takes a list of column names only (not expressions on column names). So if your grouping expression is `DatePart("yyyy", dateColumn)`, the control will actually sort on the dates in the *dateColumn* field, not on the years of those dates as most would expect.

To sort based on the dates, add a calculated column to the data table (by changing the SQL statement), and then group/sort on the calculated column instead. See the Sort property for an XML discussion of this, including a sample.

This is what the **Sorting and Grouping** editor looks like in the **C1ReportDesigner** application. Note the fields where you can specify group sorting:



Groups: Group1, Group2, Group3

Name: Group1

Group By: Country

Sort: Ascending

Keep Together: First Detail

☑ Header Section

☑ Footer Section

If you use both approaches, the sorting set in the report groups will prevail (it is applied after the data has been retrieved from the database).

**Sample Report Available**

For the complete report, see report "19: Sorting" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## *Adding Running Sums*

C1Report field objects have a RunningSum property that makes it easy to maintain running sums over groups or over the entire report.

### Adding Running Sums over a Group

To keep running sums over groups, complete the following tasks:

1. Open the **C1ReportDesigner** application. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner** application, you can modify the report properties.
3. Click the **Design** button to begin editing the report.
4. In Design mode, select the report from the drop-down list above the Properties window. The available properties for the report appear.
5. Add a calculated field to the report:

Click the **Add Calculated Field** button from the Designer toolbar.

In the **VBScript Editor**, enter the following script:

```
Sum(ProductSalesCtl)
```

Drag the mouse over the GroupHeader section of the report and the cursor changes into a cross-hair +. Click and drag to define the rectangle that the new field will occupy, and then release the button to create the new field.

6. Set the RunningSum property to **SumOverGroup**. (Note that for this property to appear, the properties filter must be turned off. It's the funnel icon above the Properties window.)

### Adding Running Sums over the Entire Report

To keep running sums over pages, you need to use script. For example, you could add a **pageSum** field to the report and update it with script. To do this, complete the following tasks:

1. Open the **C1ReportDesigner** application. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner** application, you can modify the report properties.
3. Click the **Design** button to begin editing the report.
4. In Design mode, select the report from the drop-down list above the Properties window. The available properties for the report appear.
5. Locate the OnPage property and click the empty field next to it, and then click the **ellipsis** button.
6. The **VBScript Editor** appears. Enter the following VBScript expression in the code editor:
```
' VBScript: Report.OnPage
pageSum = 0
```
7. Then select **Detail** from the drop-down list above the Properties window. The available properties for the Detail section appear.
8. Locate the OnPrint property and click the empty field next to it, and then click the **ellipsis** button.
9. The **VBScript Editor** appears. Enter the following VBScript expression in the code editor:
```
' VBScript: Detail.OnPrint
pageSum = pageSum + UnitsInStock
```

**Sample Report Available**

For the complete report, see report "17: Running Sums" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## *Adding Subtotals and Other Aggregates*

C1Report supports aggregate expressions in all its calculated fields. The aggregate expressions include all the usual **Sum**, **Avg**, **Min**, **Max**, **Count**, **Range**, **StDev**, and so on.

All aggregate functions take an expression as an argument and evaluate it within a scope that is determined by their position in the report. For example, aggregates in group headers or footers have the scope of the group. Aggregates in the report header or footer have the scope of the entire report.

For example, the following aggregate expression would return the sum of all values in the *Sales* field for the scope of the aggregate (group or report):

```
Sum(Sales)
```

The following aggregate expression would return the total amount of sales taxes paid for all values in the report (assuming an 8.5% sales tax):

```
Sum(Sales * 0.085)
```

You can reduce the scope of any aggregate using a second argument called *domain*. The *domain* argument is an expression that determines whether each value in the current scope should be included in the aggregate calculation.

For example, the following aggregate expression would return the sum of all values in the *Sales* field for products in category 1:

```
Sum(Sales, Category = 1)
```

The following aggregate expression would return the number of sales over $10,000:

```
Count(*, Sales > 10000)
```

**Sample Report Available**

For the complete report, see report "13: Subtotals and other Aggregates" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

### *Creating Cross-Tab Reports*

Cross-tab reports group data in two dimensions (down and across). They are useful for summarizing large amounts of data in a format that cross-references information.

To create cross-tab reports, you will typically start with a GROUP BY query to summarize the data into rows, and then use a transformation (pivot) service to create the grouped columns. The transformation service can be provided by the database server itself, it can be a custom program, or you can use **C1Report**'s built-in domain aggregates.

In all cases, the most important element in the cross-tab report is the original summarized view of the data. For example, a typical summarized view would look like this:

| Year | Quarter | Amount |
|---|---|---|
| 1990 | 1 | 1.1 |
| 1990 | 2 | 1.2 |
| 1990 | 3 | 1.3 |
| 1990 | 4 | 1.4 |
| 1991 | 1 | 2.1 |
| 1991 | 2 | 2.2 |
| 1991 | 3 | 2.3 |
| 1991 | 4 | 2.4 |

This data would then be transformed by adding columns for each quarter and consolidating the values into the new columns:

| Year | Total | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|---|
| 1990 | 5 | 1.1 | 1.2 | 1.3 | 1.4 |
| 1991 | 9 | 2.1 | 2.2 | 2.3 | 2.4 |

You can do this using C1Report aggregate functions. The report would be grouped by year. The Detail section would be invisible, and the group header would contain the following aggregates:

| **Year** | **Total** | **Q1** | **Q2** | **Q3** | **Q4** |
|---|---|---|---|---|---|
| [Year] | Sum(Amount) | Sum(Amount, Quarter=1) | Sum(Amount, Quarter=2) | Sum(Amount, Quarter=3) | Sum(Amount, Quarter=4) |

The first aggregate would calculate the total amount sold in the current year. The quarter-specific aggregates specify a domain to restrict the aggregate to the specified quarter.

**Sample Report Available**

For the complete report, see report "20: Cross-tab Reports" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Working with VBScript

VBScript expressions are widely used throughout a report definition to retrieve, calculate, display, group, sort, filter, parameterize, and format the contents of a report. Some expressions are created for you automatically (for example, when you drag a field from the Toolbox onto a section of your report, an expression that retrieves the value of that field is displayed in the text box). However, in most cases, you create your own expressions to provide more functionality to your report.

Note the following differences between VBScript expressions and statements:

- **Expressions** return values, you can assign them to things like **Field.Text**, for example:
  ```
  Field1.Calculated = true
  Field1.Text = "iif( 1=1, 1+2, 1+3 )"
  ```
- **Statements** don't return values. You can assign them to event properties like **OnFormat**. For example:
  ```
  c1r.OnOpen = "if 1=1 then msgbox("OK!!!") else msgbox("ooops")"
  ```

C1Report relies on VBScript to evaluate expressions in calculated fields and to handle report events.

VBScript is a full-featured language, and you have access to all its methods and functions when writing C1Report expressions. For the intrinsic features of the VBScript language, refer to the Microsoft Developer's Network (MSDN).

C1Report extends VBScript by exposing additional objects, variables, and functions. These extensions are described in the following sections.

### *VBScript Elements, Objects, and Variables*

The following tables detail VBScript elements, objects, and variables.

#### Operators

The following table contains the VBScript operators:

| Operator | Description |
|---|---|
| And | Performs a logical conjunction on two expressions. |
| Or | Performs a logical disjunction on two expressions. |
| Not | Performs a logical disjunction on two expressions. |
| Mod | Divides two numbers and returns only the remainder. |

#### Reserved symbols

The following table contains the VBScript reserved symbols and how to use them:

| Keyword | Description |
|---|---|
| True | The **True** keyword has a value equal to -1. |
| False | The **False** keyword has a value equal to 0. |
| Nothing | Used to disassociate an object variable from any actual object. To assign **Nothing** to an object variable, use the **Set** statement, for example:<br>`Set MyObject = Nothing`<br>Several object variables can refer to the same actual object. When **Nothing** is assigned to an object variable, that variable no longer refers to any actual object. When several object variables refer to the same object, memory and system resources associated with the object to which the variables refer are released only after all of them have been set to **Nothing**, either explicitly using **Set**, or implicitly after the last object variable set to **Nothing**. |
| Null | The **Null** keyword is used to indicate that a variable contains no valid data. |
| vbCr | When you call print and display functions, you can use the following constants in your code in place of the actual values. |
| vbCrLf | When you call print and display functions, you can use the following constants in your code in place of the actual values. |
| vbLf | When you call print and display functions, you can use the following constants in your code in place of the actual values. |
| vbFormFeed | When you call print and display functions, you can use the following constants in your code in place of the actual values. |
| vbNewLine | When you call print and display functions, you can use the following constants in your code in place of the actual values. |
| vbNullChar | When you call print and display functions, you can use the following constants in your code in place of the actual values. |
| vbTab | When you call print and display functions, you can use the following constants in your code in place of the actual values. |
| vbVerticalTab | When you call print and display functions, you can use the following constants in your code in place of the actual values. |
| vbBlack | Black. Value = 0x0. |
| vbRed | Red. Value = 0xFF. |
| vbGreen | Green. Value = 0xFF00. |
| vbYellow | Yellow. Value = 0xFFFF. |
| vbBlue | Blue. Value = 0xFF0000. |
| vbMagenta | Magenta. Value = 0xFF00FF. |
| vbCyan | Cyan. Value = 0xFFFF00. |
| vbWhite | White. Value = 0xFFFFFF. |

### Built-in functions

The VBScript built-in functions are listed below:

Abs

Date

Iif

Minute

Sign

Acos
Asc
Asin
Atn
CBool
CByte
CCur
CDate
CDbl
Chr
CInt
CLng
Cos
CSng
CStr
DateAdd
DateDiff
DatePart
DateSerial
DateValue
Day
Exp
Fix
Format
FormatCurrency
FormatDateTime
FormatNumber
FormatPercent
Hex
Hour
InputBox
InStr
InStrRev
Int
IsDate
IsEmpty
IsNull
IsNumeric
IsObject
LCase
Left
Len
Log
LTrim
Mid
Month
MonthName
MsgBox
Now
Oct
Pi
Replace
RGB
Right
Rnd
Round
RTrim
Second
Sgn
Sin
Space
Sqr
StrComp
String
Tan
Time
Timer
TimeSerial
TimeValue
Trim
TypeName
UCase
WeekDay
WeekDayName
Year

For more information on the VBScript functions, see the MSDN documentation. Note that the following VBScript features are **not** supported in C1Report:

- Arrays
- Functions/Subs
- Select/Case statements

Also note that the following C1Report features are **not** part of VBScript:

- Aggregate functions (Sum, Average, StDev, Var, Count, and so on)
- Report and Database field names
- Page/Pages variables
- Report object

### Statement keywords

The VBScript statement keywords are listed below:

If
Then
Else
ElseIf
EndIf
For
To
Next
Step
While
Wend
Const
Dim
Redim

### Report Field Names

Names of Field objects are evaluated and return a reference to the object, so you can access the field's properties. The default property for the Field object is Value, so by itself the field name returns the field's current value. For example:

- Visual Basic
```
MyField.BackColor = RGB(200,250,100)
MyField.Font.Size = 14
MyField * 2 ' (same as MyField.Value * 2)
```

- C#

```
MyField.BackColor = RGB(200,250,100);
MyField.Font.Size = 14;
MyField * 2; // (same as MyField.value * 2)
```

**Note:** If you give a report field the same name as a database field, you won't be able to access the report field.

### Report Section Names

Names of Section objects are evaluated and return a reference to the object, so you can access the section's properties. The default property for the Section object is Name. For example:

- Visual Basic

```
If Page = 1 Then [Page Footer].Visible = False
```

- C#

```
if (Page = 1)
{
    [Page Footer].Visible = false;
}
```

### Database Field Names

Names of fields in the report's dataset source are evaluated and return the current field value. If a field name contains spaces or periods, it must be enclosed in square brackets. For example:

```
OrderID
UnitsInStock
[Customer.FirstName]
[Name With Spaces]
```

### Report Variables

#### Page

The page variable returns or sets the value of the Page property. This property is initialized by the control when it starts rendering a report, and is incremented at each page break. You may reset it using code. For example:

- Visual Basic

```
If Country <> LastCountry Then Page = 1
LastCountry = Country
```

- C#

```
if (Country != LastCountry)
{
    Page = 1
}
else
{
    LastCountry = Country;
}
```

#### Pages

The pages variable returns a token that gets replaced with the total page count when the report finishes rendering. This is a read-only property that is typically used in page header or footer fields. For example:

- Visual Basic

```
"Page " & Page & " of " & Pages
```

- C#

```
"Page " + Page + " of " + Pages;
```

### Report Object

The report object returns a reference to the control object, so you can access the full C1Report object model from your scripts and expressions. For example:

- Visual Basic
```
"Fields: " & Report.Fields.Count
```

- C#
```
"Fields: " + Report.Fields.Count;
```

### Cancel

Set **Cancel** to **True** to cancel the report rendering process. For example:

- Visual Basic
```
If Page > 100 Then Cancel = True
```

- C#
```
if ( Page > 100 )
{
    Cancel = true;
}
```

## *Using Compatibility Functions: Iif and Format*

To increase compatibility with code written in Visual Basic and Microsoft Access (VBA), C1Report exposes two functions that are not available in VBScript: **Iif** and **Format**.

**Iif** evaluates a Boolean expression and returns one of two values depending on the result. For example:

```
Iif(SalesAmount > 1000, "Yes", "No")
```

**Format** converts a value into a string formatted according to instructions contained in a format expression. The value may be a number, Boolean, date, or string. The format is a string built using syntax similar to the format string used in Visual Basic or VBA.

The following table describes the syntax used for the format string:

| Value Type | Format String | Description |
|---|---|---|
| **Number** | Percent, % | Formats a number as a percentage, with zero or two decimal places.<br>For example:<br>`Format(0.33, "Percent") = "33%"`<br>`Format(0.3333333, "Percent") = "33.33%"` |
|  | #,###.##0 | Formats a number using a mask. The following symbols are recognized:<br># digit placeholder<br>0 digit placeholder, force display<br>, use thousand separators<br>( enclose negative values in parenthesis<br>% format as percentage<br>For example:<br>`Format(1234.1234, "#,###.##") = "1,234.12"`<br>`Format(-1234, "#.00") = "(1234.12)"`<br>`Format(.1234, "#.##") = ".12"`<br>`Format(.1234, "0.##") = "0.12"`<br>`Format(.3, "#.##%") = "30.00%"` |

| | | |
|---|---|---|
| **Currency** | Currency, $ | Formats a number as a currency value. Displays number with thousand separator, if appropriate; displays two digits to the right of the decimal separator.<br>For example:<br>`Format(1234, "$") = "$1,234.00"` |
| **Boolean** | Yes/No | Returns "Yes" or "No". |
| **Date** | Long Date | `Format(#12/5/1#, "long date") = "December 5, 2001"` |
| | Short Date | `Format(#12/5/1#, "short date") = "12/5/2001"` |
| | Medium Date | `Format(#12/5/1#, "medium date") = "05-Dec-01"` |
| | q,m,d,w,yyyy | Returns a date part (quarter, month, day of the month, week of the year, year).<br>For example:<br>`Format(#12/5/1#, "q") = "4"` |
| **String** | @@-@@/@@ | Formats a string using a mask. The "@" character is a placeholder for a single character (or for the whole value string if there is a single "@"). Other characters are interpreted as literals.<br>For example:<br>`Format("AC55512", "@@-@@/@@") = "AC-555/12"`<br>`Format("AC55512", "@") = "AC55512"` |
| | @;Missing | Uses the format string on the left of the semi-colon if the value is not null or an empty string, otherwise returns the part on the right of the semi-colon.<br>For example:<br>`Format("@;Missing", "UK") = "UK"`<br>`Format("@;Missing", "") = "Missing"` |

Note that VBScript has its own built-in formatting functions (**FormatNumber**, **FormatCurrency**, **FormatPercent**, **FormatDateTime**, and so on). You may use them instead of the VBA-style **Format** function described here.

## *Using Aggregate Functions*

Aggregate functions are used to summarize data over the group being rendered. When used in a report header field, these expressions return aggregates over the entire dataset. When used in group headers or footers, they return the aggregate for the group.

All C1Report aggregate functions take two arguments:

- A string containing a VBScript expression to be aggregated over the group.
- An optional string containing a VBScript expression used as a filter (domain aggregate). The filter expression is evaluated before each value is aggregated. If the filter returns **False**, the value is skipped and is not included in the aggregate result.

C1Report defines the following aggregate functions:

| Function | Description |
|---|---|
| Avg | Average value of the expression within the current group. For example, the following expressions calculate the average sales for the whole group and the average sales for a certain type of product: |

| | |
|---|---|
| | `Avg(SalesAmount)`<br>`Avg(SalesAmount, ProductType = 3)` |
| Sum | Sum of all values in the group. |
| Count | Count of records in the group with non-null values. Use an asterisk for the expression to include all records. For example, the following expressions count the number of employees with valid (non-null) addresses and the total number of employees:<br>`Count(Employees.Address)`<br>`Count(*)` |
| CountDistinct | Count of records in the group with distinct non-null values. |
| Min, Max | Minimum and maximum values for the expression.<br>For example:<br>`"Min Sale = " & Max(SaleAmount)` |
| Range | Range between minimum and maximum values for the expression. |
| StDev, Var | Standard deviation and variance of the expression in the current group. The values are calculated using the sample (n-1) formulas, as in SQL and Microsoft Excel. |
| StDevP, VarP | Standard deviation and variance of the expression in the current group. These values are calculated using the population (n) formulas, as in SQL and Microsoft Excel. |

To use the aggregate functions, add a calculated field to a Header or Footer section, and assign the expression to the field's Text property.

For example, the "Employee Sales by Country" report in the sample **NWind.xml** file contains several aggregate fields. The report groups records by Country and by Employee.

The **SalespersonTotal** field in the Footer section of the Employee group contains the following expression:

```
=Sum([SaleAmount])
```

Because the field is in the Employee group footer, the expression returns the total sales for the current employee.

The **CountryTotal** and **GrandTotal** fields contain exactly the same expression. However, because these fields are in the Country group footer and report footer, the expression returns the total sales for the current country and for the entire recordset.

You may need to refer to a higher-level aggregate from within a group. For example, in the "Employee Sales by Country" report, there is a field that shows sales in the current country as a percentage of the grand total. Since all aggregates calculated within a country group refer to the current country, the report cannot calculate this directly. Instead, the **PercentOfGrandTotal** field uses the following expression:

```
=[CountryTotal]/[GrandTotal]
```

**CountryTotal** and **GrandTotal** are fields located in the Country and Report Footer sections. Therefore, **CountryTotal** holds the total for the current country and **GrandTotal** holds the total for the whole recordset.

It is important to realize that evaluating aggregate functions is time-consuming, since it requires the control to traverse the recordset. Because of this, you should try to use aggregate functions in a few calculated fields only. Other fields can then read the aggregate value directly from these fields, rather than evaluating the aggregate expression again.

For example, the "Employee Sales by Country" report in the NorthWind database has a detail field, **PercentOfCountryTotal**, that shows each sale as a percentage of the country's total sales. This field contains the following expression:

```
=[SaleAmount]/[CountryTotal]
```

**SaleAmount** is a reference to a recordset field, which varies for each detail record. **CountryTotal** is a reference to a report field that contains an aggregate function. When the control evaluates this expression, it gets the aggregate value directly from the report field, and does not recalculate the aggregate.

**Sample Report Available**

For the complete report, see report "Employee Sales by Country" in the **Nwind.xml** report definition file, which is available for download from the **NorthWind** sample on the ComponentOne HelpCentral Sample page.

## Using Event Properties

You are not restricted to using VBScript to evaluate expressions in calculated fields. You can also specify scripts that are triggered when the report is rendered, and you can use those to change the formatting of the report.These scripts are contained in *event properties*. An event property is similar to a Visual Basic event handler, except that the scripts are executed in the scope of the report rather than in the scope of the application that is displaying the report. For example, you could use an event property to set a field's Font and ForeColor properties depending on its value. This behavior would then be a part of the report itself, and would be preserved regardless of the application used to render it.

Of course, traditional events are also available, and you should use them to implement behavior that affects the application rather than the report. For example, you could write a handler for the StartPage event to update a page count in your application, regardless of which particular report is being rendered.

The following table lists the event properties that are available and typical uses for them:

| Object | Property | Description |
|---|---|---|
| Report | OnOpen | Fired when the report starts rendering. Can be used to modify the ConnectionString or RecordSource properties, or to initialize VBScript variables. |
| | OnClose | Fired when the report finishes rendering. Can be used to perform clean-up tasks. |
| | OnNoData | Fired when a report starts rendering but the source recordset is empty. You can set the **Cancel** property to **True** to prevent the report from being generated. You could also show a dialog box to alert the user as to the reason why no report is being displayed. |
| | OnPage | Fired when a new page starts. Can be used to set the **Visible** property of sections of fields depending on a set of conditions. The control maintains a Page variable that is incremented automatically when a new page starts. |
| | OnError | Fired when an error occurs. |
| Section | OnFormat | Fired before the fields in a section are evaluated. At this point, the fields in the source recordset reflect the values that will be rendered, but the report fields do not. |
| | OnPrint | Fired before the fields in a section are printed. At this point, the fields have already been evaluated and you can do conditional formatting. |

The following topics illustrate typical uses for these properties.

## *Formatting a Field According to Its Value*

Formatting a field according to its value is probably the most common use for the OnPrint property. Take for example a report that lists order values grouped by product. Instead of using an extra field to display the quantity in stock, the report highlights products that are below the reorder level by displaying their name in bold red characters.

**To highlight products that are below the reorder level using code:**

To highlight products that are below the reorder level by displaying their name in bold red characters, use an event script that looks like this:

- Visual Basic

```
Dim script As String = _
  "If UnitsInStock < ReorderLevel Then" & vbCrLf & _
  "ProductNameCtl.ForeColor = RGB(255,0,0)" & vbCrLf & _
  "ProductNameCtl.Font.Bold = True" & vbCrLf & _
  "Else" & vbCrLf & _
  "ProductNameCtl.ForeColor = RGB(0,0,0)" & vbCrLf & _
  "ProductNameCtl.Font.Bold = False" & vbCrLf & _
  "End If"
c1r.Sections.Detail.OnPrint = script
```

- C#

```
string  script =
  "if (UnitsInStock < ReorderLevel) then\r\n" +
  "ProductNameCtl.ForeColor = rgb(255,0,0)\r\n" +
  "ProductNameCtl.Font.Bold = true\r\n" +
  "else\r\n" +
  "ProductNameCtl.ForeColor = rgb(0,0,0)\r\n" +
  "ProductNameCtl.Font.Bold = false\r\n" +
  "end if\r\n";
c1r.Sections.Detail.OnPrint = script;
```

The code builds a string containing the VBScript event handler, and then assigns it to the section's OnPrint property.

**To highlight products that are below the reorder level using the C1ReportDesigner:**

Alternatively, instead of writing the code, you can use the **C1ReportDesigner** application to type the following script code directly into the VBScript Editor of the Detail section's OnPrint property. Complete the following steps:

1. Select **Detail** from the Properties window drop-down list in the Designer. This reveals the section's available properties.

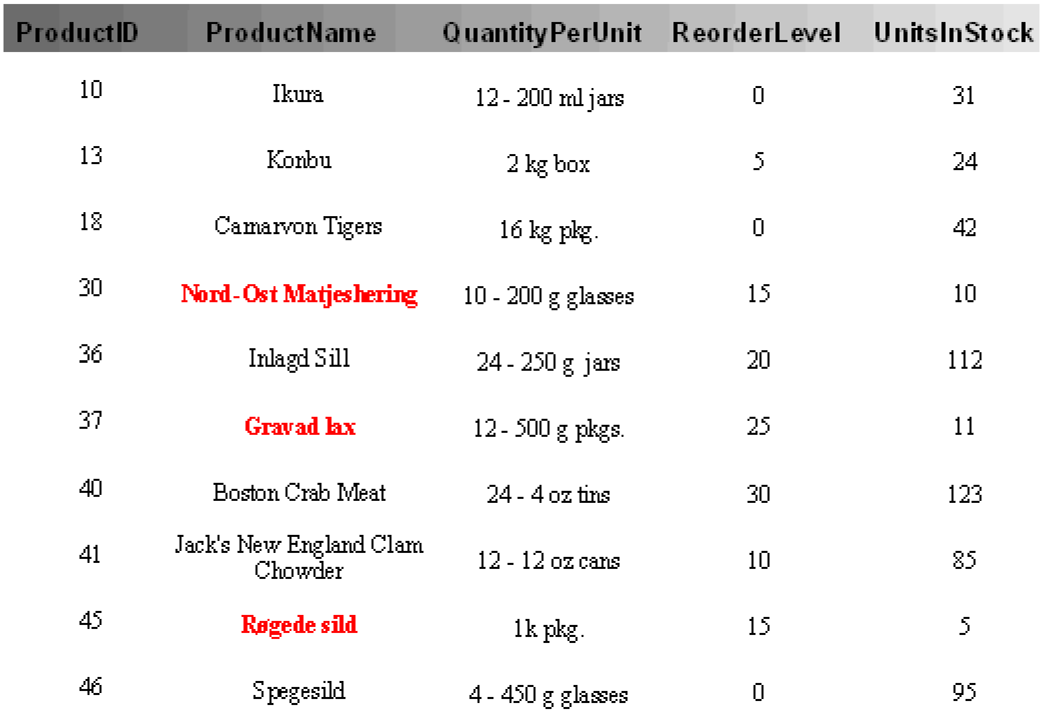
2. Click the empty box next to the OnPrint property, then click the drop-down arrow, and select **Script Editor** from the list.

3. In the **VBScript Editor**, simply type the following script in the window:

```
If UnitsInStock < ReorderLevel Then
  ProductNameCtl.ForeColor = RGB(255,0,0)
  ProductNameCtl.Font.Bold = True
Else
  ProductNameCtl.ForeColor = RGB(0,0,0)
  ProductNameCtl.Font.Bold = False
End If
```

The control executes the VBScript code whenever the section is about to be printed. The script gets the value of the "ReorderLevel" database field and sets the "ProductName" report field's Font.**Bold** and ForeColor properties according to the value. If the product is below reorder level, its name becomes bold and red.

The following screen capture shows a section of the report with the special effects:



| ProductID | ProductName | QuantityPerUnit | ReorderLevel | UnitsInStock |
|---|---|---|---|---|
| 10 | Ikura | 12 - 200 ml jars | 0 | 31 |
| 13 | Konbu | 2 kg box | 5 | 24 |
| 18 | Camarvon Tigers | 16 kg pkg. | 0 | 42 |
| 30 | Nord-Ost Matjeshering | 10 - 200 g glasses | 15 | 10 |
| 36 | Inlagd Sill | 24 - 250 g jars | 20 | 112 |
| 37 | Gravad lax | 12 - 500 g pkgs. | 25 | 11 |
| 40 | Boston Crab Meat | 24 - 4 oz tins | 30 | 123 |
| 41 | Jack's New England Clam Chowder | 12 - 12 oz cans | 10 | 85 |
| 45 | Røgede sild | 1k pkg. | 15 | 5 |
| 46 | Spegesild | 4 - 450 g glasses | 0 | 95 |

### *Hiding a Section if There is No Data for It*

You can change a report field's format based on its data by specifying an expression for the Detail section's OnFormat property.

For example, your Detail section has fields with an image control and when there is no data for that record's image you want to hide the record. To hide the Detail section when there is no data, in this case a record's image, add the following script to the Detail section's OnFormat property:

```
If isnull(PictureFieldName) Then
Detail.Visible = False
Else
Detail.Visible = True
End If
```

**To hide a section if there is no data for it using code:**

To hide a section if there is no data, in this case a record's image, for it, use an event script that looks like this:

- Visual Basic

```
c1r.Sections.Detail.OnFormat = "Detail.Visible = not
isnull(PictureFieldName)"
```

- C#

```
c1r.Sections.Detail.OnFormat = "Detail.Visible = not
isnull(PictureFieldName)";
```

**To hide a section if there is no data for it using C1ReportDesigner:**

Alternatively, instead of writing the code, you can use the **C1ReportDesigner** application to type the following script code directly into the VBScript Editor of the Detail section's OnFormat property. Complete the following steps:

1. Select **Detail** from the Properties window drop-down list in the Designer. This reveals the section's available properties.

2. Click the empty box next to the OnFormat property, then click the drop-down arrow, and select **Script Editor** from the list.

3. In the **VBScript Editor**:

   - Simply type the following script in the window:

     ```
     If isnull(PictureFieldName) Then
     Detail.Visible = False
     Else
     Detail.Visible = True
     End If
     ```

   - Or you could use the more concise version:

     ```
     Detail.Visible = not isnull(PictureFieldName)
     ```

### *Showing or Hiding a Field Depending on a Value*

Instead of changing the field format to highlight its contents, you could set another field's Visible property to **True** or **False** to create special effects. For example, if you created a new field called "BoxCtl" and formatted it to look like a bold rectangle around the product name, then you could change the script as follows:

```
If UnitsInStock < ReorderLevel Then
BoxCtl.Visible = True
Else
BoxCtl.Visible = False
End If
```

**To highlight products that are below the reorder level using code:**

To highlight products that are below the reorder level by displaying a box, use an event script that looks like this:

- Visual Basic

  ```
  Dim script As String = _
    "If UnitsInStock < ReorderLevel Then" & vbCrLf & _

    "  BoxCtl.Visible = True" & vbCrLf & _
    "Else" & vbCrLf & _
    "  BoxCtl.Visible = False" & vbCrLf & _
    "End If"
  c1r.Sections.Detail.OnPrint = script
  ```

- C#

  ```
  string script =
    "if (UnitsInStock < ReorderLevel) then\r\n" +
    "BoxCtl.Visible = true\r\n" +
    "else\r\n" +
    "BoxCtl.Visible = false\r\n" +
    "end if\r\n";
  c1r.Sections.Detail.OnPrint = script;
  ```

The code builds a string containing the VBScript event handler, and then assigns it to the section's OnPrint property.

**To highlight products that are below the reorder level using C1ReportDesigner:**

Alternatively, instead of writing the code, you can use the **C1ReportDesigner** application to type the following script code directly into the VBScript Editor of the Detail section's OnPrint property. Complete the following steps:

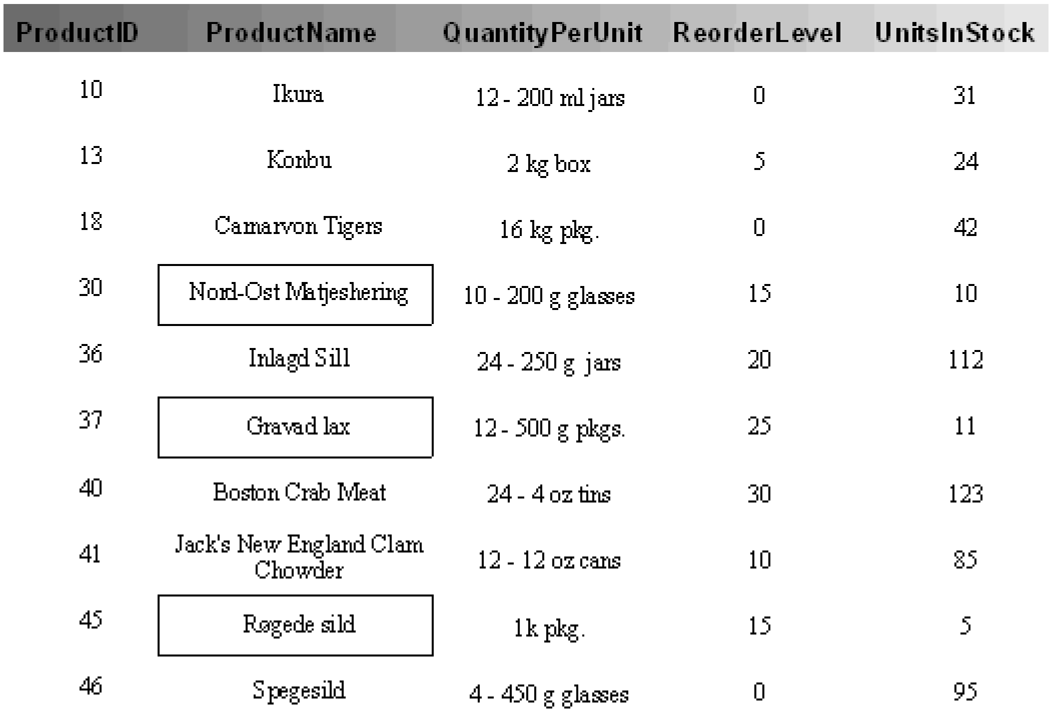
1. Select **Detail** from the Properties window drop-down list in the Designer. This reveals the section's available properties.

2. Click the empty box next to the OnPrint property, then click the drop-down arrow, and select **Script Editor** from the list.

3. In the **VBScript Editor**, simply type the following script in the window:

```
If UnitsInStock < ReorderLevel Then
BoxCtl.Visible = True
Else
BoxCtl.Visible = False
End If
```

The following screen capture shows a section of the report with the special effects:



| ProductID | ProductName | QuantityPerUnit | ReorderLevel | UnitsInStock |
|---|---|---|---|---|
| 10 | Ikura | 12 - 200 ml jars | 0 | 31 |
| 13 | Konbu | 2 kg box | 5 | 24 |
| 18 | Camarvon Tigers | 16 kg pkg. | 0 | 42 |
| 30 | Nord-Ost Matjeshering | 10 - 200 g glasses | 15 | 10 |
| 36 | Inlagd Sill | 24 - 250 g jars | 20 | 112 |
| 37 | Gravad lax | 12 - 500 g pkgs. | 25 | 11 |
| 40 | Boston Crab Meat | 24 - 4 oz tins | 30 | 123 |
| 41 | Jack's New England Clam Chowder | 12 - 12 oz cans | 10 | 85 |
| 45 | Rogede sild | 1k pkg. | 15 | 5 |
| 46 | Spegesild | 4 - 450 g glasses | 0 | 95 |

### Prompting Users for Parameters

Instead of highlighting products which are below the reorder level stored in the database, you could have the report prompt the user for the reorder level to use.

To get the value from the user, you would change the report's RecordSource property to use a parameter query. (For details on how to build parameter queries, see Parameter Queries.)

- Visual Basic

```
c1r.DataSource.RecordSource = _
  "PARAMETERS [Critical Stock Level] Short 10;" & _
  c1r.DataSource.RecordSource
```

- C#

```
c1r.DataSource.RecordSource =
"PARAMETERS [Critical Stock Level] short 10;" +
c1r.DataSource.RecordSource;
```

This setting causes the control to prompt the user for a "Critical Stock Level" value, which gets stored in a global VBScript variable where your events can use it. The default value for the variable is specified as 10.

To use the value specified by the user, the script should be changed as follows:

- Visual Basic

```
Dim script As String = _
```

```
  "level = [Critical Stock Level]" & vbCrLf & _
  "If UnitsInStock < level Then" & vbCrLf & _
  "  ProductNameCtl.ForeColor = RGB(255,0,0)" & vbCrLf & _
  "  ProductNameCtl.Font.Bold = True" & vbCrLf & _
  "Else" & vbCrLf & _
  "  ProductNameCtl.ForeColor = RGB(0,0,0)" & vbCrLf & _
  "  ProductNameCtl.Font.Bold = False" & vbCrLf & _
  "End If"
c1r.Sections("Detail").OnPrint = script
```

- C#

```
string  script =
  "level = [Critical Stock Level]\r\n" +
  "if (UnitsInStock < level) then\r\n" +
  "ProductNameCtl.ForeColor = rgb(255,0,0)\r\n" +
  "ProductNameCtl.Font.Bold = true\r\n" +
  "else\r\n" +
  "ProductNameCtl.ForeColor = rgb(0,0,0)\r\n" +
  "ProductNameCtl.Font.Bold = false\r\n" +
  "end if\r\n";
c1r.Sections.Detail.OnPrint = script;
```

The change is in the first two lines of the script. Instead of comparing the current value of the "UnitsInStock" filed to the reorder level stored in the database, the script compares it to the value entered by the user and stored in the "[Critical Stock Level]" VBScript variable.

## *Resetting the Page Counter*

The Page variable is created and automatically updated by the control. It is useful for adding page numbers to page headers or footers.

**To reset the page counter when a group starts:**

In some cases, you may want to reset the page counter when a group starts. For example, in a report that groups records by country and has a calculated page footer field with the expression:

```
=[Country] & " - Page " & [Page]
```

**Using Code:**

To reset the page counter when a group (for example, a new country) starts, set the PageFooter field's **Text** property. Enter the following code:

- Visual Basic

```
c1r.Fields("PageFooter").Text = "[Country] & "" "" & [Page]"
```

- C#

```
c1r.Fields("PageFooter").Text = "[Country] + "" "" + [Page]";
```

**Using C1ReportDesigner:**

To reset the page counter when a group (for example, a new country) starts, set the PageFooter field's **Text** property by completing the following steps:

1. Select the PageFooter's **PageNumber** field from the Properties window drop-down list in the Designer. This reveals the field's available properties.

2. Click the empty box next to the **Text** property, then click the drop-down arrow, and select **Script Editor** from the list.

3. In the **VBScript Editor**, simply type the following script in the window:

```
=[Country] & " - Page " & [Page]
```

**To reset the Page variable for each new country:**

It would be good to reset the Page variable for each new country. To do this, assign the following script to the Country Group Header section:

```
Page = 1
```

**Using Code:**

To reset the Page variable for each new country, set the Country Group Header OnPrint property. Enter the following code:

- Visual Basic

```
c1r.Sections("CountryGroupHeader").OnPrint = "Page = 1"
```

- C#

```
c1r.Sections("CountryGroupHeader").OnPrint = "Page = 1";
```

**Using C1ReportDesigner:**

To reset the Page variable for each new country, set the Country Group Header OnPrint property by completing the following steps:

1. Select the Country Group Header from the Properties window drop-down list in the Designer. This reveals the section's available properties.
2. Click the empty box next to the OnPrint property, then click the drop-down arrow, and select **Script Editor** from the list.
3. In the **VBScript Editor**, simply type the following script in the window:

```
Page = 1
```

### *Changing a Field's Dimensions to Create a Bar Chart*

This is the most sophisticated example. Instead of showing a field's value as text, you can change its dimensions to create a chart.

**Create the Chart**

To create a chart, the first thing you need to do is find out the scale, that is, the measurements that will be used to the maximum and minimum values. The "Sales Chart" report has a field designed to do this. It is a report footer field called **SaleAmountMaxFld** that has the size of the longest bar you want to appear on the chart, and holds the following expression:

```
=Max([SaleAmount])
```

**Using Code:**

To set the maximum value for the chart, set the **SaleAmountMaxFld.Text** property. Enter the following code:

- Visual Basic

```
SaleAmountMaxFld.Text = "Max([SaleAmount])"
```

- C#

```
SaleAmountMaxFld.Text = "Max([SaleAmount])";
```

**Using C1ReportDesigner:**

To set the maximum value for the chart, set the **SaleAmountMaxFld.Text** property by completing the following steps:

1. Select the **SaleAmountMaxFld** from the Properties window drop-down list in the Designer. This reveals the field's available properties.
2. Click the empty box next to the **Text** property, then click the drop-down arrow, and select **Script Editor** from the list.
3. In the **VBScript Editor**, simply type the following script in the window:
   ```
   =Max([SaleAmount])
   ```

### Draw the Chart's Bars

To draw the actual bars, the report has a detail field called **BarFld** that is formatted to look like a solid box. The Detail section has the following script assigned to its OnPrint property:

```
BarFld.Width = SaleAmountMaxFld.Width * (SaleAmountFld / SaleAmountMaxFld)
```

This expression calculates the width of the bar based on the width and value of the reference field and on the value of the **SaleAmountFld** for the current record.

#### Using Code:

To draw the actual bars for the chart, set the OnPrint property. Enter the following code:

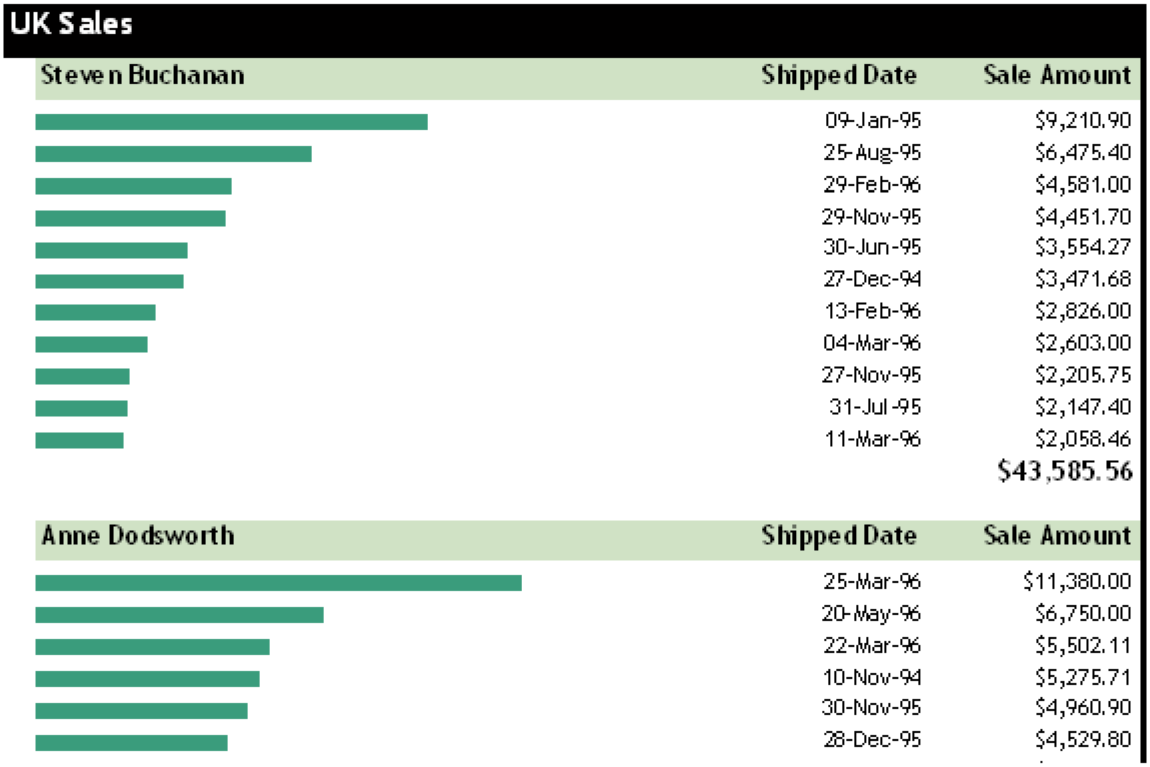
- Visual Basic
  ```
  c1r.Sections.Detail.OnPrint = & _
    "BarFld.Width = SaleAmountMaxFld.Width * " & _
    "(SaleAmountFld / SaleAmountMaxFld)"
  ```
- C#
  ```
  c1r.Sections.Detail.OnPrint = +
    "BarFld.Width = SaleAmountMaxFld.Width * " +
    "(SaleAmountFld / SaleAmountMaxFld)";
  ```

#### Using C1ReportDesigner:

To draw the actual bars for the chart, set the OnPrint property by completing the following steps:

1. Select **Detail** from the Properties window drop-down list in the Designer. This reveals the Detail section's available properties.
2. Click the empty box next to the OnPrint property, then click the drop-down arrow, and select **Script Editor** from the list.
3. In the **VBScript Editor**, simply type the following script in the window:
   ```
   BarFld.Width = SaleAmountMaxFld.Width * (SaleAmountFld / SaleAmountMaxFld)
   ```

The following screen capture shows a section of the "Sales Chart" report with the special effects:



UK Sales

| Steven Buchanan | Shipped Date | Sale Amount |
|---|---|---|
| | 09-Jan-95 | $9,210.90 |
| | 25-Aug-95 | $6,475.40 |
| | 29-Feb-96 | $4,581.00 |
| | 29-Nov-95 | $4,451.70 |
| | 30-Jun-95 | $3,554.27 |
| | 27-Dec-94 | $3,471.68 |
| | 13-Feb-96 | $2,826.00 |
| | 04-Mar-96 | $2,603.00 |
| | 27-Nov-95 | $2,205.75 |
| | 31-Jul-95 | $2,147.40 |
| | 11-Mar-96 | $2,058.46 |
| | | $43,585.56 |

| Anne Dodsworth | Shipped Date | Sale Amount |
|---|---|---|
| | 25-Mar-96 | $11,380.00 |
| | 20-May-96 | $6,750.00 |
| | 22-Mar-96 | $5,502.11 |
| | 10-Nov-94 | $5,275.71 |
| | 30-Nov-95 | $4,960.90 |
| | 28-Dec-95 | $4,529.80 |

**Sample Report Available**

For the complete report, see report "Sales Chart" in the **Nwind.xml** report definition file, which is available for download from the **NorthWind** sample on the ComponentOne HelpCentral Sample page.

## Advanced Uses

This section describes how you can add parameter queries, create unbound reports, use custom data sources, and add data security to your reports.

### *Parameter Queries*

A parameter query is a query that displays its own dialog box prompting the user for information, such as criteria for retrieving records or a value for a report field. You can design the query to prompt the user for more than one piece of information; for example, you can design it to retrieve two dates. C1Report then retrieves all records that fall between those two dates.

You can also create a monthly earnings report based on a parameter query. When you render the report, C1Report displays a dialog box asking for the month that you want the report to cover. You enter a month and C1Report prints the appropriate report.

To create a parameter query, you need to edit the SQL statement in the RecordSource property and add a PARAMETERS clause to it. The syntax used to create parameter queries is the same as that used by Microsoft Access.

1. The easiest way to create a parameter query is to start with a plain SQL statement with one or more items in the WHERE clause, then manually replace the fixed values in the clause with parameters. For example, starting with the plain SQL statement:

```
strSQL = "SELECT DISTINCTROW * FROM Employees " & _
        "INNER JOIN (Orders INNER JOIN [Order Subtotals] " & _
        "ON Orders.OrderID = [Order Subtotals].OrderID) " & _
        "ON Employees.EmployeeID = Orders.EmployeeID " & _
        "WHERE (((Orders.ShippedDate) " & _
        "Between #1/1/1994# And #1/1/2001#));"
```

2. The next step is to identify the parts of the SQL statement that will be turned into parameters. In this example, the parameters are the dates in the WHERE clause, shown above in boldface. Let's call these parameters *Beginning Date* and *Ending Date*. Since these names contain spaces, they need to be enclosed in square brackets:

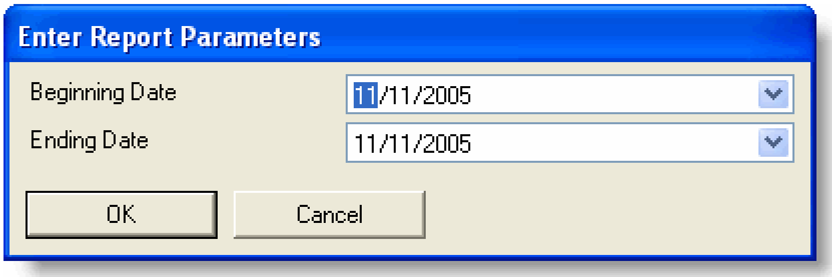
```
strSQL = "SELECT DISTINCTROW * FROM Employees " & _
        "INNER JOIN (Orders INNER JOIN [Order Subtotals] " & _
        "ON Orders.OrderID = [Order Subtotals].OrderID) " & _
        "ON Employees.EmployeeID = Orders.EmployeeID " & _
        "WHERE (((Orders.ShippedDate) " & _
        "Between [Beginning Date] And [Ending Date]));"
```

3. Finally, the parameters must be identified in the beginning of the SQL statement with a PARAMETERS clause, that includes the parameter name, type, and default value:

```
strSQL = "PARAMETERS [Beginning Date] DateTime 1/1/1994, " & _
        "[Ending Date] DateTime 1/1/2001;" & _
        "SELECT DISTINCTROW * FROM Employees " & _
        "INNER JOIN (Orders INNER JOIN [Order Subtotals] " & _
        "ON Orders.OrderID = [Order Subtotals].OrderID) " & _
        "ON Employees.EmployeeID = Orders.EmployeeID " & _
        "WHERE (((Orders.ShippedDate) " & _
        "Between [Beginning Date] And [Ending Date]));"
```

When this statement is executed, the control shows a dialog box prompting the user for *Beginning Date* and *Ending Date* values. The values supplied by the user are plugged into the SQL statement and the report is generated only for the period selected by the user.

The dialog box is created on-the-fly by C1Report. It includes all parameters in the query, and uses controls appropriate to the parameter type. For example, check boxes are used for Boolean parameters, and date-time picker controls are used for Date parameters. Here is what the dialog box looks like for the SQL statement listed above:



Enter Report Parameters

Beginning Date 11/11/2005

Ending Date 11/11/2005

The syntax for the PARAMETERS clause consists of a comma-separated list of items, ending with a semi-colon. Each item describes one parameter and includes the following information:

- **Parameter name**. If the name contains spaces, it must be enclosed in square brackets (for example, [Beginning Date]). The parameter name appears in the dialog box used to collect the information from the user, and it also appears in the WHERE clause for the SQL statement, where it is replaced with the value entered by the user.
- **Parameter type.** The following types are recognized by the control:

| Type Name | ADO Type |
|---|---|

| | |
|---|---|
| Date | adDate |
| DateTime | adDate |
| Bit, Byte, Short, Long | adInteger |
| Currency | adCurrency |
| Single | adSingle |
| Double | adDouble |
| Text, String | adBSTR |
| Boolean, Bool, YesNo | adBoolean |

- **Default value**. This is the value initially displayed in the dialog box.

The easiest way to build a parameterized query is incrementally. Start with a simple query that works, then add the PARAMETERS clause (don't forget to end the PARAMETERS clause with a semi-colon). Finally, edit the WHERE clause and add the parameter names at the proper place.

You can use the GetRecordSource method to retrieve a proper SQL statement (without the PARAMETERS clause) from a parameter query. This is useful if you want to create your own recordset using data contained in the report.

**Note:** Instead of using a parameter query, you could write code in Visual Basic or in C# to create a dialog box, get the information from the user, and fix the SQL statement or set the DataSource object's Filter property as needed. The advantage of using parameter queries is that it places the parameter logic in the report itself, and it is independent of the viewer application. (It also saves you from writing some code.)

## *Unbound Reports*

Unbound reports are reports without an underlying source recordset. This type of report can be useful in two situations:

- You create documents that have a fixed format, and only the content of a few fields that change every time you need to render the document. Business forms are a typical example: the forms have a fixed format, and the field values change.
- You want to consolidate several summary reports into a single one. In this case, you would create an unbound main report, and you would add bound subreports to it.

As an example of a simple unbound report, let's create a simple newsletter without a source recordset. This is done with the **C1ReportDesigner** application, except that you leave the ConnectionString and RecordSource properties blank and add placeholder fields to the report. The placeholder fields are simple labels whose contents will be set by an application.

Assuming you created a report with six placeholder fields named "FldHeadlineXXX" and "FldBodyXXX" (where XXX ranges from 1 to 3), you could use the following code to render the report:

- Visual Basic

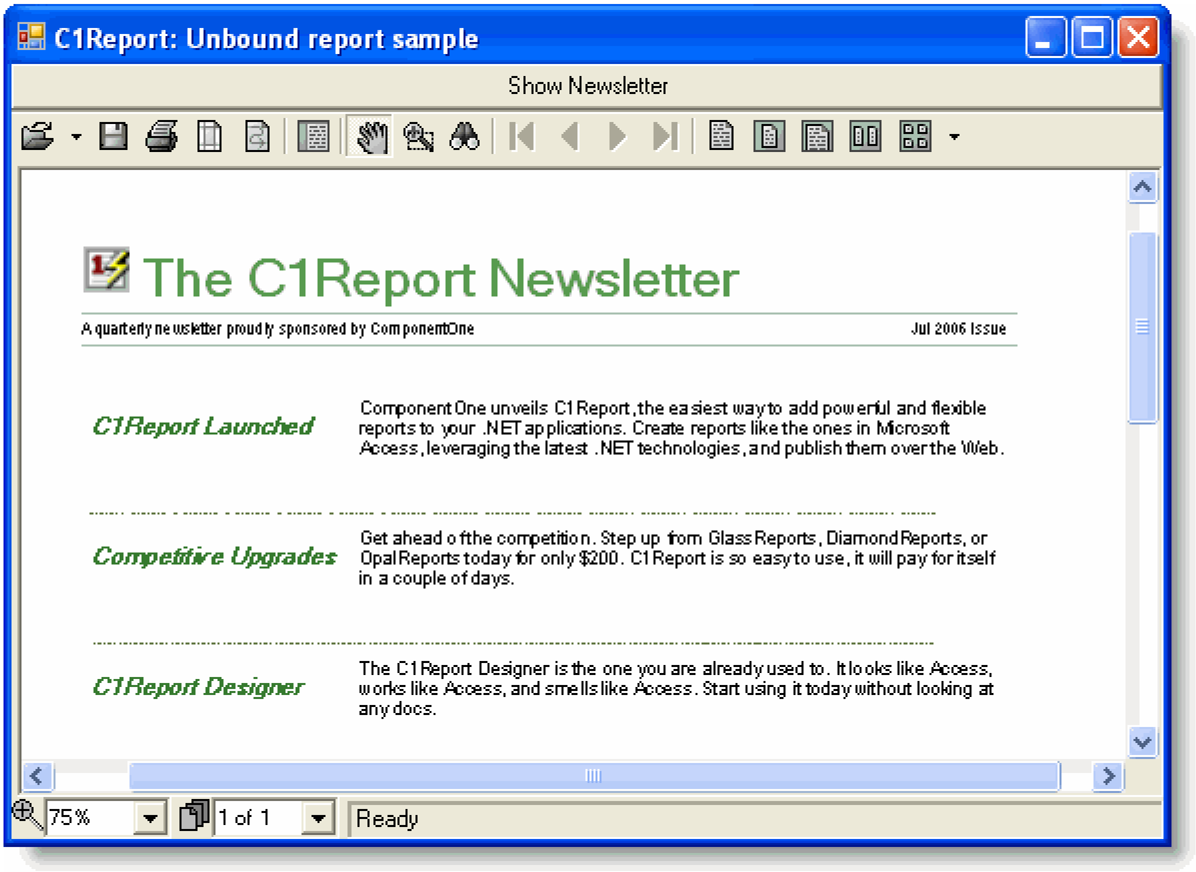
```
Private Sub MakeReport()

  ' find report definition file
  Dim path As String = Application.StartupPath
  Dim i As Integer = path.IndexOf("\bin")
  If i > -1 Then path = path.Substring(0, i)
  path = path & "\"
  ' load unbound report
  c1r.Load(path & "Newsletter.xml", "NewsLetter")

  ' set field values
  c1r.Fields("FldHeadline1").Text = "C1Report Launched"
  c1r.Fields("FldBody1").Text = "ComponentOne unveils…"
  c1r.Fields("FldHeadline2").Text = "Competitive Upgrades"
  c1r.Fields("FldBody2").Text = "Get ahead …"
  c1r.Fields("FldHeadline3").Text = "C1Report Designer"
  c1r.Fields("FldBody3").Text = "The C1Report Designer..."

  ' done, show the report
  c1ppv.Document = c1r.Document

  ' and/or save it to an HTML document so your subscribers
  ' can get to it over the Web
  c1r.RenderToFile(path & "Newsletter.htm", FileFormatEnum.HTML)
End Sub
```

- C#

```
private void MakeReport()
{

  // find report definition file
   string path = Application.StartupPath;
   int i = path.IndexOf("\bin");
  if ( i > -1 ) { path = path.Substring(0, i)
  path = path + "\";

  // load unbound report
  c1r.Load(path + "Newsletter.xml", "NewsLetter");

  // set field values
  c1r.Fields["FldHeadline1"].Text = "C1Report Launched";
  c1r.Fields["FldBody1"].Text = "ComponentOne unveils…";
  c1r.Fields["FldHeadline2"].Text = "Competitive Upgrades";
  c1r.Fields["FldBody2"].Text = "get { ahead …";
  c1r.Fields["FldHeadline3"].Text = "C1Report Designer";
  c1r.Fields["FldBody3"].Text = "The C1Report Designer...";

  // done, show the report
  c1ppv.Document = c1r.Document;

  // and/or save it to an HTML document so your subscribers
  // can get to it over the Web
  c1r.RenderToFile(path + "Newsletter.htm", FileFormatEnum.HTML);
}
```

Here's what this issue of ComponentOne's newsletter looks like. Notice that our simple program does not deal with any formatting at all; it simply supplies the report contents. The report definition created with the **C1ReportDesigner** application takes care of all the formatting, including a headline with a logo, page footers, fonts and text positioning.

Separating format from content is one of the main advantages of unbound reports.



The C1Report Newsletter

A quarterly newsletter proudly sponsored by ComponentOne Jul 2006 Issue

*C1Report Launched* ComponentOne unveils C1Report, the easiest way to add powerful and flexible reports to your .NET applications. Create reports like the ones in Microsoft Access, leveraging the latest .NET technologies, and publish them over the Web.

*Competitive Upgrades* Get ahead of the competition. Step up from GlassReports, DiamondReports, or OpalReports today for only $200. C1Report is so easy to use, it will pay for itself in a couple of days.

*C1Report Designer* The C1Report Designer is the one you are already used to. It looks like Access, works like Access, and smells like Access. Start using it today without looking at any docs.

### *Custom Data Sources*

By default, C1Report uses the ConnectionString and RecordSource properties to create an internal **DataTable** object that is used as a data source for the report. However, you can also create your own recordsets and assign them directly to the Recordset property. In this case, C1Report uses the recordset provided instead of opening its own.

You can assign three types of objects to the Recordset property: **DataTable**, **DataView**, or any object that implements the IC1ReportRecordset interface.

## Using Your Own DataTable Objects

The main reason to use your own **DataTable** objects is in situations where you already have the object available, and want to save some time by not creating a new one. You may also want to implement security schemes or customize the object in some other way.

To use your own **DataTable** object, simply assign it to the RecordSet property before you render the report. For example:

- Visual Basic

```
Private Sub CreateReport(strSelect As String, strConn As String)

  ' fill a DataSet object
  Dim da As OleDbDataAdapter
  da = new OleDbDataAdapter(strSelect, strConn)
  Dim ds As DataSet = new DataSet()
  da.Fill(ds)
  ' get the DataTable object
  Dim dt As DataTable = ds.Tables(0)

  ' load report
  c1r.Load("RepDef.xml", "My Report")

  ' render report
  c1r.DataSource.Recordset = ds.Tables(0)
  c1ppv.Document = c1r.Document

End Sub
```

- C#

```
private void CreateReport(string  strSelect, string  strConn)
{

  // fill a DataSet object
   OleDbDataAdapter da;
  da = new OleDbDataAdapter(strSelect, strConn);
  DataSet  DataSet ds = new DataSet();
  da.Fill(ds);

  // get the DataTable object
   DataTable dt = ds.Tables[0];

  // load report
  c1r.Load("RepDef.xml", "My Report");

  // render report
  c1r.DataSource.Recordset = ds.Tables[0];
  c1ppv.Document = c1r.Document;

}
```

The code above creates a **DataTable** object using standard ADO.NET calls, and then assigns the table to the Recordset property. Note that you could also create and populate the **DataTable** object on the fly, without relying on and actual database.

## Writing Your Own Custom Recordset Object

For the ultimate in data source customization, you can implement your own data source object. This option is indicated in situations where:

- Your data is already loaded in memory.
- Some or all of the data is calculated on demand, and does not even exist until you request it.
- The data comes from disparate data sources and you don't have an easy way to create a standard DataTable object from it.

To implement your own data source object, you need to create an object that implements the IC1ReportRecordset interface. This interface contains a few simple methods described in the reference section of this document.

After you have created the custom data source object, all you need to do is create an instance of it and assign that to the Recordset property.

**Sample Report Available**

For the complete project, see the **CustomData** sample, which is available for download from the ComponentOne HelpCentral Sample page.

### *Data Security*

Data security is an important issue for most corporations. If you plan to create and distribute a phone-directory report for your company, you want to show employee names and phone extensions. You probably don't want people to change the report definition and create a report that includes people's salaries. Another concern is that someone could look at the report definition file, copy a connection string and start browsing (or hacking) your database.

These are legitimate concerns that affect all types of data-based applications, including C1Report. This section discusses some measures you can take to protect your data.

## Using Windows NT Integrated Security

One of the strengths of Windows NT is the way it handles security. When a user logs on, the system automatically gives him a set of permissions granted by the system administrator. After this, each application or service can query Windows NT to see what resources he can access. Most popular database providers offer this type of security as an option.

Under this type of scenario, all you need to do is make sure that the people with whom you want to share your data have the appropriate permissions to read it. In this case, the ConnectionString in the report definition file doesn't need to contain any passwords. Authorized users get to see the data and others do not.

## Building a ConnectionString with a User-Supplied Password

Building a connection string with a user-supplied password is a very simple alternative to protect your data. For example, before rendering a report (or when the control reports a "failed to connect" error), you can prompt the user for a password and plug that into the connection string:

- Visual Basic

```
' build connection string with placeholder for password
Dim strConn
strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
          "Data Source=C:\SecureData\People.mdb;" & _
          "Password={{THEPASSWORD}};"

' get password from the user
Dim strPwd$
strPwd = InputBox("Please enter your password:")
If Len(strPwd) = 0 Then Exit Sub

' build new connection string and assign it to the control
strConn = Replace(strConn, "{{THEPASSWORD}}", strPwd)
vsr.DataSource.ConnectionString = strConn
```

- C#

```
// build connection string with placeholder for password
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" +
          "Data Source=C:\SecureData\People.mdb;" +
          "Password={{THEPASSWORD}};";

// get password from the user
string strPwd = InputBox("Please enter your password:");
if (strPwd.Length == 0) return;
```

```
// build new connection string and assign it to the control
strConn = Replace(strConn, "{{THEPASSWORD}}", strPwd);
c1r.DataSource.ConnectionString = strConn;
```

## Creating Application-Defined Aliases

Another possible scenario is one where you want to allow certain users to see the reports, but you don't want to give them any special authorizations or information about where the data is stored.

There are two simple ways to achieve this with C1Report. One is by using embedded reports. Load the report definition into your application at design time, using the **Load Report** dialog box, and the report will be embedded in the application. This way, you don't have to distribute a report definition file and no one will have access to the data source information.

The second way would be for your application to define a set of connection string aliases. The report definition file would contain the alias, and your application would replace it with the actual connection string before rendering the reports. The alias would be useless in any other applications (such as **C1ReportDesigner**). Depending on how concerned you are with security, you could also perform checks on the RecordSource property to make sure no one is trying to get unauthorized access to certain tables or fields in the database.

The following code shows how you might implement a simple alias scheme:

- Visual Basic

```
Private Sub RenderReport(strReportName As String)

  ' load report requested by the user
  c1r.Load("c:\Reports\MyReports.xml", strReportName)

  ' replace connection string alias
  Dim strConn$
  Select Case c1r.DataSource.ConnectionString

    Case "$$CUSTOMERS"
    Case "$$EMPLOYEES"
       strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
                 "Data Source=C:\SecureData\People.mdb;" & _
                 "Password=slekrsldksd;"
    Case "$$SALES"
      strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
                "Data Source=C:\SecureData\Numbers.mdb;" & _
                "Password=slkkdmssids;"
  End Select

  ' set connection string, render report
  c1r.DataSource.ConnectionString = strConn
  ppv1.Document = c1r.Document

End Sub
```

- C#

```
private void RenderReport(string  strReportName) {

  // load report requested by the user
  c1r.Load("c:\Reports\MyReports.xml", strReportName);

  // replace connection string alias
  string strConn$;
  switch (i) { c1r.DataSource.ConnectionString;
```

```
    case "$$CUSTOMERS";
    case "$$EMPLOYEES";
      strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" + _
                "Data Source=C:\SecureData\People.mdb;" +
         "Password=slekrsldksd;";
    case "$$SALES";
      strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" +
             "Data Source=C:\SecureData\Numbers.mdb;" +
                "Password=slkkdmssids;";
  }

  // set connection string, render report
  c1r.DataSource.ConnectionString = strConn;
  ppv1.Document = c1r.Document;
}
```

## Creating Custom Datasets

You can also assign an arbitrary dataset created by your application to the Recordset object. This way, you can adopt whatever security measures you see fit, and you don't need to bother with the ConnectionString and RecordSource properties at all.

# Working with C1RdlReport

The C1RdlReport component, a component that represents an RDL (Report Definition Language) report defined using the 2008 version of the RDL specification. The C1RdlReport component is similar to the C1Report component with the addition of RDL support. The C1RdlReport component is located in the C1.C1Report.2.dll assembly.

RDL import in C1PrintDocument (provided by **ImportRdl** and **FromRdl** methods) is now obsolete. C1RdlReport should be used instead.

**Important Notes for .NET 2.0/3.0 version users:**

The C1.C1Report.2 assembly uses the **System.Windows.Forms.DataVisualization.dll** assembly that is only included in .NET Framework 3.5 and later. The assembly is installed on your system with when you install **Reports for WinForms**. It MUST BE INCLUDED with other ComponentOne Reports assemblies when you deploy your application that using **Reports for WinForms** to other systems.

Also, if you update the **Reports for WinForms** DLLs manually, you MUST put **DataVisualization** where those assemblies are, and make sure that your project references it. This does not apply to .NET 4.0 users as **DataVisualization** is already included in the Framework.

## Report Definition Language (RDL)

Report Definition Language (RDL) is a Microsoft-standard XML schema for representing reports. The goal of RDL is to promote the interoperability of reporting products by defining a common schema that allows interchange of report definitions. RDL is designed to be output format neutral. This means that reports defined using RDL should be able to be outputted to a variety of formats including Web and print-ready formats or data-focused formats like XML.

The C1RdlReport component was added to **Reports for WinForms** in the 2010 v3 release to leverage the flexibility of RDL files. Using C1RdlReport, you can easily import RDL files into your reporting applications – adding flexibility and functionality through the use of the familiar format.

## C1RdlReport Advantages

The C1RdlReport control has several advantages over using the standard Microsoft Reporting Services. The major advantages provided by C1RdlReport include:

- Support for the current RDL 2008 specification.
- Programmatic access to the RDL object model (which follows the 2008 RDL specification) - this allows you to modify existing or create new RDL reports completely in code.
- Generation of RDL reports that can consume any data source (such as .mdb files).
- A self-contained RDL reporting solution without external dependencies such as the need to access a Microsoft Reporting Services server.
- Seamless integration with C1PrintPreviewControl and with other **Reports for WinForms** reporting engines including C1Report and C1PrintDocument.

## C1RdlReport Limitations

The C1RdlReport control supports most of the features of RDL. However, the initial release of C1RdlReport includes some limitations.

The following objects are not supported:

- Gauge objects are not supported.

The following RDL properties are not currently supported:

- Document.AutoRefresh
- Document.Width
- Document.Language
- Document.DataTransform
- Document.DataSchema
- Document.DataElementName
- Document.DataElementStyle
- DataSet.CaseSensitivity
- DataSet.Collation
- DataSet.AccentSensitivity
- DataSet.KanatypeSensitivity
- DataSet.WidthSensitivity
- DataSet.InterpretSubtotalsAsDetails
- TextBox.UserSort
- TextBox.ListStyle
- TextBox.ListLevel
- TextRun.ToolTip
- TextRun.MarkupType

**Important Note for .NET 2.0/3.0 version users:**

The C1.C1Report.2 assembly uses the **System.Windows.Forms.DataVisualization.dll** assembly that is only included in .NET Framework 3.5 and later. The assembly is installed on your system with when you install **Reports for WinForms**. It MUST BE INCLUDED with other ComponentOne Reports assemblies when you deploy your application that using **Reports for WinForms** to other systems.

Also, if you update the **Reports for WinForms** DLLs manually, you MUST put **DataVisualization** where those assemblies are, and make sure that your project references it. This does not apply to .NET 4.0 users as **DataVisualization** is already included in the Framework.

## Loading an RDL file

To load an RDL file into the C1RdlReport component you can use the Load method. To remove an RDL file, you would use the Clear method. This method clears any RDL file previously loaded into the C1RdlReport control. The C1RdlReport component includes design-time options to load and clear an RDL file.

**To load an RDL file in the Tasks menu:**

Complete the following steps:

1. In Design View, click the C1RdlReport component's smart tag to open the **C1RdlReport Tasks** menu.
2. In the **C1RdlReport Tasks** menu choose **Load Report**. The **Open** dialog will appear.
3. In the **Open** dialog box, locate and select an RDL file and then click **Open**.

The report will be loaded and if the C1RdlReport control is connected to a previewing control, such as C1PrintPreviewControl, the report will appear previewed in the previewing control at design time.

**To load an RDL file in code:**

To load an RDL file into the C1RdlReport component you can use the Load method. Complete the following steps:

1. In Design View, double-click on the form to open the Code Editor.
2. Add the following code to the **Load** event, replacing "C:/Report.rdl" with the location and name of the RDL file you want to load.

   - Visual Basic

     ```
     C1RdlReport1.Load("C:/Report.rdl")
     C1RdlReport1.Render()
     ```

   - C#

     ```
     c1RdlReport1.Load(@"C:/Report.rdl");
     c1RdlReport1.Render();
     ```

   The report will be loaded and if the C1RdlReport control is connected to a previewing control, such as C1PrintPreviewControl, when you run the application, the report will appear in the previewing control.

# Working with C1ReportDesigner

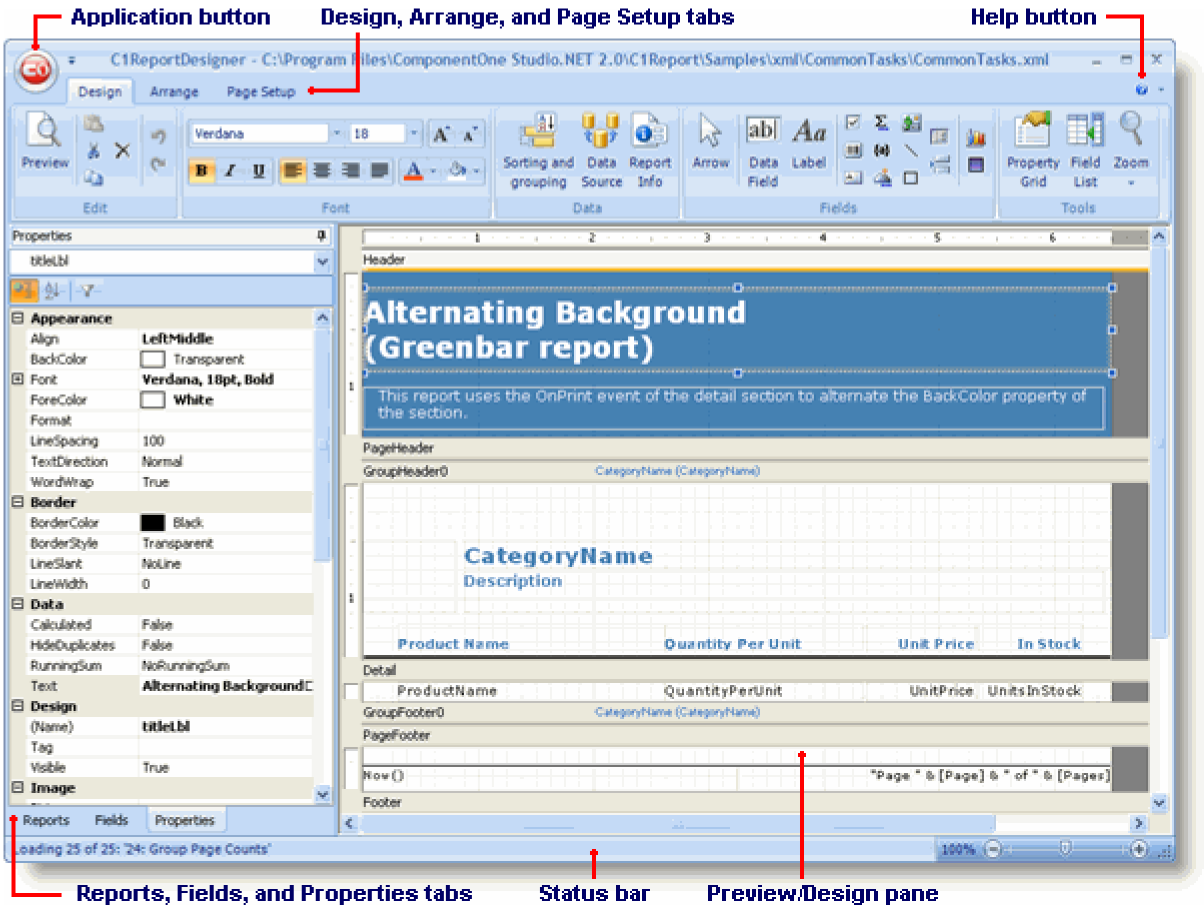
The following topics contain important information about **C1ReportDesigner**, a stand-alone application similar to the report designer in Microsoft Access, including how to create a basic report definition file, modify, print, and export the report definition. This section also demonstrates how to import reports created with Microsoft Access.

## About C1ReportDesigner

The **C1ReportDesigner** application is a tool used for creating and editing C1Report report definition files. The Designer allows you to create, edit, load, and save files (XML) that can be read by the C1Report component. It also allows you to import report definitions from Microsoft Access files (MDB) and VSReport 1.0 (VSR).

To run the Designer, double-click the **C1ReportDesigner.exe** file located by default in the following path: **C:\Program Files\ComponentOne\Studio for WinForms\C1Report\Designer**. Note that this directory reflects the default installation path and its path may be different if you made changes to the installation path. For steps on running the Designer from Visual Studio, see Accessing C1ReportDesigner from Visual Studio.

Here's what the Designer looks like with the CommonTasks.xml file opened:



Appearance

| | |
|---|---|
| Align | LeftMiddle |
| BackColor | □ Transparent |
| Font | Verdana, 18pt, Bold |
| ForeColor | □ White |
| Format | |
| LineSpacing | 100 |
| TextDirection | Normal |
| WordWrap | True |

Border

| | |
|---|---|
| BorderColor | ■ Black |
| BorderStyle | Transparent |
| LineSlant | NoLine |
| LineWidth | 0 |

Data

| | |
|---|---|
| Calculated | False |
| HideDuplicates | False |
| RunningSum | NoRunningSum |
| Text | Alternating Background |

Design

| | |
|---|---|
| (Name) | titleLbl |
| Tag | |
| Visible | True |

Image

Loading 25 of 25: '24: Group Page Counts'

The main Designer window has the following components:

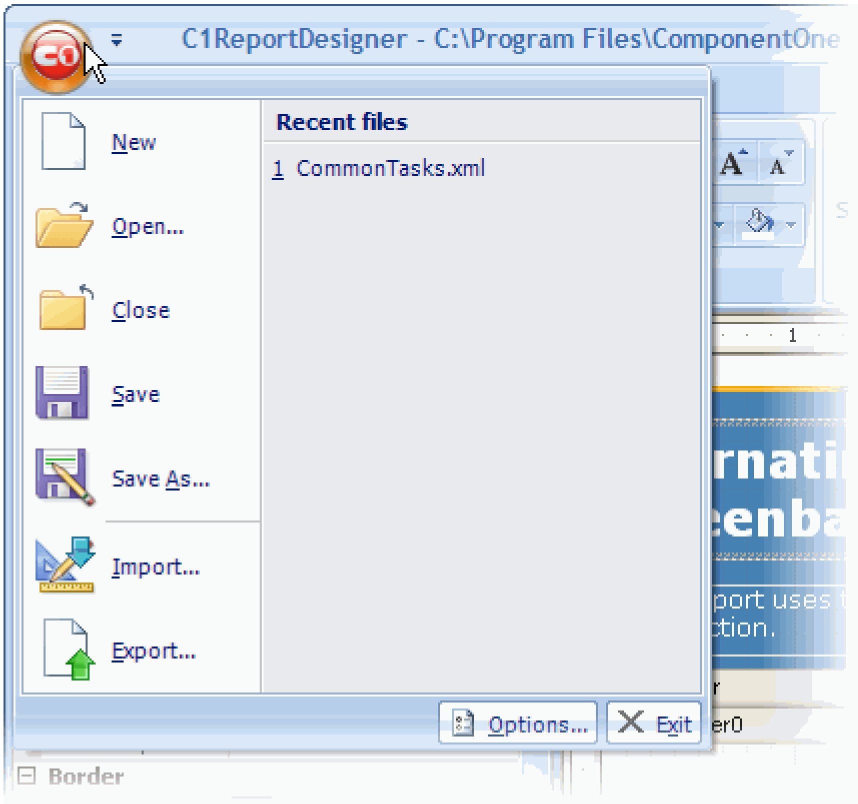
- **Application button**: Click the application button to open a menu to load and save report definition files and to import and export report definitions. See Application Button for more information.
- **Design tab**: Provides shortcuts to the Edit, Font, Data, Fields, and Tools menu functions. See Design Tab for more information.
- **Arrange tab**: Provides shortcuts to the Edit, AutoFormat, Grid, Control Alignment, Position, and Size menu functions. See Arrange Tab for more information.
- **Page Setup tab**: Provides shortcuts to the Edit and Page Layout menu functions. See Page Setup Tab for more information.
- **Preview tab**: Appears only when the report is being viewed in a print preview. See Preview Tab for more information.
- **Help button**: Provides options to open the online help file and view the **About** screen, which displays information about the application.
- **Reports tab**: Lists all reports contained in the current report definition file. You can double-click a report name to preview or edit the report. You can also use the list to rename, copy, and delete reports.
- **Fields tab**: Lists all the fields contained in the current report.
- **Properties tab**: Allows you to edit properties for the objects that are selected in the Designer.
- **Pages tab**: Only available when in preview mode, this tab includes thumbnails of all the pages in the document.
- **Properties tab**: Only available when in preview mode, this tab displays a text outline of the document.
- **Find tab**: Only available when in preview mode, this tab displays fine pane allowing you to search for text in the document.
- **Preview/Design pane**: This is the main working area of the Designer. In preview mode, it displays the current report. In design mode, it shows the report's sections and fields and allows you to change the report definition.
- **Status bar**: The status bar displays information about what the Designer is working on (for example, loading, saving, printing, rendering, importing, and so on). You can zoom in and out of a selected report by dragging the zoom slider at the right of the status bar.

The topics that follow explain how you can use the **C1ReportDesigner** application to create, edit, use, and save report definition files.

## *Application Button*

The **Application** button provides a shortcut menu to load and save report definition files and to import and export report definitions. You can also access the **C1ReportDesigner** application's options through the **Application** menu.

Click the **Application** button to open the menu. It will appear similar to the following:



The menu includes the following options:

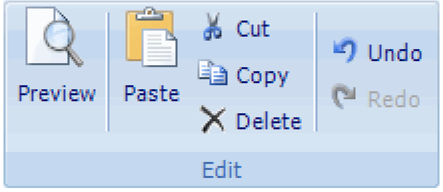
- **New**: Creates a new report definition file.
- **Open**: Brings up the **Open Report Definition File** dialog box, enabling you to select an existing file to open.
- **Close**: Closes the current report definition file.
- **Save**: Saves the report definition file, to the location previously saved.
- **Save As**: Opens the **Save Report Definition** dialog box allowing you to save your report definition as an XML file.
- **Import**: Opens the **Import Report Definition** dialog box enabling you to import Microsoft Access (.mdb and .adp) files and Crystal Reports (.rpt) files. See Importing Microsoft Access Reports and Importing Crystal Reports for more information.
- **Export**: Exports the current report file as an HTML, PDF, RTF, XLS, XLSX, TIF, TXT, or ZIP file.
- **Recent files**: Lists recently opened report definition files. To reopen a file, select it from the list.
- **Options**: Opens the **C1ReportDesigner Options** dialog box which allows you to customize the default appearance and behavior of the **C1ReportDesigner** application. See Setting C1ReportDesigner Options for more information.
- **Exit**: Closes the **C1ReportDesigner** application.

### *Design Tab*

The **Design** tab is located in the **C1ReportDesigner**'s Ribbon menu and provides shortcuts to the Edit, Font, Data, Fields, and Tools menu functions. For more information, see the following topics.

## Edit Group

The **Edit** group on the **Design** tab appears similar to the following:

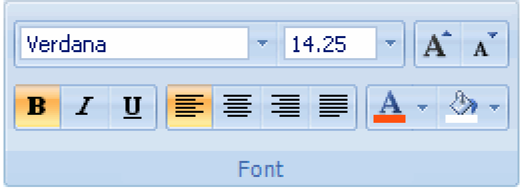


It consists of the following options:

- **Preview**: The Preview button opens a print preview view of the report. To exit the preview, click the **Close Print Preview** button. See Preview Tab and Previewing and Printing a Report for more information.
- **Paste**: Pastes the last copied item.
- **Cut**: Cuts the selected item, removing it from the report and allowing it to be pasted elsewhere.
- **Copy:** Copies the selected item so that it can be pasted elsewhere.
- **Undo**: Undoes the last change that was made to the report definition.
- **Redo**: Redoes the last change that was made to the report definition.

## Font Group

The **Font** group on the **Design** tab appears similar to the following:



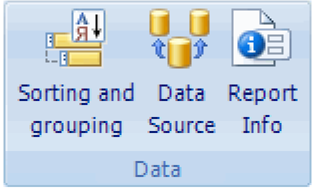
And consists of the following options:

- **Font Name**: Displays the current font of the selected text and allows you to choose another font for the selected item (to do so, click the drop-down arrow next to the font name).
- **Font Size**: Displays the current font size of the selected text and allows you to choose another font size. Type a number in the font size box or click the drop-down arrow to choose a font size.
- **Increase Font Size**: Increases the font size by one point.
- **Decrease Font Size:** Decreases the font size by one point.
- **Bold**: Bold the selected text (you can also press CTRL+B).
- **Italic**: Italicizes the selected text (you can also press CTRL+I).
- **Underline**: Underlines the selected text (you can also press CTRL+U).
- **Left:** Aligns text to the left.

- **Center**: Aligns text to the center.
- **Right**: Aligns text to the right.
- **Justify**: Justifies the selected text.
- **Text Color**: Allows you to select the color of the selected text.
- **Fill Color**: Allows you to select the background color of the selected text.

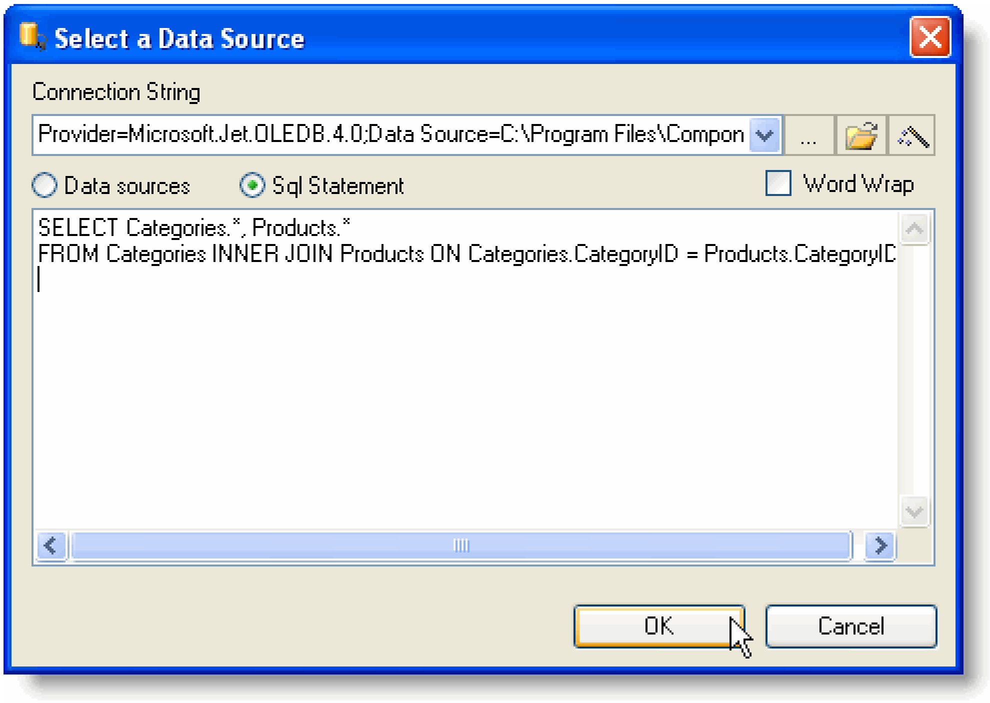
## Data Group

The **Data** group appears similar to the following image:



And consists of the following items:

- **Sorting and grouping**: Clicking this button opens the **Sorting and Grouping** dialog box where you can add and delete sorting and grouping criteria. For more information see Grouping Data and Sorting Data.
- **Data Source**: Clicking this button opens the **Select a Data Source** dialog box. The **Select a Data Source** dialog box allows you to choose a new data source, change the connection string, and edit the SQL statement. Clicking the **Data sources** radio button displays the tables and views in the current data source. Clicking the **Sql statement** radio button allows you to view the current SQL statement:



Connection String

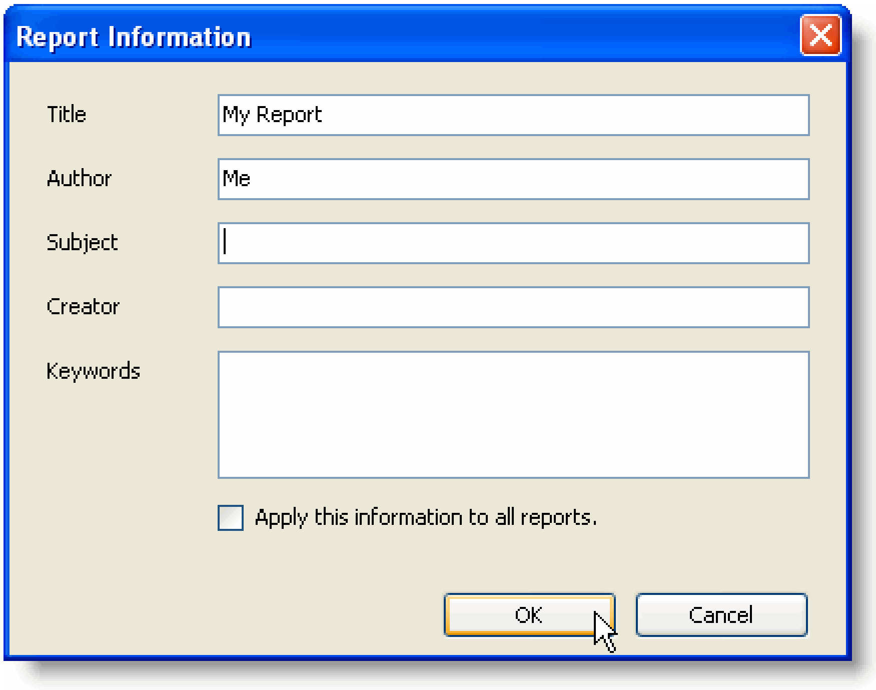
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\Compon

○ Data sources ● Sql Statement □ Word Wrap

```
SELECT Categories.*, Products.*
FROM Categories INNER JOIN Products ON Categories.CategoryID = Products.CategoryIC
```

To select change the connection string, click the ellipses button. This will open the **Data Link Properties** dialog box. To open an XML Schema Definition file (XDS) click the **Open** button (See Loading Data from an XSD File for more information). To edit or change the SQL statement, click the **Build SQL Statement** button which will open the **Sql Builder** dialog box.

- **Report Info**: Opens the **Report Information** dialog box. This dialog box allows you to set the report's Title, Author, Subject, Creator, and Keywords. You can also choose to apply report information to all reports.


Report Information

Title: My Report

Author: Me

Subject:

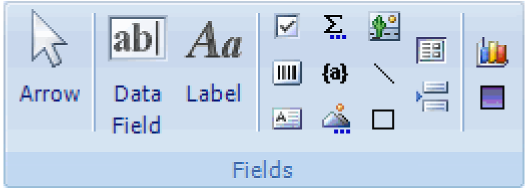
Creator:

Keywords:

□ Apply this information to all reports.

## Fields Group

The **Fields** group of the **Design** tab in the **C1ReportDesigner** application provides tools for creating report fields. This toolbar is enabled only in design mode. The **Fields** group on the **Design** tab appears similar to the following:



Each field button creates a field and initializes its properties. The **Fields** group consists of the following options:

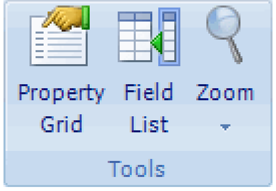
- **Arrow**: Returns the mouse cursor to an arrow cursor.
- **Data Field**: Creates a field that is bound to the source recordset. When you click this button, a menu appears and you can select the recordset field. Bound fields are not limited to displaying raw data from the database. You can edit their Text property and use any VBScript expression.

- **Label**: Creates a field that displays static text.
- **Add CheckBox** (☑): Creates a bound field that displays a Boolean value as a check box. By default, the check box displays a regular check mark. You can change it into a radio button or cross mark by changing the value of the field's Checkbox property after it has been created.
- **Add BarCode** (▥): Creates a field that displays a barcode. When you click this button, a menu appears where you can select other fields that are contained in the same report definition file to be displayed as a barcode.
- **Add Rtf Field** (▤): Creates an RTF field. When you click this button, a menu appears where you can select other fields that are contained in the same report definition file to be displayed in RTF format.
- **Add Calculated Field** (Σ): Creates a calculated field. When you click this button, the code editor dialog box appears so you can enter the VBScript expression whose value you want to display.
- **Add Common Calculated Field** ({a}): Creates a field with a commonly used expression. When you click this button, a menu appears and you can select expressions that render the date or time when the report was created or printed, the page number, page count, or "page n of m", or the report name.
- **Add Unbound Picture** (🖼): Creates a field that displays a static picture, such as a logo. When you click this button, a dialog box appears to prompt you for a picture file to insert in the report. A copy is made of the picture you select and placed in the same directory as the report file. You must distribute this file with the application unless you embed the report file in the application. When you embed a report file in your application, any unbound picture files are embedded too.
- **Add Bound Picture** (🖼): Creates a field that displays a picture (or object) stored in the recordset. When you click this button, a menu appears so you can select a picture field in the source recordset (if there is one; not all recordsets contain this type of field).
- **Add Line** (╲): Creates a line. Lines are often used as separators.
- **Add Rectangle** (□): Creates a rectangle. Rectangles are often used to highlight groups of fields or to create tables and grids.
- **Add SubReport** (▤): Creates a field that displays another report. When you click this button, a menu appears and you can select other reports that are contained in the same report definition file. See Creating a Master-Detail Report Using Subreports for more information.
- **Add Page Break** (▤): Creates a field that inserts a page break.
- **Add Chart Field** (📊):Creates a field that displays a chart. Unlike most bound fields, Chart fields display multiple values. To select the data you want to display, set the Chart field's **Chart.DataX** and **Chart.DataY** properties. To format the values along the X and Y axis, set the **Chart.FormatX** and **Chart.FormatY** properties. See Adding Chart Fields for more information.
- **Add Gradient Field** (■): Creates a gradient field. Gradients are often used as a background feature to make other fields stand out. See Adding Gradient Fields for more information.

See Enhancing the Report with Fields for more information. For more information on adding fields to your report, see Creating Report Fields.

## Tools Group

The **Tools** group on the **Design** tab appears similar to the following:



It consists of the following options:

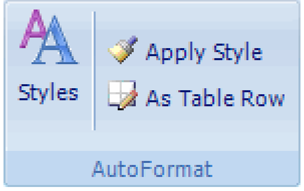
- **Property Grid**: Selecting this item brings the **Properties** tab into view on the left pane. Note that you can also use F4 to view the **Properties** tab.
- **Field List**: Selecting this item brings the **Fields** tab into view on the left pane.
- **Zoom**: Allows you to select a value to set the zoom level of the report.

### *Arrange Tab*

The **Arrange** tab is located in the **C1ReportDesigner**'s Ribbon menu and provides shortcuts to the Edit, AutoFormat, Grid, Control Alignment, Position, and Size menu functions. See Edit Group for information about that group, and the following topics for the other groups.

## AutoFormat Group

The **AutoFormat** Group on the **Arrange** tab of the Ribbon appears similar to the following image:

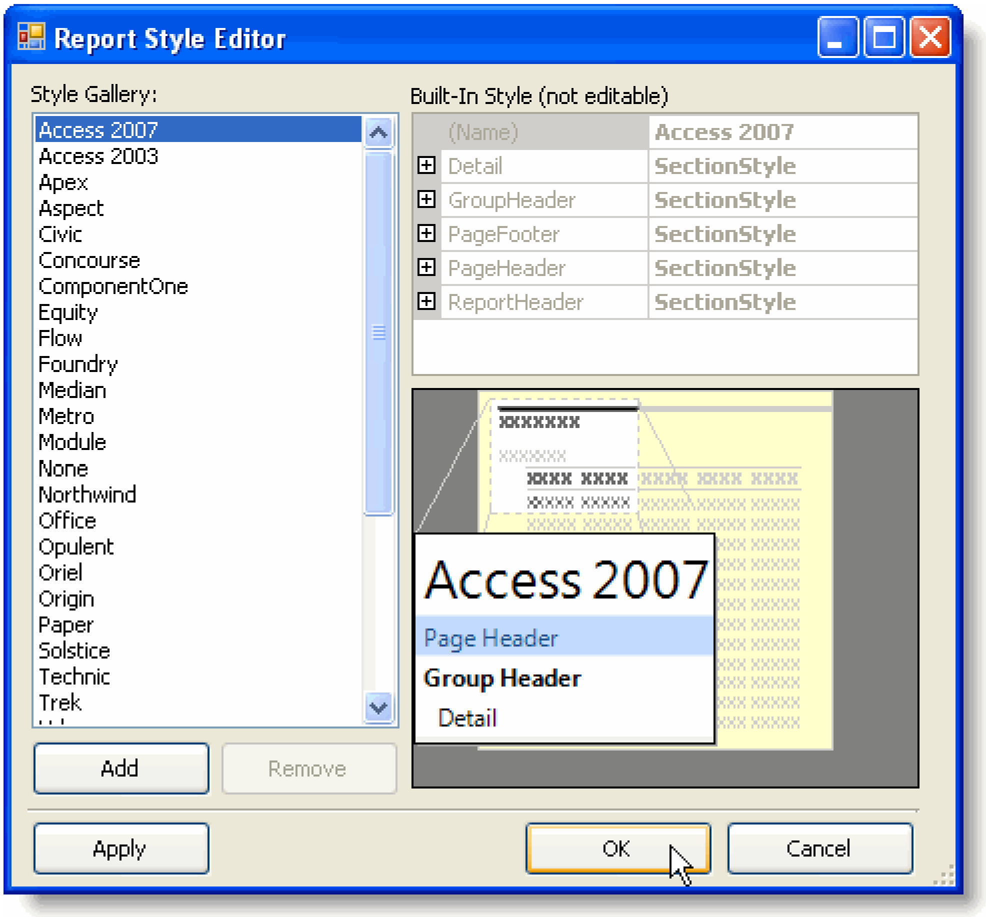


The **AutoFormat** group consists of the following options:

- **Styles**: Opens the **Report Style Editor** dialog box, where you can choose a built-in style or create and edit your own custom style.
- **Apply Style**: Applies style to the current selection.
- **As Table Row**: Formats the current selection as a table row.

#### Report Style Editor

You can access the **Report Style Editor** dialog box by clicking the **Styles** button of the **AutoFormat** group. The **Report Style Editor** dialog box appears similar to the following image:



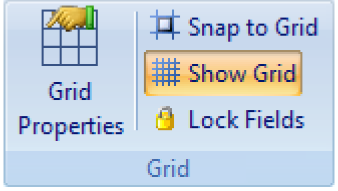
| (Name) | Access 2007 |
|---|---|
| ⊞ Detail | SectionStyle |
| ⊞ GroupHeader | SectionStyle |
| ⊞ PageFooter | SectionStyle |
| ⊞ PageHeader | SectionStyle |
| ⊞ ReportHeader | SectionStyle |

It includes several elements including:

- **Style Gallery List**: The Style Gallery list displays all the currently available built-in and custom styles. See Style Gallery for information about the available built-in styles.
- **Add Button**: The **Add** button adds a custom style to the Style Gallery list. The style that is added will be based on the style that was selected in the Style Gallery list when the **Add** button was clicked.
- **Remove Button**: The **Remove** button removes a selected custom style. The button is only available when a custom style is selected in the Style Gallery list.
- **Property Grid**: The Property grid lets you change the properties and edit a custom style. The Property grid is only available and editable when a custom style is selected in the Style Gallery list.
- **Preview Window**: The Preview window displays a preview of the style selected in the Style Gallery list.
- **Apply Button**: Clicking the **Apply** button applies the style to your selection without closing the dialog box.
- **OK Button**: Clicking the **OK** button closes the dialog box, applies your changes, and sets the style you choose as the current selected style.
- **Cancel Button**: Clicking the **Cancel** button allows you to cancel any changes you've made to styles.

## Grid Group

The **Grid** group on the **Arrange** tab of the Ribbon appears similar to the following image:

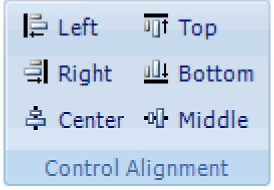


The **Grid** group consists of the following buttons:

- **Grid Properties**: Opens the **C1ReportDesigner Options** dialog box. For details see Setting C1ReportDesigner Options.
- **Snap to Grid**: Snaps elements to the grid. When this item is selected elements cannot be placed in between lines of the grid.
- **Show Grid**: Shows a grid in the background of the report in the preview. The grid can help you place and align elements. By default, this option is selected.
- **Lock Fields**: After you've placed the fields in the desired positions, you can lock them to prevent inadvertently moving them with the mouse or keyboard. Use the **Lock Fields** button to lock and unlock the fields.

## Control Alignment Group

The **Control Alignment** group on the **Arrange** tab of the Ribbon appears similar to the following image:



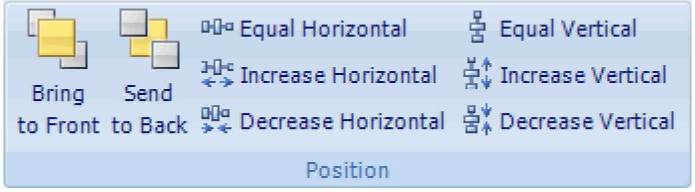
The **Control Alignment** group consists of the following options:

- **Left**: Horizontally aligns the selected item to the left.
- **Right**: Horizontally aligns the selected item to the right.
- **Center**: Horizontally aligns the selected item to the center.
- **Top**: Vertically aligns the selected item to the top.
- **Bottom**: Vertically aligns the selected item to the bottom.
- **Middle**: Vertically aligns the selected item to the middle.

Items can be both horizontally and vertically aligned – so, for example, an item can be both aligned to the left and the top.

## Position Group

The **Position** group on the **Arrange** tab of the Ribbon controls spacing between elements and how elements are layered. It appears similar to the following image:

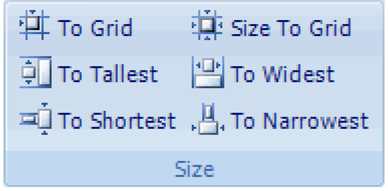


The **Position** group consists of the following options:

- **Bring to Front**: Brings the selected item to the front, in front of all layered items.
- **Send to Back**: Sends the selected item to the back, behind all layered objects.
- **Equal Horizontal**: Provides equal horizontal spacing between elements.
- **Increase Horizontal**: Increases the horizontal spacing between elements.
- **Decrease Horizontal**: Decreases the horizontal spacing between elements.
- **Equal Vertical**: Provides equal vertical spacing between elements.
- **Increase Vertical**: Increases the vertical spacing between elements.
- **Decrease Vertical**: Decreases the vertical spacing between elements.

## Size Group

The **Size** group on the **Arrange** tab of the Ribbon controls how elements are aligned and sized. It appears similar to the following image:



The **Size** group consists of the following options:

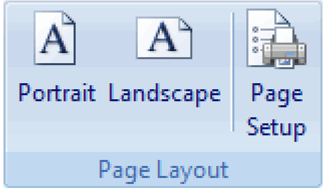
- **To Grid**: Aligns the selected items to the grid.
- **To Tallest**: Sets the height of all selected elements to the tallest selected item.
- **To Shortest**: Sets the height of all selected items to the shortest selected item.
- **Size To Grid**: Sizes the selected items to the grid.
- **To Widest**: Sets the width of all selected items to the width of the widest selected item.
- **To Narrowest**: Sets the width of all selected items to the width of the narrowest selected item.

### *Page Setup Tab*

The **Page Setup** tab is located in the **C1ReportDesigner**'s Ribbon menu and provides shortcuts to the Edit and Page Layout menu functions. See Edit Group and Page Layout Group for more information.

## Page Layout Group

The **Page Layout** group on the **Page Setup** tab of the Ribbon appears similar to the following image:



The **Page Layout** group consists of the following options:

- **Portrait**: Changes the layout of your report to **Portrait** view (where the height is longer than the width).
- **Landscape**: Changes the layout of your report to **Landscape** view (where the height is shorter than the width).
- **Page Setup**: Opens the printer's **Page Setup** dialog box.

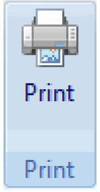
### *Preview Tab*

The **Preview** tab is located in the **C1ReportDesigner**'s Ribbon menu and provides shortcuts to the Print, Page Layout, Zoom, Navigation, Tools, Export, and Close Preview menu functions. To access the Preview tab and a print preview of your report, click the **Preview** option located in the **Edit** group of the Ribbon. See Edit Group for more information.

See Page Layout Group for information about that group, and the following topics for the other groups in the **Preview** tab.

## Print Group

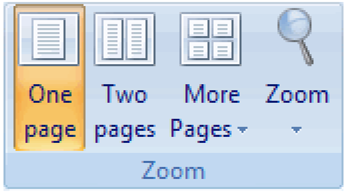
The **Print** group on the **Preview** tab of the Ribbon appears similar to the following image:



The **Print** group consists of the Print option. Clicking the **Print** option opens your printer options for printing the report.

## Zoom Group

The **Zoom** group on the **Preview** tab of the Ribbon controls how the print preview is zoomed and appears similar to the following image:

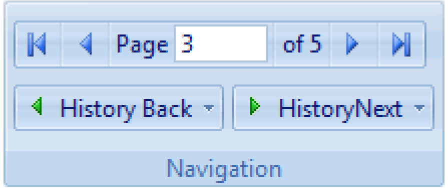


The **Zoom** group consists of the following options:

- **One page**: Allows you to preview one page at a time.
- **Two pages**: Allows you to preview two pages at a time.
- **More pages**: Clicking the drop-down arrow allows you to preview multiple pages at a time and includes the following options: **Four pages**, **Eight pages**, **Twelve pages**.
- **Zoom**: Zooms the page in to a specific percent or to fit in the window.

## Navigation Group

The **Navigation** group on the **Preview** tab of the Ribbon controls how the print preview is navigated and appears similar to the following image:

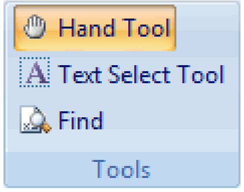


The **Navigation** group consists of the following options:

- **First page**: Navigates to the first page of the preview.
- **Previous page**: Navigates to the previous page of the preview.
- **Page**: Entering a number in this textbox navigates the preview to that page.
- **Next page**: Navigates to the next page of the preview.
- **Last page**: Navigates to the last page of the preview.
- **History Back**: Returns to the last page viewed.
- **History Next**: Moves to the next page viewed. This is only visible after the **History Back** button is clicked.

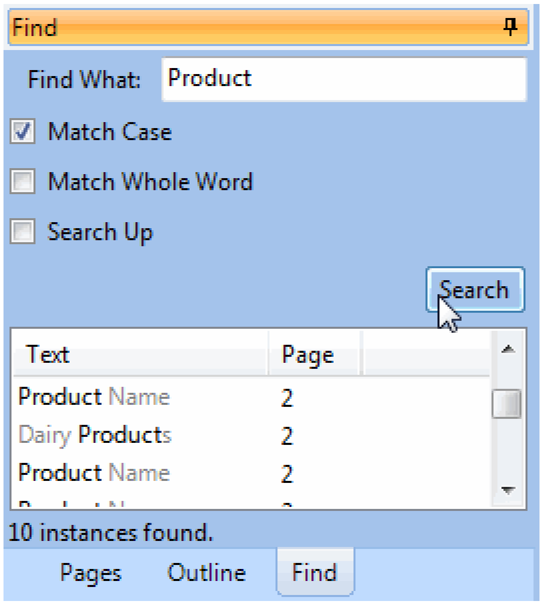
## Tools Group

The **Tools** group on the **Preview** tab of the Ribbon contains tools for selecting and locating items in the preview and appears similar to the following image:



The **Tools** group consists of the following buttons:

- **Hand Tool**: The hand tool allows you to move the preview through a drag-and-drop operation.
- **Text Select Tool**: The text select tool allows you to select text through a drag-and-drop operation. You can then copy and paste this text to another application.
- **Find**: Clicking the **Find** option opens the **Find** pane where you can search for text in the document. To find text enter the text to find, choose search options (if any), and click **Search**:



Find What: Product

☑ Match Case

□ Match Whole Word

□ Search Up

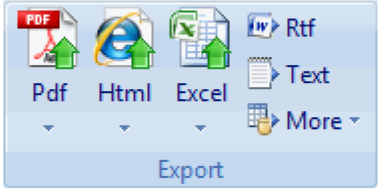
| Text | Page |
|---|---|
| Product Name | 2 |
| Dairy Products | 2 |
| Product Name | 2 |

10 instances found.

You can click a search result to locate it in the document.

## Export Group

The **Export** group on the **Preview** tab of the Ribbon contains options for exporting the report to various formats and appears similar to the following image:



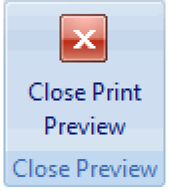
Each item in the Export group opens the **Export Report to File** dialog box where you can choose a location for your exported file. The **Export** group consists of the following options:

- **Pdf**: Exports the document to a PDF file. The drop-down arrow includes options for **PDF (system fonts)** and **PDF (embedded fonts)** to choose if you want to use system fronts or embed your chosen fonts in the PDF file.
- **Html**: Exports the document to an HTML file. You can then copy and paste this text to another application. The drop-down arrow includes options for **Plain HTML**, **Paged HTML**, and **Drilldown HTML** allowing you to choose if you want to export to a plain HTML file, multiple HTML files that can be paged using included arrow links, or an HTML file that displays content that can be drilled down to.
- **Excel**: Exports the document to a Microsoft Excel file. The drop-down arrow includes options for **Microsoft Excel 97** and **Microsoft Excel 2007 – OpenXML** allowing you to choose if you want to save the document as an XLS or XLSX file.
- **Rtf**: Exports the document to a Rich Text File (RTF).
- **Text**: Exports the document to a Text file (TXT).
- **More**: Clicking the **More** drop-down arrow includes additional options to export the report including: **Tagged Image File Format** (export as TIFF), **RTF (fixed positioning)**, **Single Page Text**, **Compressed Metafile** (export as ZIP).

**Note:** When a document is exported to the RTF or the DOCX formats with the "preserve pagination" option selected, text is placed in text boxes and the ability to reflow text in the resulting document may be limited.

## Close Preview Group

The **Close Preview** group on the **Preview** tab of the Ribbon appears similar to the following image:



It includes just one button, **Close Print Preview**, enabling you to close the preview view and return to the design view. To return to the Review view again, click the **Preview** button in the **Edit** group.

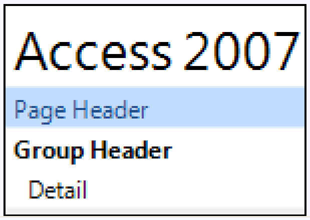
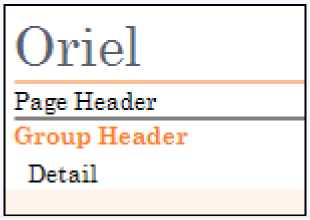
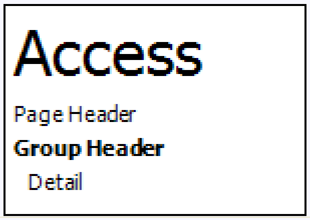
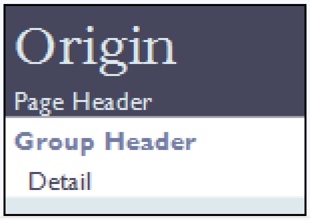
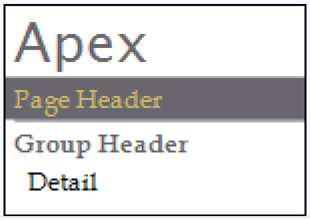
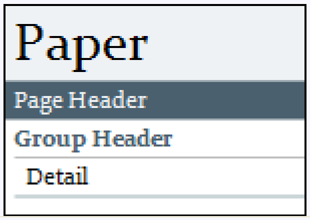
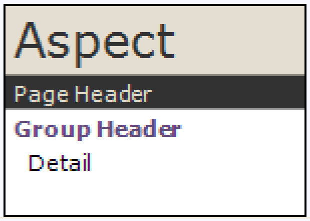
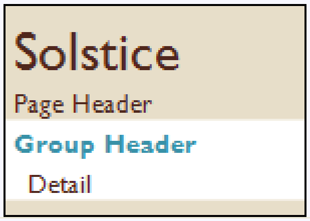
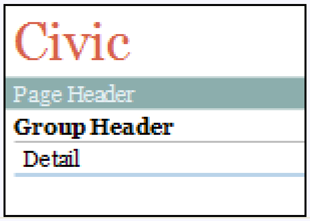
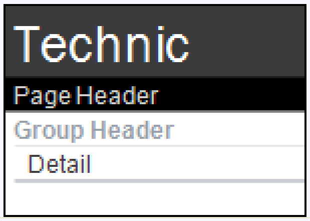
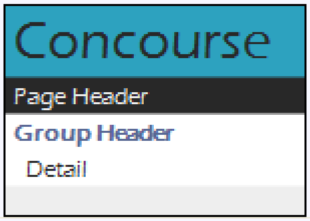
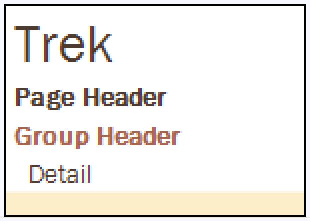
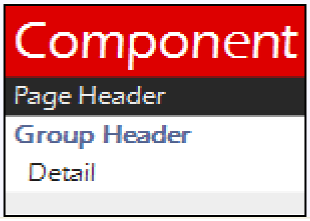
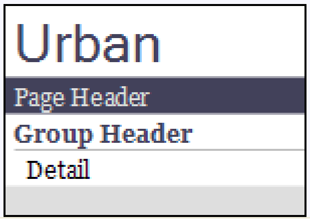
### *Style Gallery*

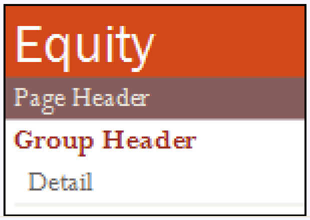
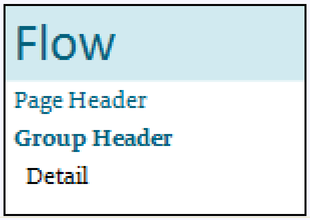
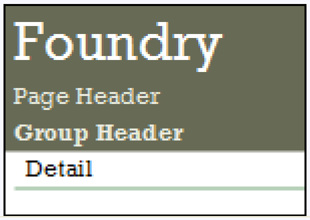
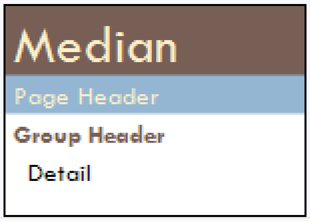
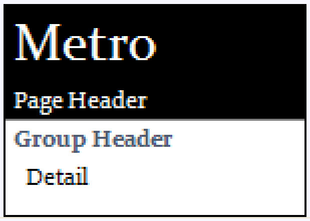
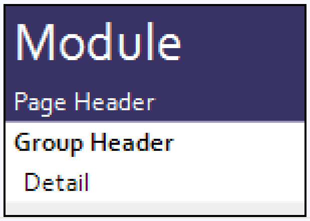
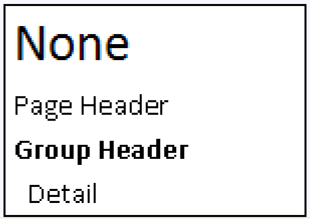
The **Style Gallery** dialog box details all the available built-in and custom styles that you can use to format your report. Built-in styles include standard Microsoft AutoFormat themes, including Vista and Office 2007 themes. You can access the **Style Gallery** from the **C1ReportDesigner** application by selecting the **Arrange** tab and clicking **Styles**.

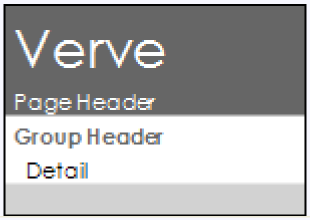
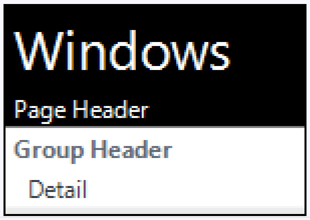
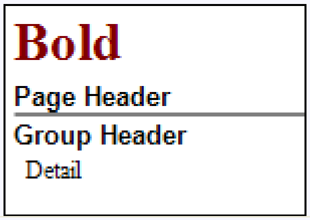
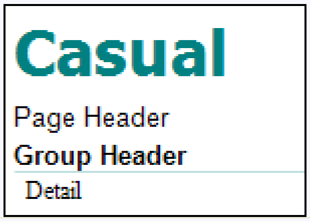
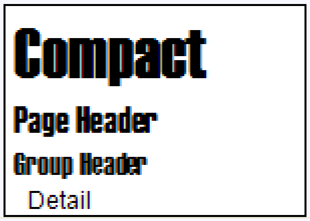
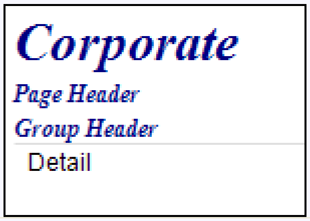
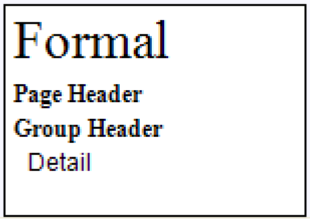
The following built-in styles are included:

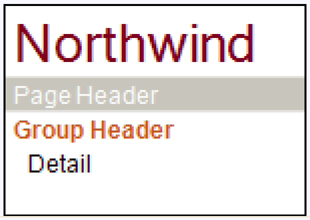
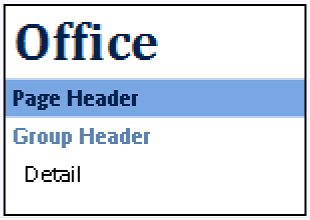
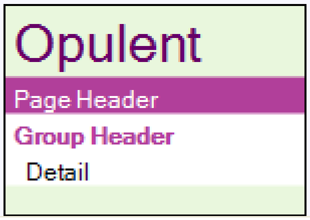
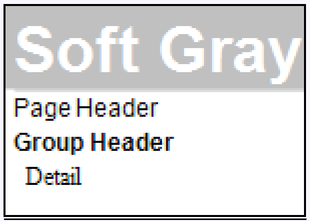
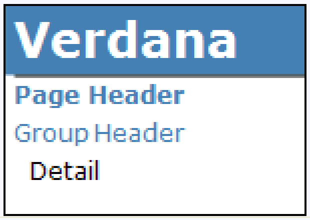
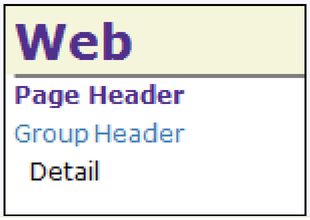
| Style Name | Preview |
|---|---|

| Style Name | Preview |
|---|---|

|  |  |  |  |
|---|---|---|---|
| Access 2007 | <br>Access 2007<br>Page Header<br>Group Header<br>Detail | Oriel | <br>Oriel<br>Page Header<br>Group Header<br>Detail |
| Access 2003 | <br>Access<br>Page Header<br>Group Header<br>Detail | Origin | <br>Origin<br>Page Header<br>Group Header<br>Detail |
| Apex | <br>Apex<br>Page Header<br>Group Header<br>Detail | Paper | <br>Paper<br>Page Header<br>Group Header<br>Detail |
| Aspect | <br>Aspect<br>Page Header<br>Group Header<br>Detail | Solstice | <br>Solstice<br>Page Header<br>Group Header<br>Detail |
| Civic | <br>Civic<br>Page Header<br>Group Header<br>Detail | Technic | <br>Technic<br>Page Header<br>Group Header<br>Detail |
| Concourse | <br>Concourse<br>Page Header<br>Group Header<br>Detail | Trek | <br>Trek<br>Page Header<br>Group Header<br>Detail |
| ComponentOne | <br>Component<br>Page Header<br>Group Header<br>Detail | Urban | <br>Urban<br>Page Header<br>Group Header<br>Detail |

| | |
|---|---|
| Equity | <br>Equity<br>Page Header<br>Group Header<br>Detail |
| Flow | <br>Flow<br>Page Header<br>Group Header<br>Detail |
| Foundry | <br>Foundry<br>Page Header<br>Group Header<br>Detail |
| Median | <br>Median<br>Page Header<br>Group Header<br>Detail |
| Metro | <br>Metro<br>Page Header<br>Group Header<br>Detail |
| Module | <br>Module<br>Page Header<br>Group Header<br>Detail |
| None | <br>None<br>Page Header<br>Group Header<br>Detail |

| | |
|---|---|
| Verve | <br>Verve<br>Page Header<br>Group Header<br>Detail |
| Windows Vista | <br>Windows<br>Page Header<br>Group Header<br>Detail |
| Bold | <br>Bold<br>Page Header<br>Group Header<br>Detail |
| Casual | <br>Casual<br>Page Header<br>Group Header<br>Detail |
| Compact | <br>Compact<br>Page Header<br>Group Header<br>Detail |
| Corporate | <br>Corporate<br>Page Header<br>Group Header<br>Detail |
| Formal | <br>Formal<br>Page Header<br>Group Header<br>Detail |

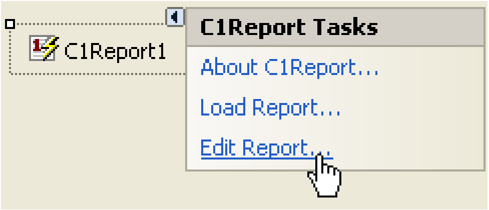
| | |
|---|---|
| Northwind |  |
| Office |  |
| Opulent |  |
| Soft Gray |  |
| Verdana |  |
| WebReport |  |

## Accessing C1ReportDesigner from Visual Studio

To access the **C1ReportDesigner** application from Visual Studio, use any one of the following methods:

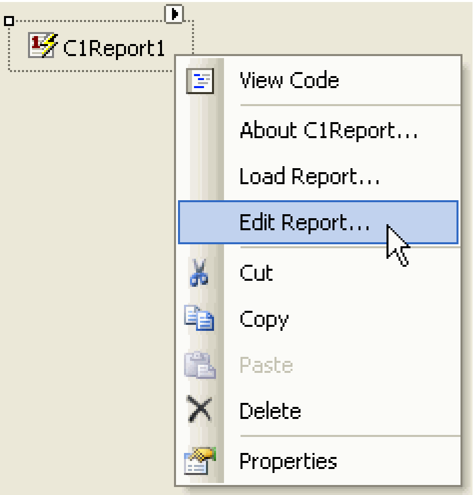
- **C1Report Tasks Menu**

  Click the smart tag (▸) in the upper-right corner of the C1Report component to open the **C1Report Tasks** menu, and select **Edit Report**.

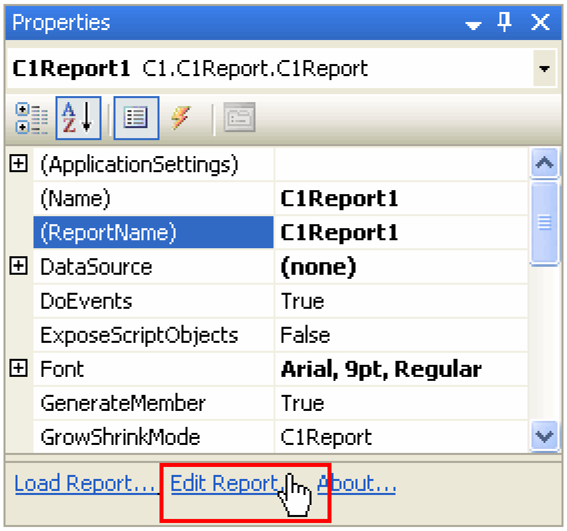
- **Context Menu**

  Right-click the C1Report component and select **Edit Report** from the context menu.



- **Properties Window**

  Below the Properties window, click the **Edit Report** link.

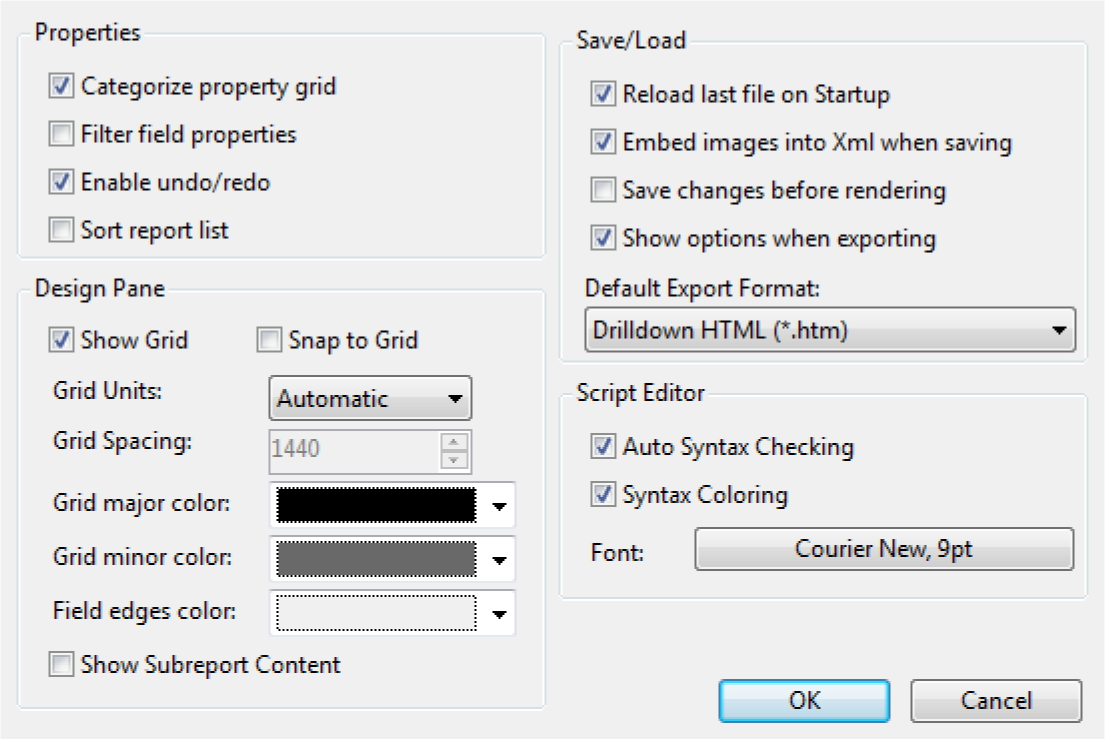


C1Report1 C1.C1Report.C1Report

| | |
|---|---|
| ⊞ (ApplicationSettings) | |
| (Name) | **C1Report1** |
| (ReportName) | **C1Report1** |
| ⊞ DataSource | **(none)** |
| DoEvents | True |
| ExposeScriptObjects | False |
| ⊞ Font | **Arial, 9pt, Regular** |
| GenerateMember | True |
| GrowShrinkMode | C1Report |

## Setting C1ReportDesigner Options

To access the **C1ReportDesigner Options** dialog box, click the **Application** button and then **Options**. For more information about the **Application** button, see Application Button. The **C1ReportDesigner Options** dialog box appears similar to the following image:



Properties

☑ Categorize property grid

□ Filter field properties

☑ Enable undo/redo

□ Sort report list

Design Pane

☑ Show Grid □ Snap to Grid

Grid Units: Automatic

Grid Spacing: 1440

Grid major color: ■

Grid minor color: ▨

Field edges color:

□ Show Subreport Content

Save/Load

☑ Reload last file on Startup

☑ Embed images into Xml when saving

□ Save changes before rendering

☑ Show options when exporting

Default Export Format: Drilldown HTML (*.htm)

Script Editor

☑ Auto Syntax Checking

☑ Syntax Coloring

Font: Courier New, 9pt

The **C1ReportDesigner Options** dialog box includes options to control the appearance and behavior of the application. Options include:

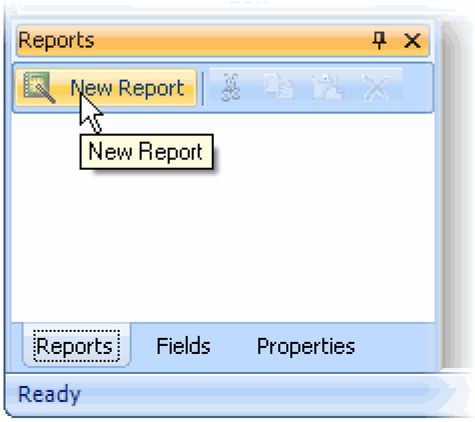
- **Categorize property grid**: Categorizes the Properties grid by property type. The Properties grid can be accessed by clicking the **Properties** tab located in the bottom of the left pane in Design view.
- **Filter field properties**: Filters the Properties grid by properties that have been set. The Properties grid can be accessed by clicking the **Properties** tab located in the bottom of the left pane in Design view.
- **Enable undo/redo**: Enables undo and redo in the application.
- **Sort report list**: Sorts the list of reports listed on the **Reports** tab. Reports can be accessed by clicking the **Reports** tab located in the bottom of the left pane in Design view.
- **Show Grid**: Shows the grid in the report preview window.
- **Snap to Grid**: Snaps all objects the grid in the report. If this option is selected, you will not be able to place objects between grid lines.
- **Grid Units**: Indicates how the grid is spaced. Options include **Automatic**, **English (in)**, **Metric (cm)**, and **Custom**.
- **Grid Spacing**: Sets the spacing of grid lines. This option is only available when the **Grid Units** option is set to **Custom**.
- **Grid major color**: Set the color of major grid lines.
- **Grid minor color**: Sets the color of minor grid lines.
- **Field edges color**: Sets the color of field edges in the report.
- **Show Subreport Content**: Shows sub-report content in the report.
- **Reload last file on Startup**: If this option is checked, the last opened file will appear whenever the **C1ReportDesigner** application is opened.

- **Embed images into Xml when saving**: When the report is saved, images will be embedded into XML if this option is checked.
- **Save changes before rendering**: Checking this option saves the report before rendering.
- **Save options when exporting**: Checking this option saves the report's options when exporting.
- **Default Export Format**: Sets the default export format. For more information about exporting see Export Group.
- **Auto Syntax Checking**: Determines if syntax is automatically checked in the **VBScript Editor** dialog box.
- **Syntax Coloring**: Determines if syntax text is automatically colored  in the **VBScript Editor** dialog box.
- **Font**: Defines the appearance of the text used in the **VBScript Editor** dialog box.
- **OK**: Click **OK** to save your changes.
- **Cancel**: Click **Cancel** to cancel any changes that you have made.

## Creating a Basic Report Definition

You can design your report to display your data in a variety of ways on the printed page. Using the **C1ReportDesigner** application, you can design comprehensive lists, summaries, or special subsets of data, like an invoice.

The easiest way to start a new report is to use the **C1Report Wizard**. In the **C1Report Wizard** click the **Application** button and select **New** from the menu to create a new report. To access the **C1Report Wizard**, click the **New Report** button from the **Reports** tab:



From Visual Studio, you can also access the **C1Report Wizard** from the C1Report context menu or **C1Report Tasks** menu, click **Edit Report**. For more information accessing the **Edit Report** link, see C1Report Tasks Menu or C1Report Context Menu.

The **C1Report Wizard** appears, and will guide you through five easy steps:

1. **Select the data source for the new report.**

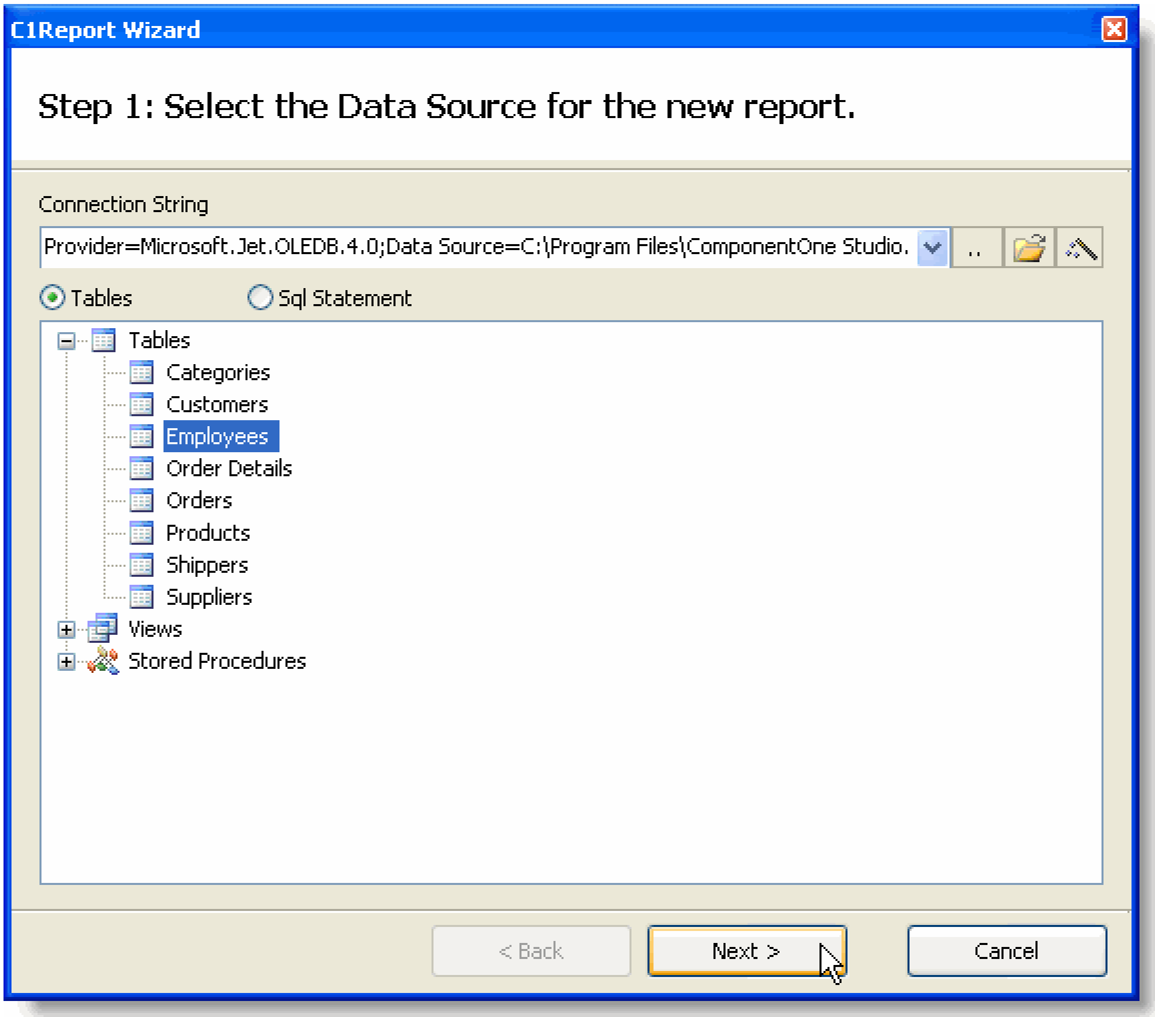
   Use this page to select the ConnectionString and RecordSource that will be used to retrieve the data for the report.

   You can specify the ConnectionString in three ways:

- Type the string directly into the editor.
- Use the drop-down list to select a recently used connection string (the Designer keeps a record of the last eight connection strings).
- Click the button with the **ellipsis** button **(…)** to bring up the standard connection string builder.

You can specify the RecordSource string in two ways:

- Click the **Table** option and select a table from the list.
- Click the **SQL** option and type (or paste) an SQL statement into the editor.



C1Report Wizard

Step 1: Select the Data Source for the new report.

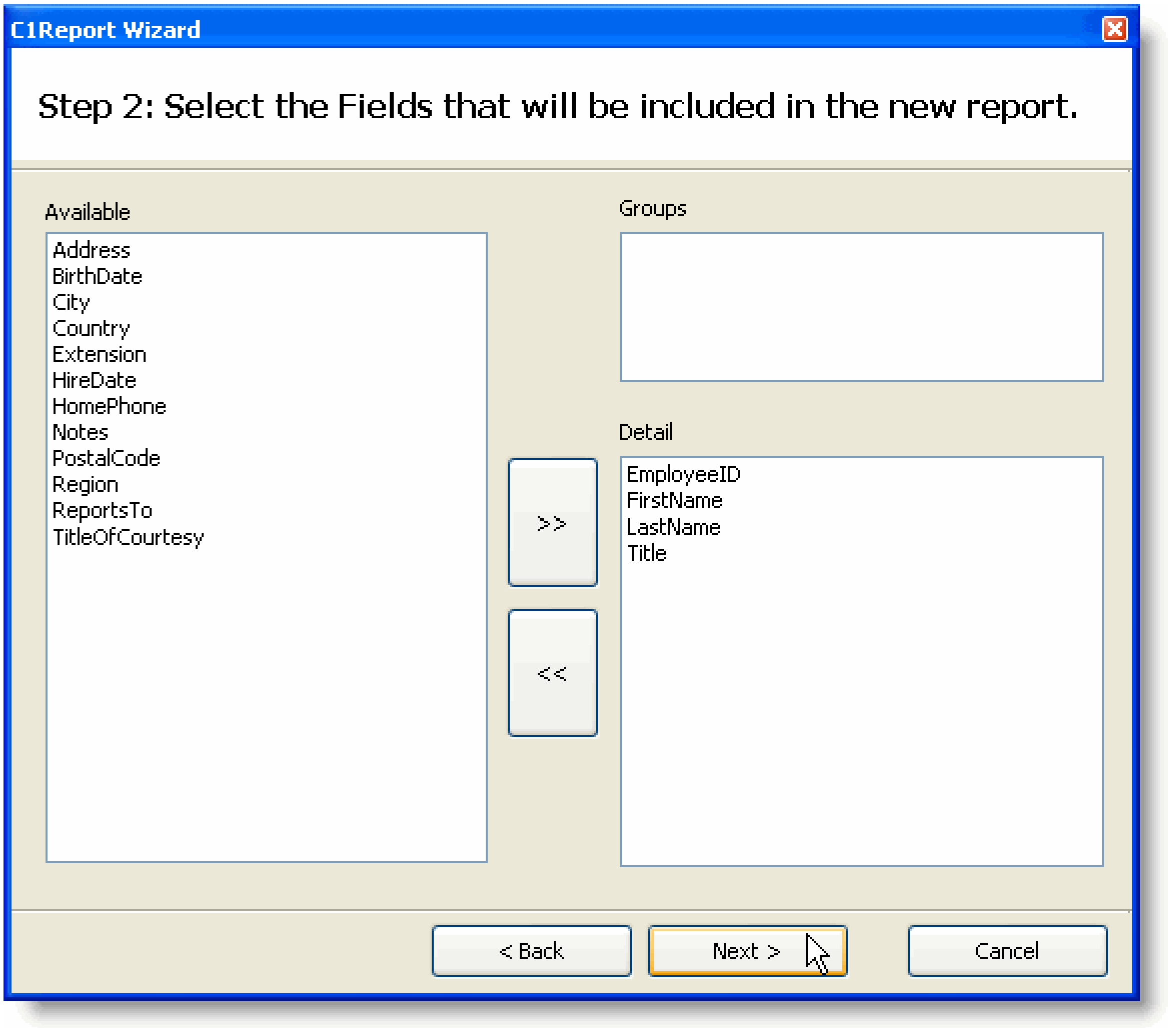
Connection String: Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\ComponentOne Studio.

● Tables ○ Sql Statement

2. **Select the fields you want to include in the report.**

   This page contains a list of the fields available from the recordset you selected in Step 1, and two lists that define the group and detail fields for the report. Group fields define how the data will be sorted and summarized, and detail fields define what information you want to appear in the report.

   You can move fields from one list to another by dragging them with your mouse pointer. Drag fields into the **Detail** list to include them in the report, or drag within the list to change their order. Drag fields back into the **Available** list to remove them from the report.


C1Report Wizard

Step 2: Select the Fields that will be included in the new report.

Available

Address
BirthDate
City
Country
Extension
HireDate
HomePhone
Notes
PostalCode
Region
ReportsTo
TitleOfCourtesy

Groups

Detail

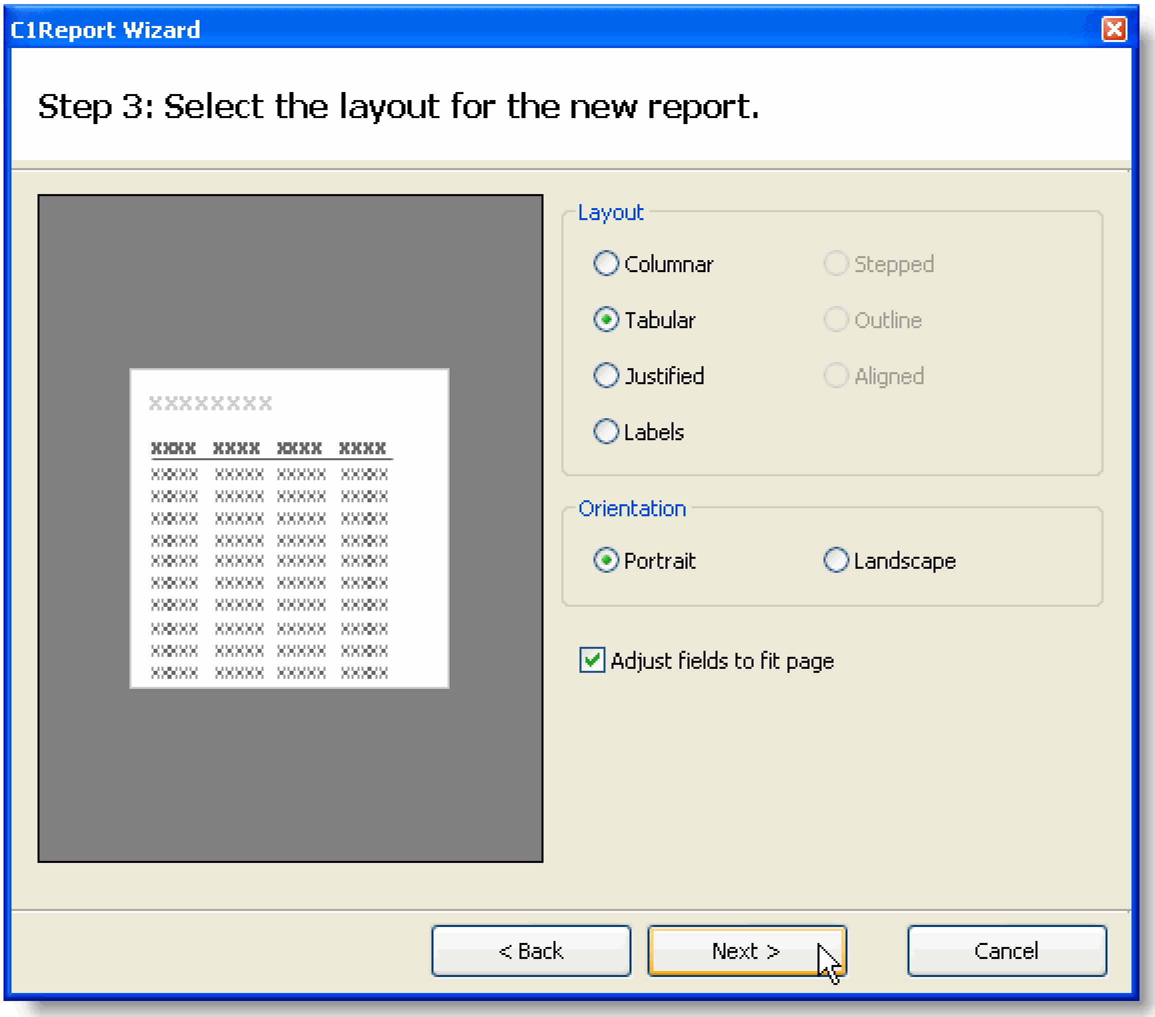
EmployeeID
FirstName
LastName
Title

3. **Select the layout for the new report.**

   This page offers you several options to define how the data will be organized on the page. When you select a layout, a thumbnail preview appears on the left to give you an idea of what the layout will look like on the page. There are two groups of layouts, one for reports that have no groups and one for reports with groups. Select the layout that best approximates what you want the final report to look like.

   This page also allows you to select the page orientation and whether fields should be adjusted to fit the page width.

   The **Labels** layout option is used to print Avery-style labels. If you select this option, you will see a page that prompts you for the type of label you want to print.



C1Report Wizard

Step 3: Select the layout for the new report.

Layout

○ Columnar ○ Stepped

● Tabular ○ Outline

○ Justified ○ Aligned

○ Labels

Orientation

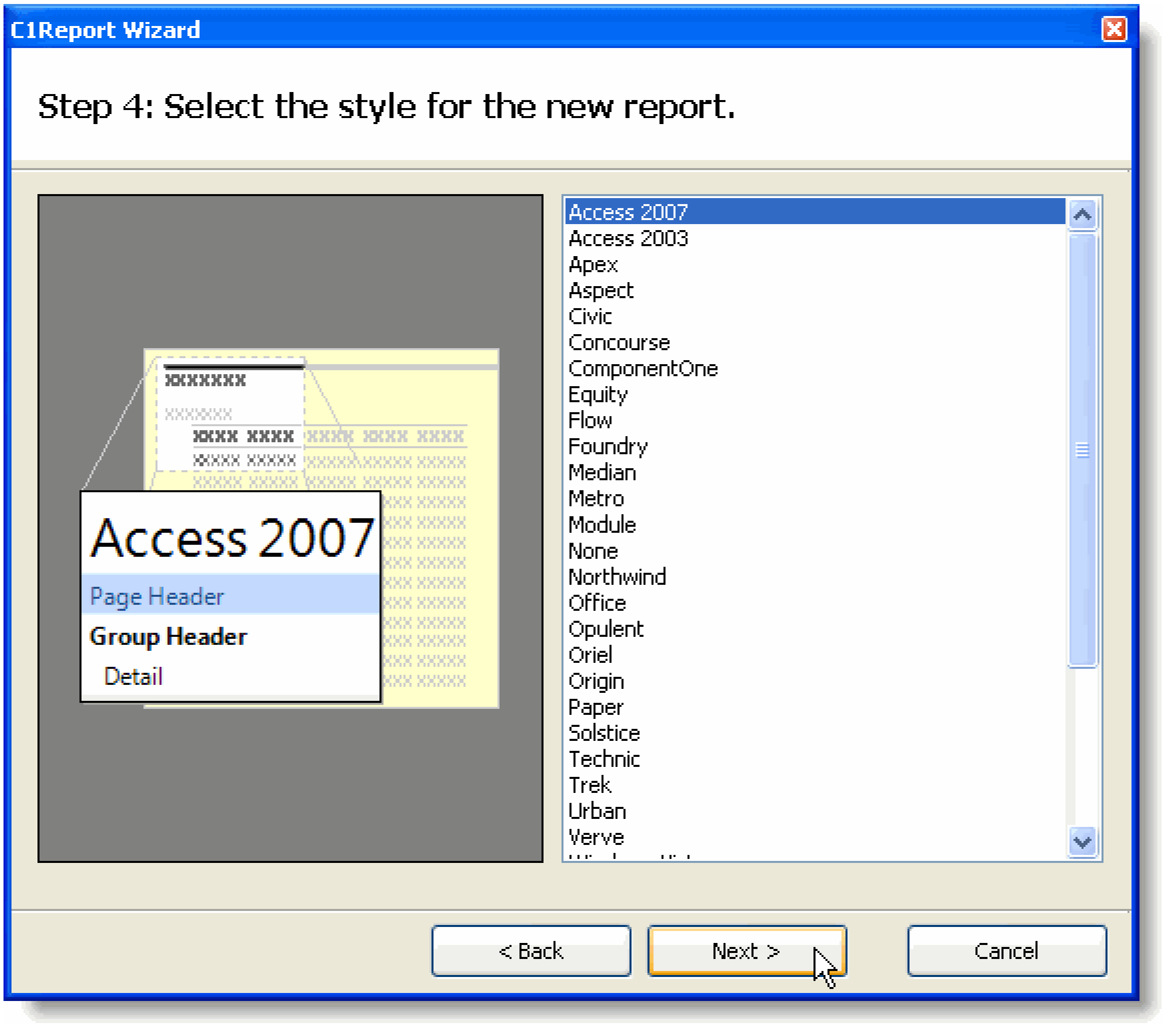
● Portrait ○ Landscape

☑ Adjust fields to fit page

4. **Select the style for the new report.**

- **Style Layouts**

  This page allows you to select the style that will be used in the new report. Like the previous page, it shows a preview to give you an idea of what each style looks like. Select the one that you like best (and remember, you can refine it and adjust the details later).



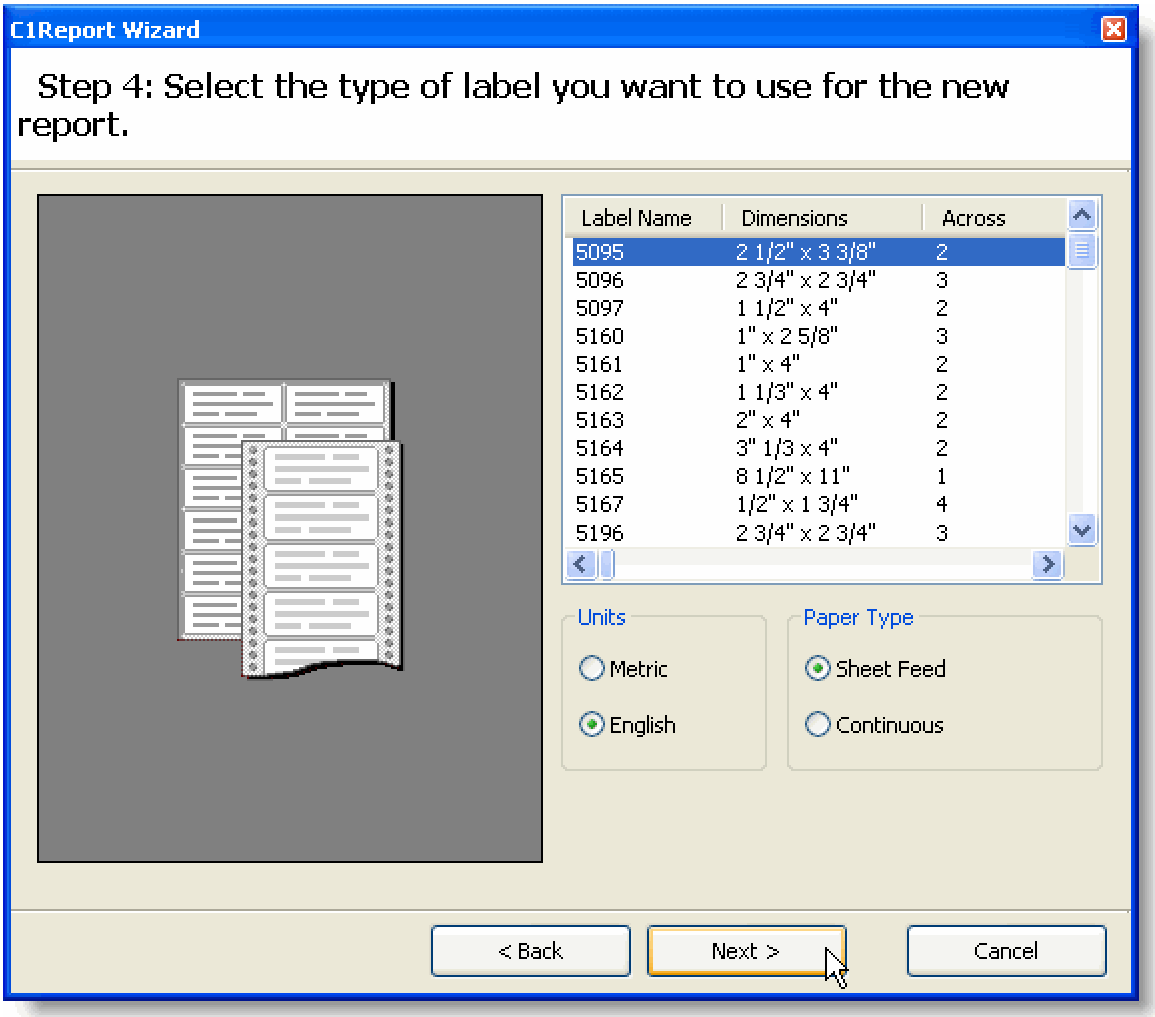
C1Report Wizard

Step 4: Select the style for the new report.

- Access 2007
- Access 2003
- Apex
- Aspect
- Civic
- Concourse
- ComponentOne
- Equity
- Flow
- Foundry
- Median
- Metro
- Module
- None
- Northwind
- Office
- Opulent
- Oriel
- Origin
- Paper
- Solstice
- Technic
- Trek
- Urban
- Verve

- **Label Layout Only**

  This page allows you to select the type of label you want to create. The Designer has over 170 predefined label types for you to choose from. The labels are divided into four groups, depending on whether they use metric or English measurements, and on the type of paper they use (sheets or continuous forms).



C1Report Wizard

Step 4: Select the type of label you want to use for the new report.

| Label Name | Dimensions | Across |
|---|---|---|
| 5095 | 2 1/2" x 3 3/8" | 2 |
| 5096 | 2 3/4" x 2 3/4" | 3 |
| 5097 | 1 1/2" x 4" | 2 |
| 5160 | 1" x 2 5/8" | 3 |
| 5161 | 1" x 4" | 2 |
| 5162 | 1 1/3" x 4" | 2 |
| 5163 | 2" x 4" | 2 |
| 5164 | 3" 1/3 x 4" | 2 |
| 5165 | 8 1/2" x 11" | 1 |
| 5167 | 1/2" x 1 3/4" | 4 |
| 5196 | 2 3/4" x 2 3/4" | 3 |

Units

○ Metric

● English

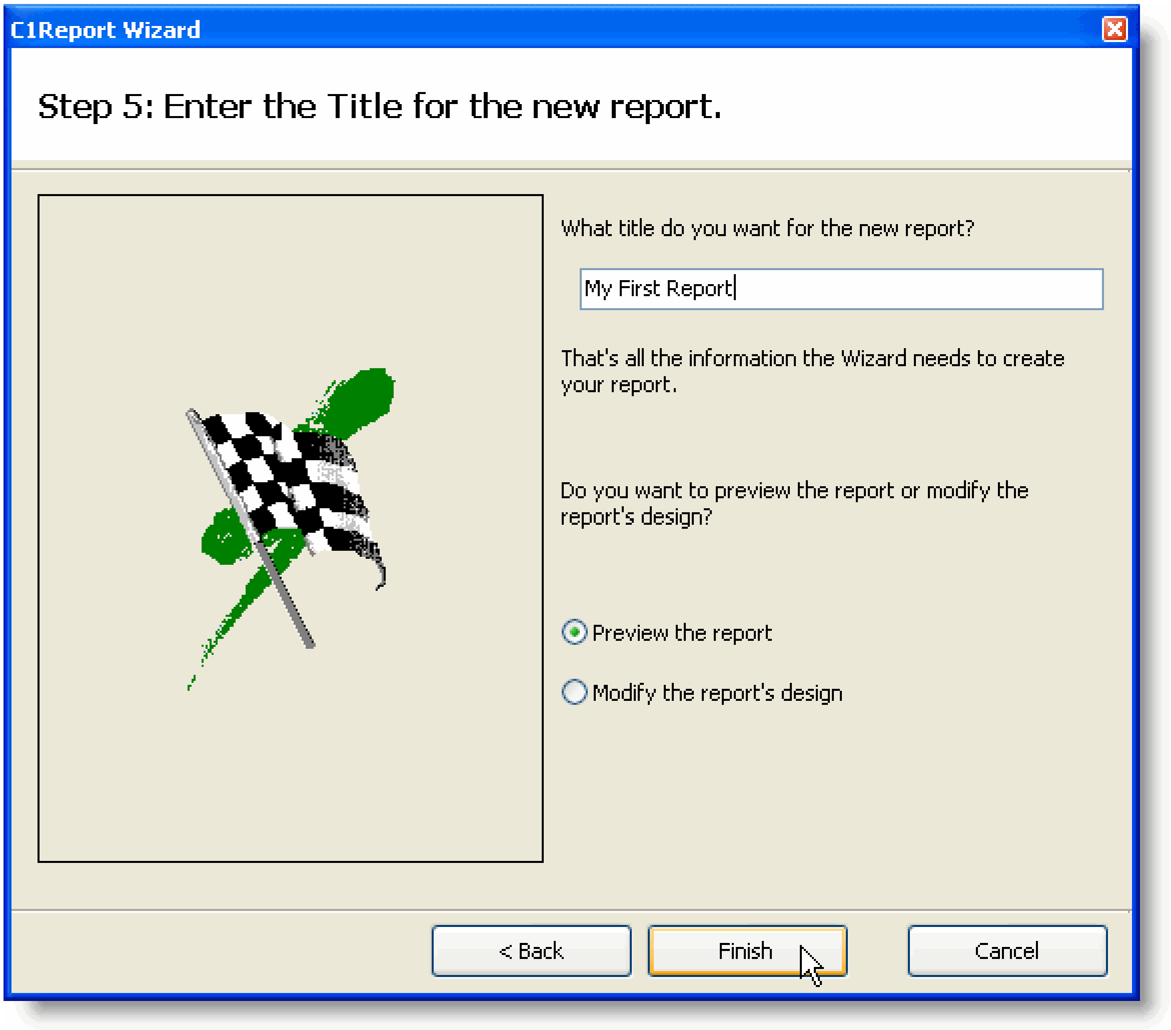
Paper Type

● Sheet Feed

○ Continuous

5. **Select a title for the new report.**

This last page allows you to select a title for the new report and to decide whether you would like to preview the new report right away or whether you would like to go into edit mode and start improving the design before previewing it.



Step 5: Enter the Title for the new report.

What title do you want for the new report?

My First Report

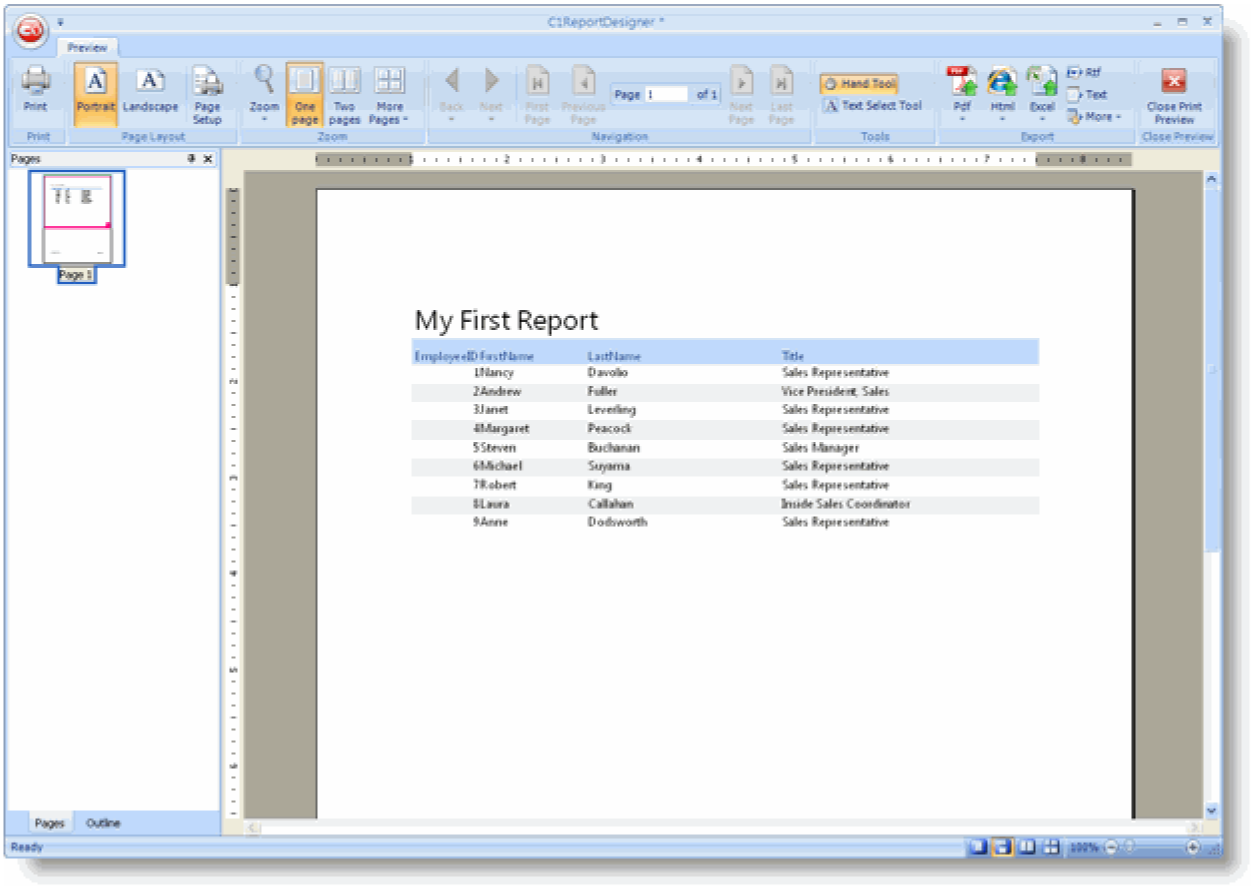
That's all the information the Wizard needs to create your report.

Do you want to preview the report or modify the report's design?

● Preview the report

○ Modify the report's design

If you choose to preview the report and click finish, you will immediately see the report in the preview pane of the **Designer**. It should look like this:



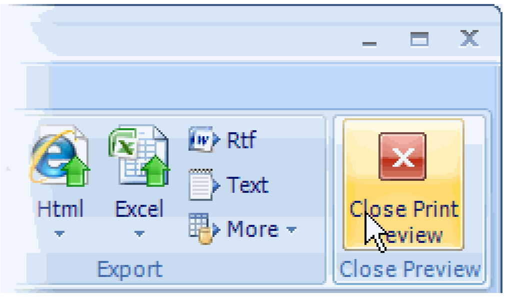
My First Report

| EmployeeID | FirstName | LastName | Title |
|---|---|---|---|
| 1 | Nancy | Davolio | Sales Representative |
| 2 | Andrew | Fuller | Vice President, Sales |
| 3 | Janet | Leverling | Sales Representative |
| 4 | Margaret | Peacock | Sales Representative |
| 5 | Steven | Buchanan | Sales Manager |
| 6 | Michael | Suyama | Sales Representative |
| 7 | Robert | King | Sales Representative |
| 8 | Laura | Callahan | Inside Sales Coordinator |
| 9 | Anne | Dodsworth | Sales Representative |

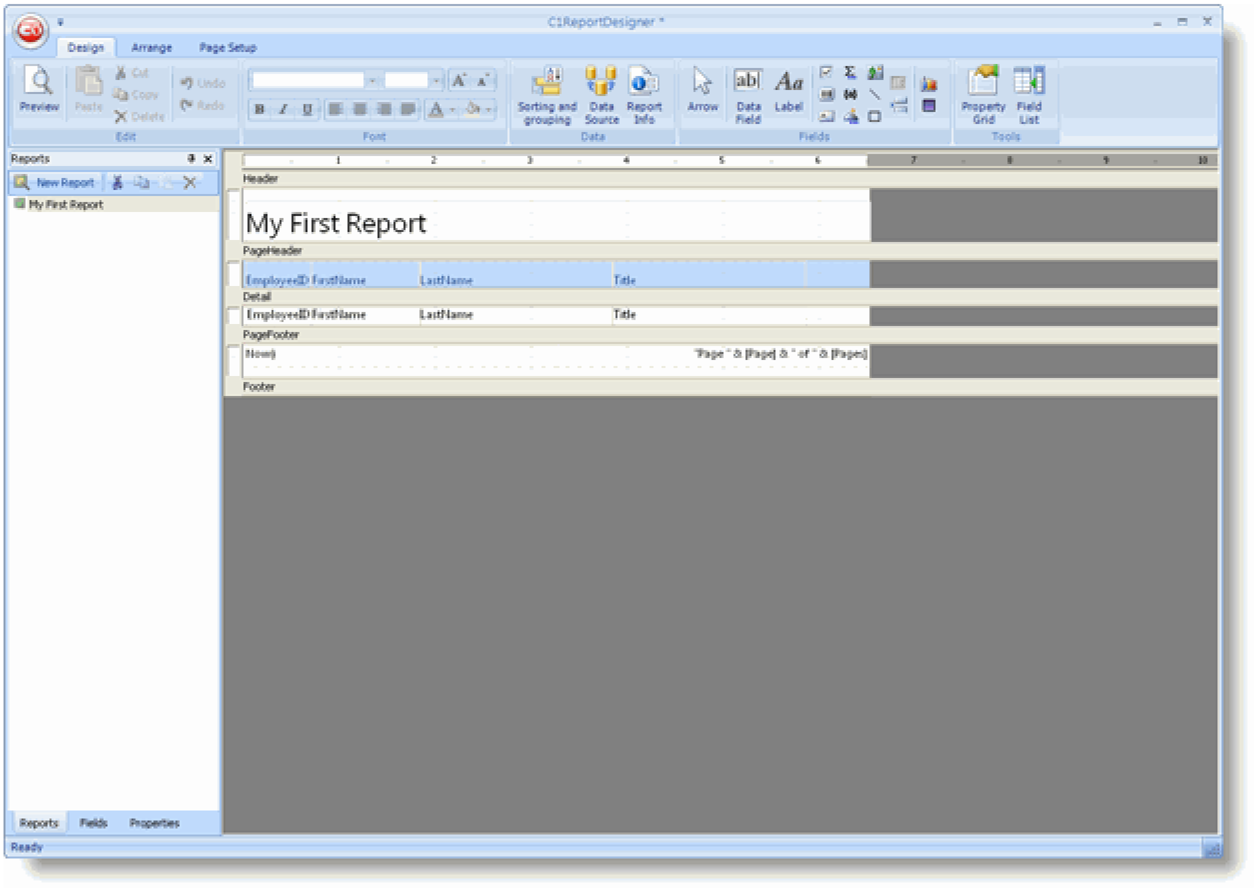
## Modifying the Report Layout

The report generated for you by the wizard is a good starting point, but you will usually need to adjust and enhance it to get exactly what you want. You can do this with the **C1ReportDesigner** application.

To start using the Designer, click the **Close Print Preview** button in the **Close Preview** group of the **Preview** tab:



The right pane of the main window switches from Review mode into Design mode, and it shows the controls and fields that make up the report:



The picture shows how the report is divided into sections (Header, Page Header, Detail, and so on). The sections contain fields that hold the labels, variables, and expressions that you want in the printed report. In this example, the Header section contains a label with the report title. The Page Header section contains labels that identify the fields in the Detail section, and the Page Footer section contains fields that show the current time, the page number and the total page count for the report.

The sections of the report determine what each page, group, and the beginning and end of the report look like. The following table describes where each section appears in the report and what it is typically used for:

| Section | Appears | Typically Contains |
|---|---|---|
| Report Header | Once per report | The report title and summary information for the whole report. |
| Page Header | Once per page | Labels that describe detail fields, and/or page numbers. |
| Group Header | Once per group | Fields that identify the current group, and possibly aggregate values for the group (for example, total, percentage of the grand total). |
| Detail | Once per record | Fields containing data from the source recordset. |
| Group Footer | Once per group | Aggregate values for the group. |
| Page Footer | Once per page | Page number, page count, date printed, report name. |
| Report Footer | Once per report | Summary information for the whole report. |

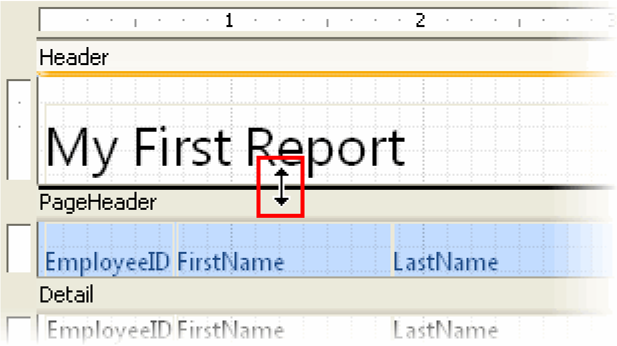
Note that you cannot add and delete sections directly. The number of groups determines the number of sections in a report. Every report has exactly five fixed sections (Report Header/Footer, Page Header/Footer, and Detail) plus two sections per group (a Header and Footer). You can hide sections that you don't want to display by setting their Visible property to **False**.

You can modify sections by changing their properties with the Properties window, or move and resize them with the mouse.

### Resizing a Section

To resize a section, select its border and with your mouse pointer drag to the position where you want it. The rulers on the left and on top of the design window show the size of each section (excluding the page margins). Note that you cannot make the section smaller than the height and width required to contain the fields in it. To reduce the size of a section beyond that, move or resize the fields in it first, then resize the Section.

To see how this works, move the mouse to the area between the bottom of the Page Header section and the gray bar on top of the Detail Section. The mouse cursor changes to show that you are over the resizing area. Click the mouse and drag the line down until the section is about twice its original height.



Release the mouse button and the section is resized.

## Enhancing the Report with Fields

To enhance your report, you can add fields (for example, lines, rectangles, labels, pictures, charts, and so on) to any Section. You can also modify the existing fields by changing their properties with the Properties window, or move and resize the fields with the mouse.

### Report Fields

The **Fields** group of the **Design** tab in the **C1ReportDesigner** application provides tools for creating report fields. This toolbar is enabled only in design mode. Each button creates a field and initializes its properties. For more information about the **Fields** group, see Fields Group. For more information on adding fields to your report, see Creating Report Fields.

### *Adding Chart Fields*

Chart fields are implemented using the **C1Chart** control.

> **Note:** You must deploy the **C1Chart** assembly with your application if you use charts.

To add a Chart field to your report, complete the following steps:

1. Open the report in the **C1ReportDesigner** application.

2. Click the **Add Chart Field** button in the toolbar, and mark the area in the report where the Chart should be displayed.
3. Then set the field properties as usual.

The only unusual aspect of Chart fields is that unlike most bound fields, they display multiple values. To select the data you want to display, set the Chart field's **Chart.DataX** and **Chart.DataY** properties. To format the values along the X and Y axis, set the **Chart.FormatX** and **Chart.FormatY** properties. You can also customize the chart appearance by setting other properties such as **Chart.ChartType**, **Chart.Palette**, and so on.

To create a new report with an embedded chart, use the **C1Report Wizard**. Complete the following steps:

1. **Select the data source for the new report.**
    a. Click the **Build a connection string** ellipses button **(…)**. The **Data Link Properties** dialog box appears.

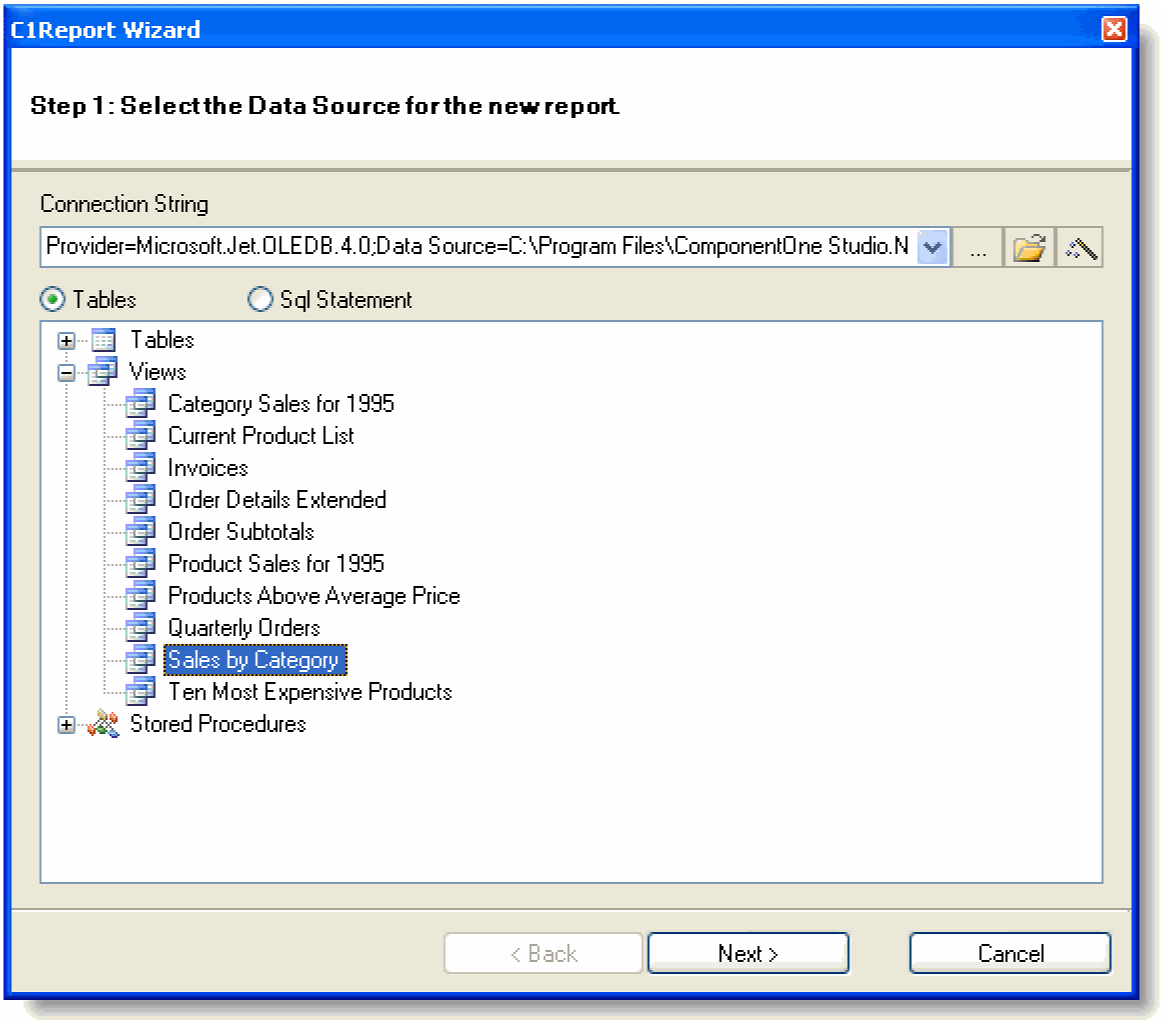
Select the **Provider** tab and then select **Microsoft Jet 4.0 OLE DB Provider** from the list.

Click **Next**.

In the Connection page, click the **ellipsis** button to browse for the **Nwind.mdb** database.

This is the standard Visual Studio Northwind database. By default, the database is installed in the ComponentOne Samples\Common directory.

Select the Tables radio button, and then select the **Sales by Category** view. The following image shows this step:



Step 1: Select the Data Source for the new report.

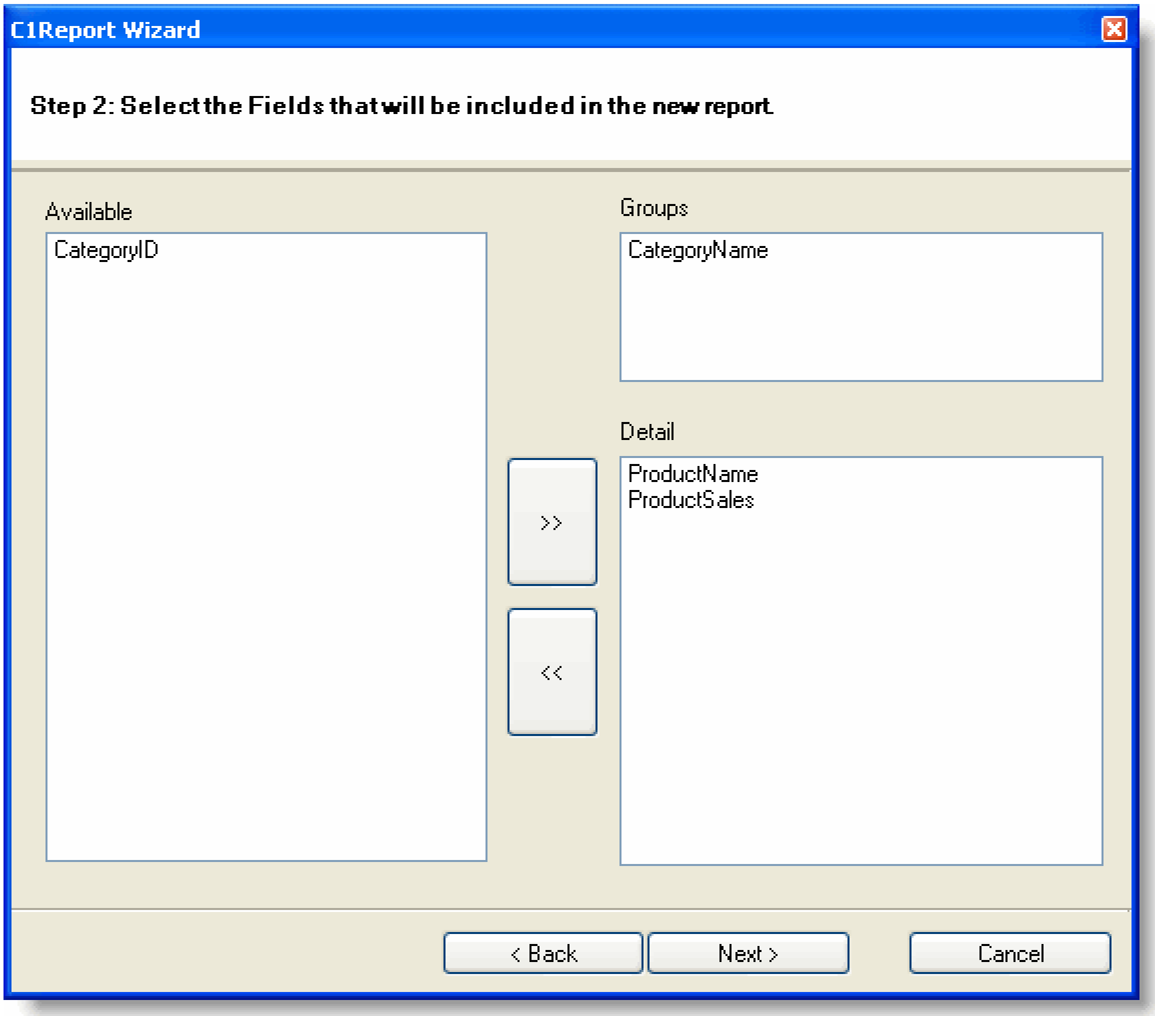
Connection String

Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\ComponentOne Studio.N

● Tables ○ Sql Statement

2. **Select the fields you want to display.**

   This example groups the data by Category and show ProductName and ProductSales in the Detail section of the report: To add groups and detail fields, with your mouse pointer drag them from the **Available** list on the left to the **Groups** or **Detail** lists on the right:



C1Report Wizard

Step 2: Select the Fields that will be included in the new report.

Available: CategoryID

Groups: CategoryName

Detail:
ProductName
ProductSales

Continue clicking **Next** until the wizard is done. The wizard creates the initial version of the report.

3. **Add the Chart to the Group Header section of the report.**

Charts usually make sense in the Group Header sections of a report, to summarize the information for the group. To add the chart to the Group Header section:

a. Click the **Close Print Preview** button in the **Close Preview** group to switch to Design mode to begin editing the report.

Expand the Group Header Section by performing a drag-and-drop operation with the section's borders.

Then click the **Add Chart Field** button in the **Fields** group of the **Design** tab and place the field in the report in the Group Header Section.

Resize the chart by clicking and dragging the chart field.

From the Properties window, set the **Chart.DataY** property to the name of the field that contains the values to be charted, in this case, **ProductSales**.

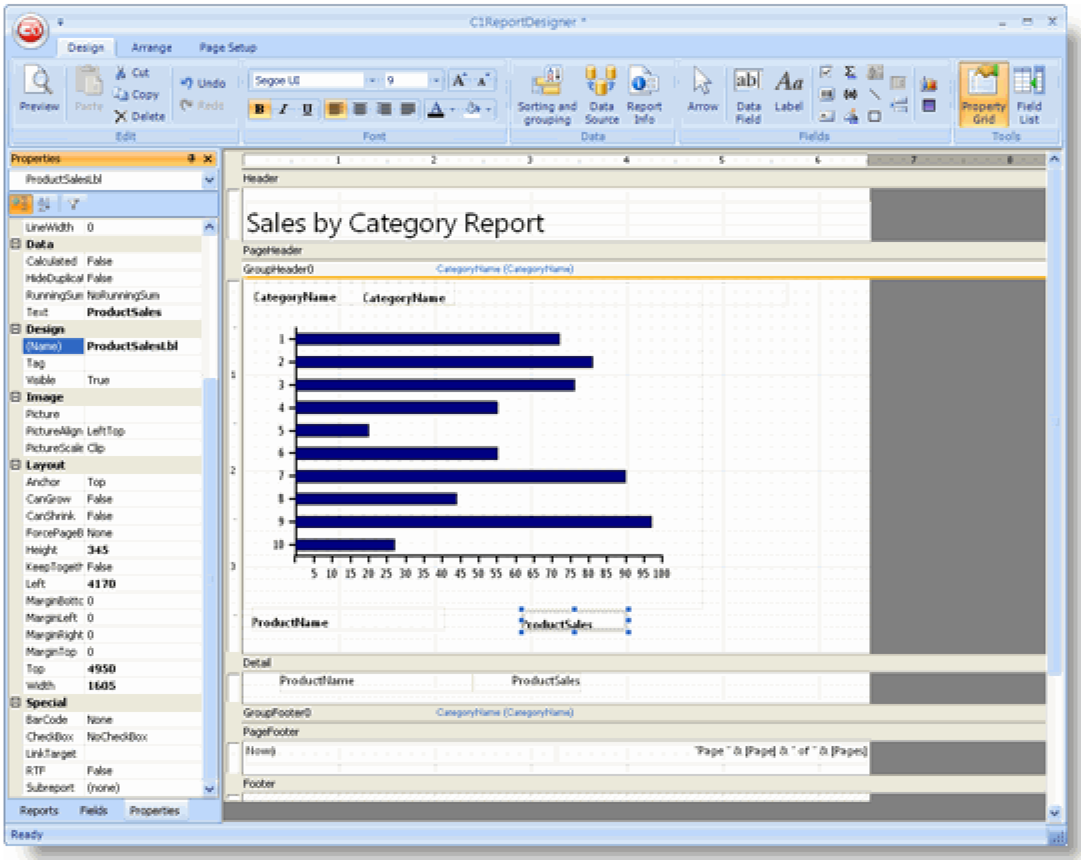
Note that the **Chart.DataY** property may specify more than one chart series. Just add as many fields or calculated expressions as you want, separating them with semicolons.

Also set the **Chart.DataX** property to the name of the field that contains the labels for each data point, in this case, **ProductName**.

From the Properties window, set the **Chart.FormatY** property to "#,###" to set the values along the axis to thousand-separated values.

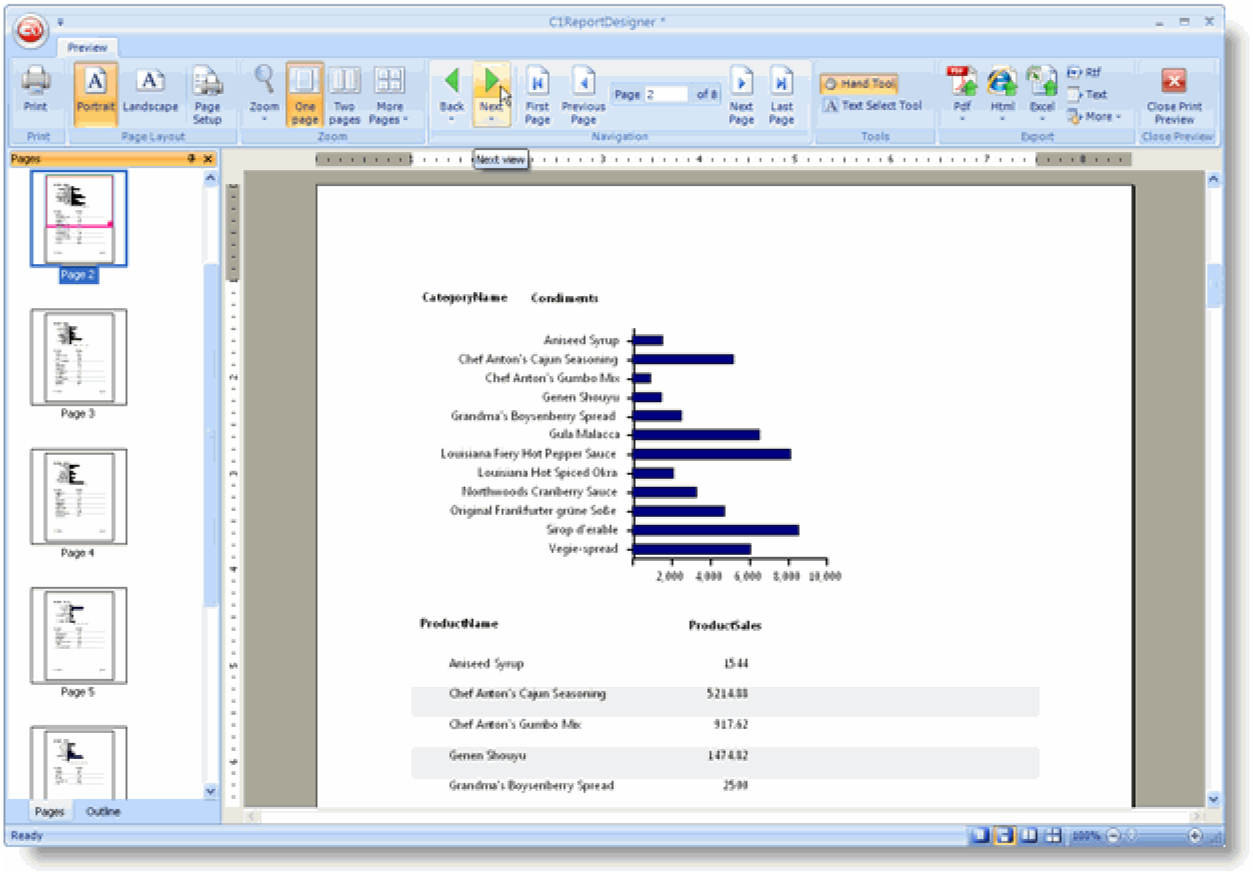
The Chart control will now display some sample data so you can see the effect of the properties that are currently set (the actual data is not available at design time). You may want to experiment changing the values of some properties such as **Chart.ChartType**, **Chart.DataColor**, and **Chart.GridLines**. You can also use the regular field properties such as **Font** and **ForeColor**.

Your report should look similar to the following report:



| | |
|---|---|
| LineWidth | 0 |
| **Data** | |
| Calculated | False |
| HideDuplicat | False |
| RunningSum | NoRunningSum |
| Text | **ProductSales** |
| **Design** | |
| (Name) | **ProductSalesLbl** |
| Tag | |
| Visible | True |
| **Image** | |
| Picture | |
| PictureAlign | LeftTop |
| PictureScale | Clip |
| **Layout** | |
| Anchor | Top |
| CanGrow | False |
| CanShrink | False |
| ForcePageB | None |
| Height | **345** |
| KeepTogeth | False |
| Left | **4170** |
| MarginBottc | 0 |
| MarginLeft | 0 |
| MarginRight | 0 |
| MarginTop | 0 |
| Top | **4950** |
| Width | **1605** |
| **Special** | |
| BarCode | None |
| CheckBox | NoCheckBox |
| LinkTarget | |
| RTF | False |
| Subreport | (none) |

Click the **Preview** button to see the report and click the **Next page** button to scroll through the report to view the Chart field for each group. The sample report should look like the following image:



CategoryName Condiments

| ProductName | ProductSales |
|---|---|
| Aniseed Syrup | 1544 |
| Chef Anton's Cajun Seasoning | 5214.88 |
| Chef Anton's Gumbo Mix | 917.62 |
| Genen Shouyu | 1474.82 |
| Grandma's Boysenberry Spread | 2500 |

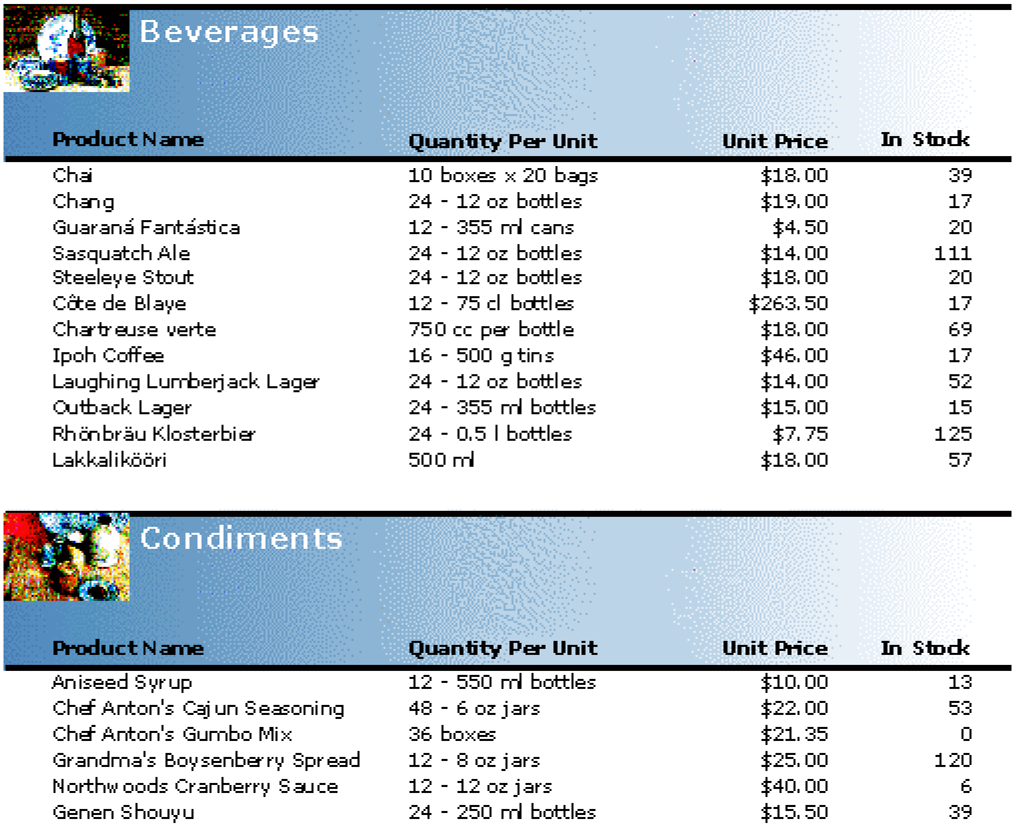
Note that the Report field is sensitive to its position in the report. Because it is in a Group Header section, it only includes the data within that group. If you place the Chart field in a Detail section, it will include all the data for the entire report. This is not useful because there will be one chart in each Detail section and they will all look the same. If you need more control over what data should be displayed in the chart, you can use the **DataSource** property in the chart field itself.

You can now save the report and use it in your WinForms and ASP.NET applications.

## *Adding Gradient Fields*

**Gradient** fields are much simpler than charts. They are mainly useful as a background feature to make other fields stand out.

The following image shows a report that uses gradient fields over the labels in the Group Header section:



Beverages

| Product Name | Quantity Per Unit | Unit Price | In Stock |
|---|---|---|---|
| Chai | 10 boxes x 20 bags | $18.00 | 39 |
| Chang | 24 - 12 oz bottles | $19.00 | 17 |
| Guaraná Fantástica | 12 - 355 ml cans | $4.50 | 20 |
| Sasquatch Ale | 24 - 12 oz bottles | $14.00 | 111 |
| Steeleye Stout | 24 - 12 oz bottles | $18.00 | 20 |
| Côte de Blaye | 12 - 75 cl bottles | $263.50 | 17 |
| Chartreuse verte | 750 cc per bottle | $18.00 | 69 |
| Ipoh Coffee | 16 - 500 g tins | $46.00 | 17 |
| Laughing Lumberjack Lager | 24 - 12 oz bottles | $14.00 | 52 |
| Outback Lager | 24 - 355 ml bottles | $15.00 | 15 |
| Rhönbräu Klosterbier | 24 - 0.5 l bottles | $7.75 | 125 |
| Lakkalikööri | 500 ml | $18.00 | 57 |

Condiments

| Product Name | Quantity Per Unit | Unit Price | In Stock |
|---|---|---|---|
| Aniseed Syrup | 12 - 550 ml bottles | $10.00 | 13 |
| Chef Anton's Cajun Seasoning | 48 - 6 oz jars | $22.00 | 53 |
| Chef Anton's Gumbo Mix | 36 boxes | $21.35 | 0 |
| Grandma's Boysenberry Spread | 12 - 8 oz jars | $25.00 | 120 |
| Northwoods Cranberry Sauce | 12 - 12 oz jars | $40.00 | 6 |
| Genen Shouyu | 24 - 250 ml bottles | $15.50 | 39 |

To create a similar gradient field, complete the following steps:

1. In Design mode of the Designer, select the **Add Gradient Field** button from the **Fields** group in the **Design** tab.
2. Move your mouse cursor (which has changed to a cross-hair) over the labels in the Group Header section and drag the field to the desired size.
3. To ensure that the field is behind the labels, right-click the gradient field and select **Send To Back**.
4. Then set the **Gradient.ColorFrom** and **Gradient.ColorTo** properties to **SteelBlue** and **White**, respectively.

Note that you may change the angle of the gradient field by setting the **Gradient.Angle** property another value (default value is 0).

### *Selecting, Moving, and Copying Fields*

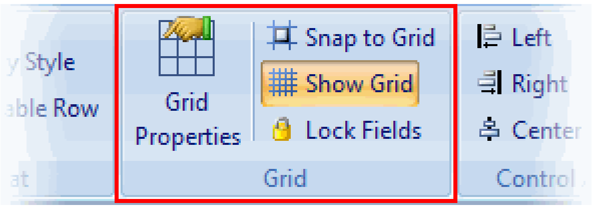
You can use the mouse to select fields in the **C1ReportDesigner** application as usual:

- Click a field to select it.
- Shift-click a field to toggle its selected state.
- Control-drag creates a copy of the selected fields.
- Click the empty area and drag your mouse pointer to select multiple fields.
- With your mouse pointer, drag field corners to resize fields.
- Double-click right or bottom field corners to auto size the field.

To select fields that intersect vertical or horizontal regions of the report, click and drag the mouse on the rulers along the edges of the Designer. If fields are small or close together, it may be easier to select them by name. You can select fields and sections by picking them from the drop-down list above the Properties window.

### Show a grid

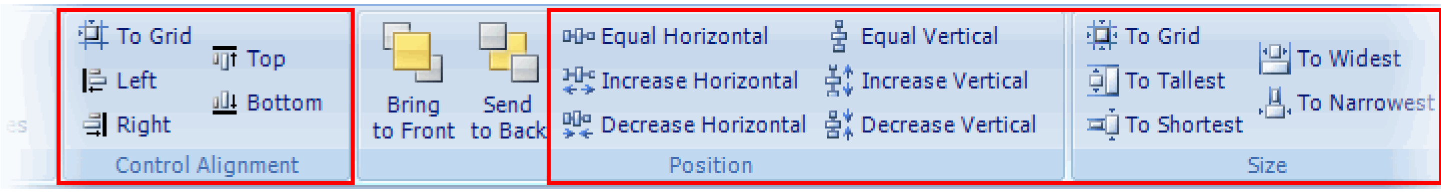
The **Snap to grid** and **Show grid** buttons located in the **Grid** group in the **Appearance** tab provide a grid that helps position controls at discrete positions. While the grid is on, the top left corner of the fields will snap to the grid when you create or move fields. You can change the grid units (English or metric) by clicking the **Application** button and selecting **Options** from the menu.



### Lock fields

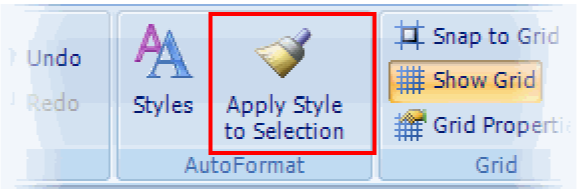
After you have placed fields in the desired positions, you can lock them to prevent inadvertently moving them with the mouse or keyboard. Use the **Lock Fields** button to lock and unlock the fields.

### Format fields



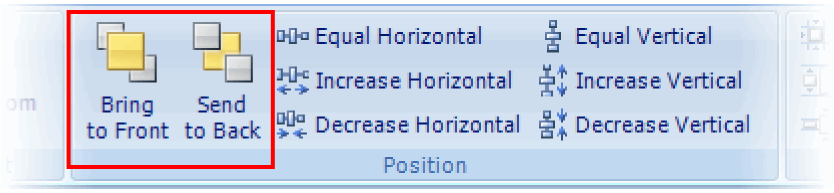
When multiple fields are selected, you can use the buttons on the **Control Alignment**, **Position**, and **Size** groups of the **Appearance** tab to align, resize, and space them. When you click any of these buttons, the last field in the selection is used as a reference and the settings are applied to the remaining fields in the selection.

### Apply styles



The **Apply Style to Selection** button applies the style of the reference field to the entire selection. The style of a field includes all font, color, line, alignment, and margin properties. You can use the Properties window to set the value of individual properties to the entire selection.

### Determine order for overlapping fields



If some fields overlap, you can control their z-order using the **Bring to Front/Send to Back** buttons in the **Position** group. This determines which fields are rendered before (behind) the others.

**Move fields using the keyboard**

The **C1ReportDesigner** application also allows you to select and move fields using the keyboard:

- Use the TAB key to select the next field.
- Use SHIFT-TAB to select the previous field.
- Use the arrow keys to move the selection one pixel at a time (or shift arrow to by 5 pixels).
- Use the DELETE key to delete the selected fields.
- When a single field is selected, you can type into it to set the **Text** property.

### *Changing Field, Section, and Report Properties*

Once an object is selected, you can use the Properties window to edit its properties.

When one or more fields are selected, the Properties window shows property values that all fields have in common, and leaves the other properties blank. If no fields are selected and you click on a section (or on the bar above a section), the **Section** properties are displayed. If you click the gray area in the background, the **Report** properties are displayed.

To see how this works, click the label in the Header section and change its Font and ForeColor properties. You can also change a field's position and dimensions by typing new values for the Left, Top, Width, and Height properties.

The Properties window expresses all measurements in *twips* (the native unit used by C1Report), but you can type in values in other units (in, cm, mm, pix, pt) and they will be automatically converted into *twips*. For example, if you set the field's Height property to **0.5in**, the Properties window will convert it into 720 *twips*.

### *Changing the Data Source*

The data source is defined by the ConnectionString, RecordSource, and Filter properties. These are regular **Report** properties and may be set one of the following ways:

- From the Properties window, select the **ellipsis** button next to the **DataSource** property (if you click the gray area in the background, the **Report** properties are displayed).

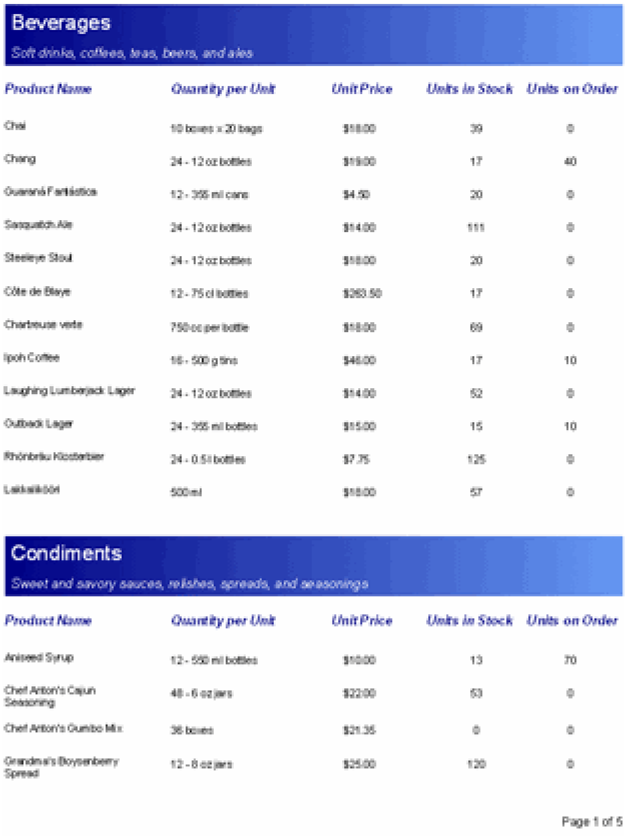
  OR

- Click the **DataSource** button in the **Data** group of the **Design** tab to open the **Select a Data Source** dialog box that allows you to set the ConnectionString and RecordSource properties directly.

## Creating a Master-Detail Report Using Subreports

Subreports are regular reports contained in a field in another report (the main report). Subreports are usually designed to display detail information based on a current value in the main report, in a master-detail scenario.

In the following example, the main report contains categories and the subreport in the Detail section contains product details for the current category:



**Beverages**

*Soft drinks, coffees, teas, beers, and ales*

| *Product Name* | *Quantity per Unit* | *Unit Price* | *Units in Stock* | *Units on Order* |
|---|---|---|---|---|
| Chai | 10 boxes x 20 bags | $18.00 | 39 | 0 |
| Chang | 24 - 12 oz bottles | $19.00 | 17 | 40 |
| Guaraná Fantástica | 12 - 355 ml cans | $4.50 | 20 | 0 |
| Sasquatch Ale | 24 - 12 oz bottles | $14.00 | 111 | 0 |
| Steeleye Stout | 24 - 12 oz bottles | $18.00 | 20 | 0 |
| Côte de Blaye | 12 - 75 cl bottles | $263.50 | 17 | 0 |
| Chartreuse verte | 750 cc per bottle | $18.00 | 69 | 0 |
| Ipoh Coffee | 16 - 500 g tins | $46.00 | 17 | 10 |
| Laughing Lumberjack Lager | 24 - 12 oz bottles | $14.00 | 52 | 0 |
| Outback Lager | 24 - 355 ml bottles | $15.00 | 15 | 10 |
| Rhönbräu Klosterbier | 24 - 0.5 l bottles | $7.75 | 125 | 0 |
| Lakkalikööri | 500 ml | $18.00 | 57 | 0 |

**Condiments**

*Sweet and savory sauces, relishes, spreads, and seasonings*

| *Product Name* | *Quantity per Unit* | *Unit Price* | *Units in Stock* | *Units on Order* |
|---|---|---|---|---|
| Aniseed Syrup | 12 - 550 ml bottles | $10.00 | 13 | 70 |
| Chef Anton's Cajun Seasoning | 48 - 6 oz jars | $22.00 | 53 | 0 |
| Chef Anton's Gumbo Mix | 36 boxes | $21.35 | 0 | 0 |
| Grandma's Boysenberry Spread | 12 - 8 oz jars | $25.00 | 120 | 0 |

Page 1 of 5

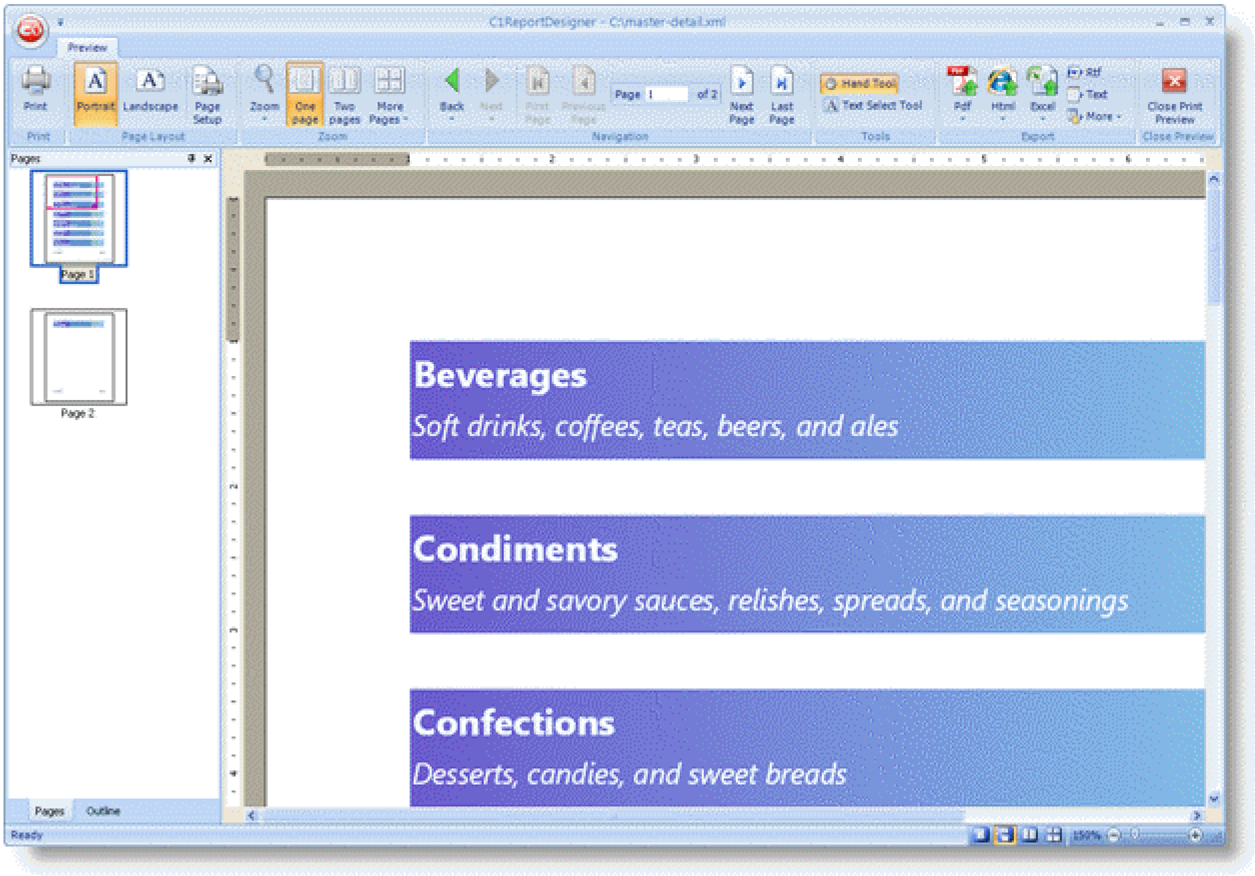
To create a master-detail report based on the **Categories** and **Products** tables, you need to create a Categories report (master view) and a Products report (details view).

### Step 1: Create the master report

1. Create a basic report definition using the **C1Report Wizard**.
   a. Select the **Categories** table from the Northwind database (**Nwind.mdb** located in the ComponentOne Samples\Common folder).

Include the **CategoryName** and **Description** fields in the report.

2. In the **C1ReportDesigner** application, click the **Close Print Preview** button to begin editing the report.
3. Set the Page Header and Header section's **Visible** property to **False**.
4. In the Detail section, select the **DescriptionCtl** and move it directly below the **CategoryNameCtl**.
5. Use the Properties window to change the Appearance settings (Font and ForeColor). Note that for this example, a **Gradient** field was added to the Detail Section. For information on **Gradient** fields, see Adding Gradient Fields.
6. Select the **Preview** button, the Categories report should now look similar to the following image:



Beverages
Soft drinks, coffees, teas, beers, and ales

Condiments
Sweet and savory sauces, relishes, spreads, and seasonings

Confections
Desserts, candies, and sweet breads

### Step 2: Create the detail report

1. In the **C1ReportDesigner** application, click the **New Report** button to create a basic report definition using the **C1Report Wizard**.

   a. Select the **Products** table from the Northwind database.

Include the following fields in the report: **ProductName**, **QuantityPerUnit**, **UnitPrice**, **UnitsInStock**, and **UnitsOnOrder**.

2. In the Report Designer, click the **Close Print Preview** button to begin editing the report.

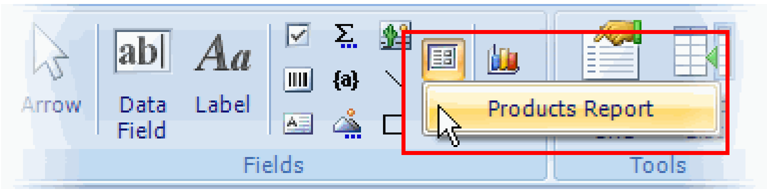
   a. Set the Page Header and Header section's **Visible** property to **False**.

In the Detail section, arrange the controls so that they are aligned with the heading labels. Use the Properties window to change the Appearance settings.

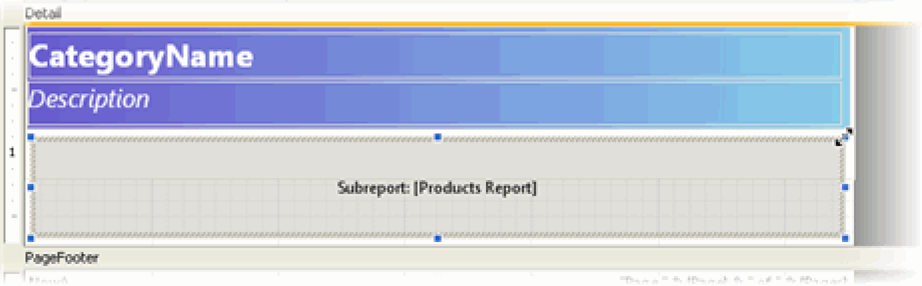
### Step 3: Create the subreport field

The **C1ReportDesigner** application now has two separate reports**, Categories Report** and **Products Report**. The next step is to create a subreport:

1. From the Reports list in the Designer, select **Categories Report** (master report).
2. In design mode, from the click the **Add Subreport** button in the **Fields** group of the **Design** tab and select **Products Report** from the drop-down menu.



3. In the Detail section of your report, click and drag the mouse pointer to make the field for the subreport:

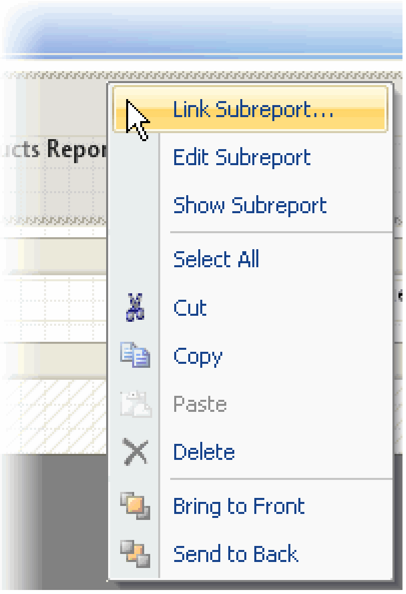


### Step 4: Link the subreport to the master report

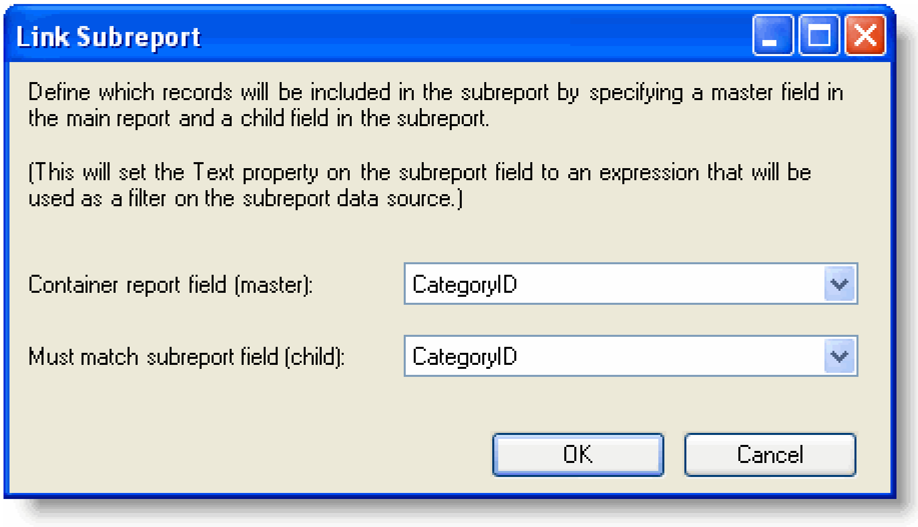
The master-detail relationship is controlled by the Text property of the subreport field. This property should contain an expression that evaluates into a filter condition that can be applied to the subreport data source.

The Report Designer can build this expression automatically for you. Complete the following steps:

1. Right-click the subreport field and select **Link Subreport** from the menu.



2. A dialog box appears that allows you to select which fields should be linked.



Link Subreport

Define which records will be included in the subreport by specifying a master field in the main report and a child field in the subreport.

(This will set the Text property on the subreport field to an expression that will be used as a filter on the subreport data source.)

Container report field (master): CategoryID

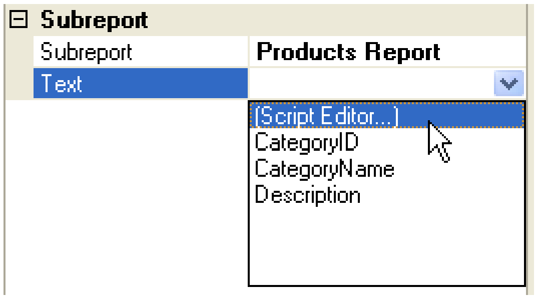
Must match subreport field (child): CategoryID

3. Once you make a selection and click **OK**, the Report Designer builds the link expression and assigns it to the Text property of the subreport field. In this case, the expression is:

```
"[CategoryID] = '" & [CategoryID] & "'"
```

Alternatively, you can also link the subreport to the master report by completing the following steps:

1. From the Properties window, click the Text property of the subreport field and select **Script Editor** from the drop-down list.



Subreport

Subreport: Products Report

Text:

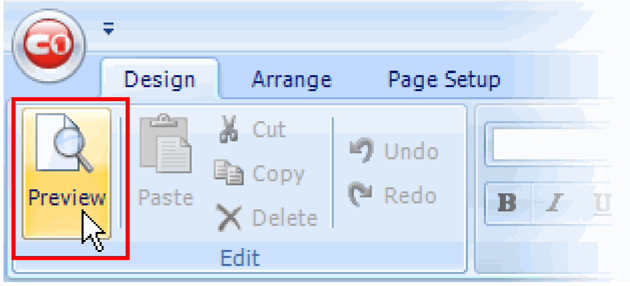
2. Enter the following expression in the VBScript Editor:

```
"[CategoryID] = '" & [CategoryID] & "'"
```

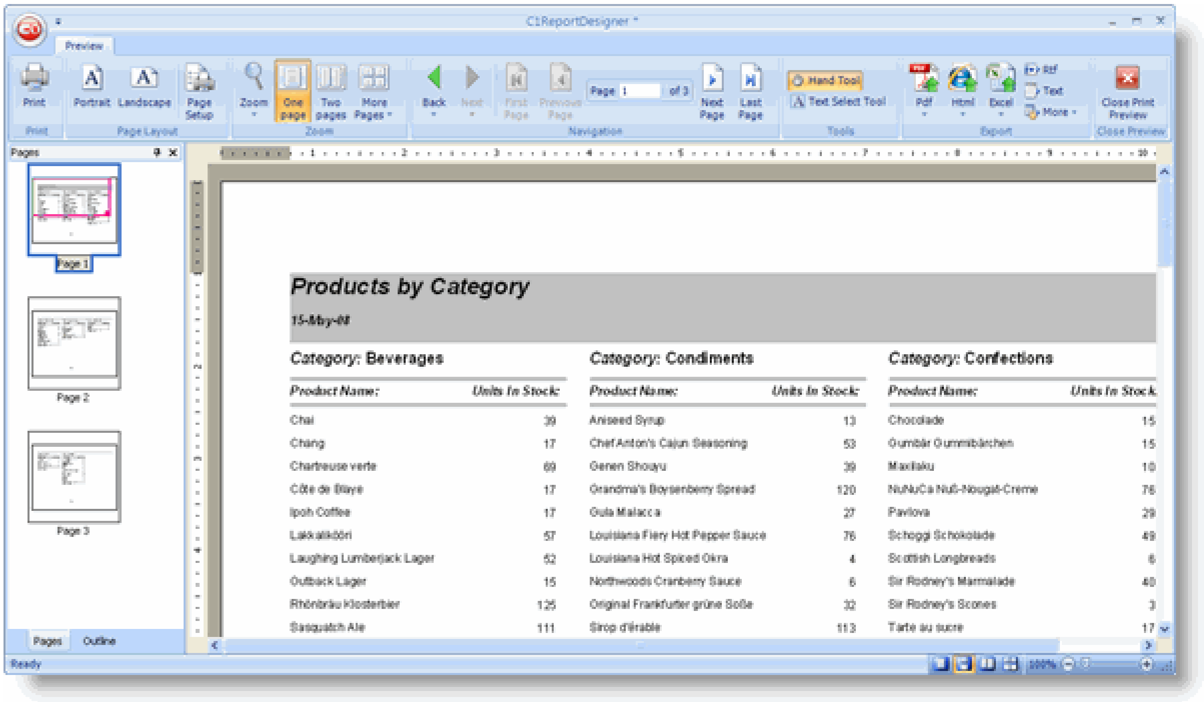
3. Click **OK** to close the VBScript Editor and build the expression.

## Previewing and Printing a Report

To preview a report, select the report to view from the Reports list on the left pane of the Designer window and click the **Preview** button, which appears on each Ribbon tab:



Alternatively, you can select **View | Preview** from the menu. The report is displayed on the right pane, as shown in the following screen shot:



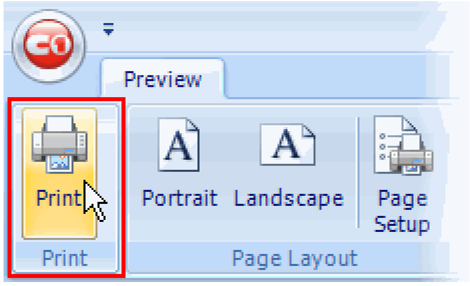
**Products by Category**

15-May-08

| Category: Beverages | | Category: Condiments | | Category: Confections | |
|---|---|---|---|---|---|
| *Product Name:* | *Units In Stock:* | *Product Name:* | *Units In Stock:* | *Product Name:* | *Units In Stock* |
| Chai | 39 | Aniseed Syrup | 13 | Chocolade | 15 |
| Chang | 17 | Chef Anton's Cajun Seasoning | 53 | Gumbär Gummibärchen | 15 |
| Chartreuse verte | 69 | Genen Shouyu | 39 | Maxilaku | 10 |
| Côte de Blaye | 17 | Grandma's Boysenberry Spread | 120 | NuNuCa Nuß-Nougat-Creme | 76 |
| Ipoh Coffee | 17 | Gula Malacca | 27 | Pavlova | 29 |
| Lakkalikööri | 57 | Louisiana Fiery Hot Pepper Sauce | 76 | Schoggi Schokolade | 49 |
| Laughing Lumberjack Lager | 52 | Louisiana Hot Spiced Okra | 4 | Scottish Longbreads | 6 |
| Outback Lager | 15 | Northwoods Cranberry Sauce | 6 | Sir Rodney's Marmalade | 40 |
| Rhönbräu Klosterbier | 125 | Original Frankfurter grüne Soße | 32 | Sir Rodney's Scones | 3 |
| Sasquatch Ale | 111 | Sirop d'érable | 113 | Tarte au sucre | 17 |

The main window has a preview navigation toolbar, with buttons that let you page through the document and select the zoom mode.

At this point, you can print the report by clicking the **Print** button:



## Exporting and Publishing a Report

Instead of printing the report, you may want to export it into a file and distribute it electronically to your clients or co-workers. The Designer supports the following export formats:

| Format | Description |
|---|---|
| Paged HMTL (*.htm) | Creates one HTML file for each page in the report. The HTML pages contain links that let the user navigate the report. |
| Drill-Down HTML (*.htm) | Creates a single HTML file with sections that can be collapsed and expanded by the user by clicking on them. |
| Plain HMTL (*.htm) | Creates a single HTML file with no drill-down functionality. |
| PDF with system fonts (*.pdf) | Creates a PDF file that can be viewed on any computer equipped with Adobe's Acrobat viewer or browser plug-ins. |
| PDF with embedded fonts (*.pdf) | Creates a PDF file with embedded font information for extra portability. This option significantly increases the size of the PDF file. |
| RTF (*.rtf) | Creates an RTF file that can be opened by most popular word processors (for example, Microsoft Word, WordPad). |
| RTF with fixed positioning (*.rtf) | Creates an RTF file with fixed positioning that can be opened by most popular word processors (for example, Microsoft Word, WordPad). |
| Microsoft Excel 97 (*.xls) | Creates an XLS file that can be opened by Microsoft Excel. |
| Microsoft Excel 2007/2010 Open XML (*.xlsx) | Creates an XLS file that can be opened by Microsoft Excel 2007 and later. |
| TIFF (*.tif) | Creates a multi-page TIFF (Tag Image File Format) file. |
| Text (*.txt) | Creates a plain text file. |
| Single Page Text (*.txt) | Creates a single-page plain text file. |
| Compressed Metafile (*.txt) | Creates a compressed metafile text file. |

To create an export file, select **File | Export** from the menu and use the **File Save** dialog box to select the type of file you want to create, specifying its name and location.

**Note:** When a document is exported to the RTF or the DOCX formats with the "preserve pagination" option selected, text is placed in text boxes and the ability to reflow text in the resulting document may be limited.

## Managing Report Definition Files

A report definition file may contain several reports. Occasionally, you may want to copy or move a report from one file to another.

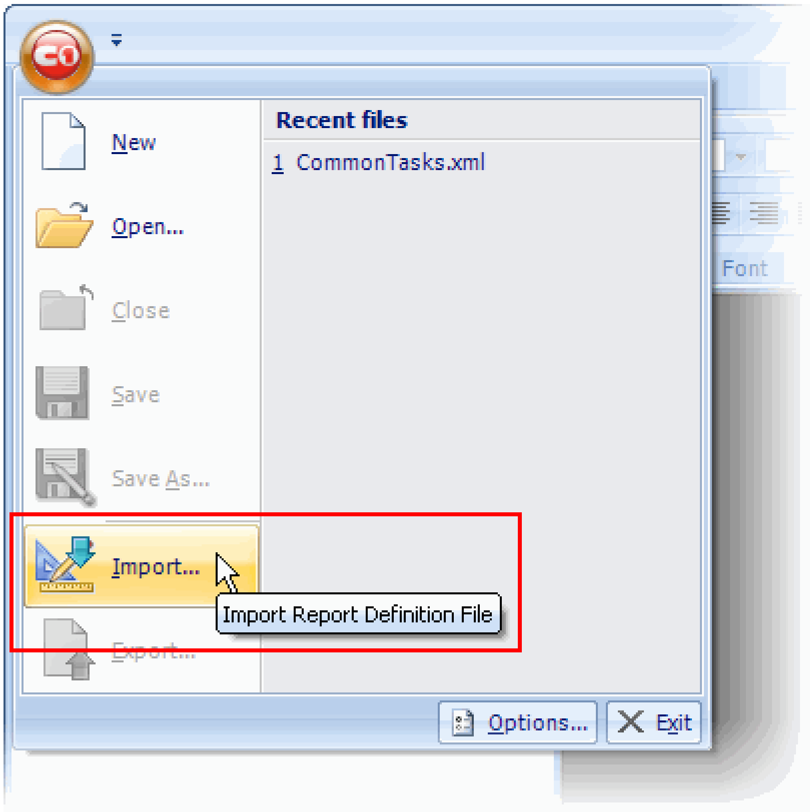
To move a report from one file to another, open two instances of the **C1ReportDesigner** application and drag the report from one instance to the other. If you hold down the CTRL key while doing this, the report will be copied. Otherwise, it will be moved.

You can also copy a report within a single file. This creates a new copy of the report, which is a good way to start designing a new report that is similar to an existing one.

Note that the report definition files are saved in XML, so you can also edit and maintain them using any text editor.

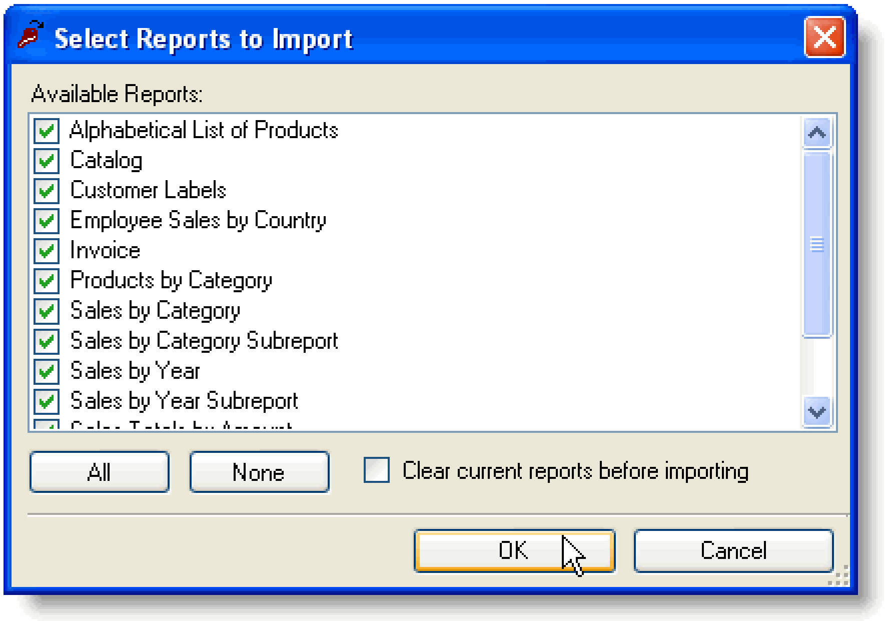
## Importing Microsoft Access Reports

One of the most powerful features of the **C1ReportDesigner** application is the ability to import reports created with Microsoft Access. **This feature requires Access to be installed on the computer.** Once the report is imported into the Designer, Access is no longer required.



To import reports from an Access file, click the **Application** button and select **Import** from the menu. A dialog box prompts you for the name of the file you want to import.

Select a Microsoft Access file (MDB or ADP) and the Designer scans the file and shows a dialog box where you can select which reports you would like to import:



Available Reports:

☑ Alphabetical List of Products
☑ Catalog
☑ Customer Labels
☑ Employee Sales by Country
☑ Invoice
☑ Products by Category
☑ Sales by Category
☑ Sales by Category Subreport
☑ Sales by Year
☑ Sales by Year Subreport
☑ Sales Totals by Amount

□ Clear current reports before importing

The dialog box also allows you to specify if the Designer should clear all currently defined reports before starting the import process.

The import process handles most elements of the source reports, with a few exceptions:

- **Event handler code**

  Access reports can use VBA, macros and forms to format the report dynamically. C1Report can do the same things, but it only uses VBScript. Because of this, all report code needs to be translated manually.

- **Form-oriented field types**

  Access reports may include certain fields that are not handled by the Designer's import procedure. The following field types are **not** supported: Chart, CommandButton, ToggleButton, OptionButton, OptionGroup, ComboBox, ListBox, TabCtl, and CustomControl.

- **Reports that use VBScript reserved words**

  Because Access does not use VBScript, you may have designed reports that use VBScript reserved words as identifiers for report objects or dataset field names. This causes problems when the VBScript engine tries to evaluate the expression, and prevents the report from rendering correctly.

  Reserved words you shouldn't use as identifiers include **Date**, **Day**, **Hour**, **Length**, **Minute**, **Month**, **Second**, **Time**, **TimeValue**, **Value**, **Weekday**, and **Year**. For a complete list, please refer to a VBScript reference.

- **Reports that sort dates by quarter (or weekday, month of the year, and so on)**

  C1Report uses the ADO.NET dataset Sort property to sort groups. This property sorts datasets according to field values only and does not take expressions. (Note that you can group according to an arbitrary expression, but you can't sort.) An Access report that sorts groups by quarter will sort them by date after it is imported. To fix this, you have two options: create a field that contains the value for the expression you want to sort on or change the SQL statement that creates the dataset and perform the sorting that way.

These limitations affect a relatively small number of reports, but you should preview all reports after importing them, to make sure they still work correctly.

### Importing the Nwind.mdb File

To illustrate how the Designer fares in a real-life example, try importing the Nwind.mdb file. It contains the following 13 reports. (The Nwind.xml file that ships with C1Report already contains all the following modifications.)

1. **Alphabetical List of Products**

   No action required.

2. **Catalog**

   No action required.

3. **Customer Labels**

   No action required.

4. **Employee Sales by Country**

   This report contains code which needs to be translated manually. The following code should be assigned to the Group 1 Header OnPrint property:

   - Visual Basic

     ```
     If SalespersonTotal > 5000 Then
       ExceededGoalLabel.Visible = True
       SalespersonLine.Visible = True
     Else
       ExceededGoalLabel.Visible = False
       SalespersonLine.Visible = False
     End If
     ```

   - C#

     ```
     if (SalespersonTotal > 5000)
     {
       ExceededGoalLabel.Visible = true;
       SalespersonLine.Visible = true;
     } else {
       ExceededGoalLabel.Visible = false;
       SalespersonLine.Visible = false;
     }
     ```

5. **Invoice**

   No action required.

6. **Products by Category**

   No action required.

7. **Sales by Category**

   This report contains a Chart control that is not imported. To add a chart to your report, you could use an unbound picture field, then write a VB event handler that would create the chart and assign it to the field as a picture.

8. **Sales by Category Subreport**

   No action required.

9. **Sales by Year**

   This report contains code and references to a Form object which need to be translated manually. To replace the Form object, edit the RecordSource property to add a [Show Details] parameter:

   - Visual Basic

     ```
     PARAMETERS (Beginning Date) DateTime 1/1/1994,
       (Ending Date) DateTime 1/1/2001,
     ```

```
  (Show Details) Boolean False; ...
```

- C#

```
PARAMETERS [Beginning Date] DateTime 1/1/1994,
  [Ending Date] DateTime 1/1/2001,
  [Show Details] Boolean False; ...
```

Use the new parameter in the report's **OnOpen** event:

- Visual Basic

```
Dim script As String = _
  "bDetails = [Show Details]" & vbCrLf & _
  "Detail.Visible = bDetails" & vbCrLf & _
  "[Group 0 Footer].Visible = bDetails" & vbCrLf & _
  "DetailsLabel.Visible = bDetails" & vbCrLf & _
  "LineNumberLabel2.Visible = bDetails" & vbCrLf & _
  "Line15.Visible = bDetails" & vbCrLf & _
  "SalesLabel2.Visible = bDetails" & vbCrLf & _
  "OrdersShippedLabel2.Visible = bDetails" & vbCrLf & _
  "ShippedDateLabel2.Visible = bDetails" & vbCrLf & _
  "Line10.Visible = bDetails"
c1r.Sections.Detail.OnPrint = script
```

- C#

```
string script = "bDetails = [Show Details]" +
  "Detail.Visible = bDetails\r\n" +
  "[Group 0 Footer].Visible = bDetails\r\n" +
  "DetailsLabel.Visible = bDetails\r\n" +
  "LineNumberLabel2.Visible = bDetails\r\n" +
  "Line15.Visible = bDetails\r\n" +
  "SalesLabel2.Visible = bDetails\r\n" +
  "OrdersShippedLabel2.Visible = bDetails\r\n" +
  "ShippedDateLabel2.Visible = bDetails\r\n" +
  "Line10.Visible = bDetails";
c1r.Sections.Detail.OnPrint = script;
```

Finally, two more lines of code need to be translated:

- Visual Basic

```
Sections ("Detail").OnPrint = _
  "PageHeader.Visible = True"
Sections("Group 0 Footer).OnPrint = _
  "PageHeader.Visible = False"
```

- C#

```
Sections ("Detail").OnPrint =
  "PageHeader.Visible = true";
Sections("Group 0 Footer).OnPrint =
  "PageHeader.Visible = false";
```

### 10. Sales by Year Subreport

No action required.

### 11. Sales Totals by Amount

This report contains code that needs to be translated manually. The following code should be assigned to the Page Header OnPrint property:

- Visual Basic

```
PageTotal = 0
```

- C#
  ```
  PageTotal = 0;
  ```

The following code should be assigned to the Detail OnPrint property:

- Visual Basic
  ```
  PageTotal = PageTotal + SaleAmount
  HiddenPageBreak.Visible = (Counter = 10)
  ```

- C#
  ```
  PageTotal = PageTotal + SaleAmount;
  HiddenPageBreak.Visible = (Counter = 10);
  ```

12. **Summary of Sales by Quarter**

    This report has a group that is sorted by quarter (see item 4 above). To fix this, add a field to the source dataset that contains the value of the **ShippedDate** quarter, by changing the RecordSource property as follows:

    ```
    SELECT DISTINCTROW Orders.ShippedDate,
      Orders.OrderID,
      [Order Subtotals].Subtotal,
      DatePart("q",Orders.ShippedDate) As ShippedQuarter
      FROM Orders INNER JOIN [Order Subtotals]
      ON Orders.OrderID = [Order Subtotals].OrderID
      WHERE (((Orders.ShippedDate) Is Not Null));
    ```

    Change the group's GroupBy property to use the new field, **ShippedQuarter**.

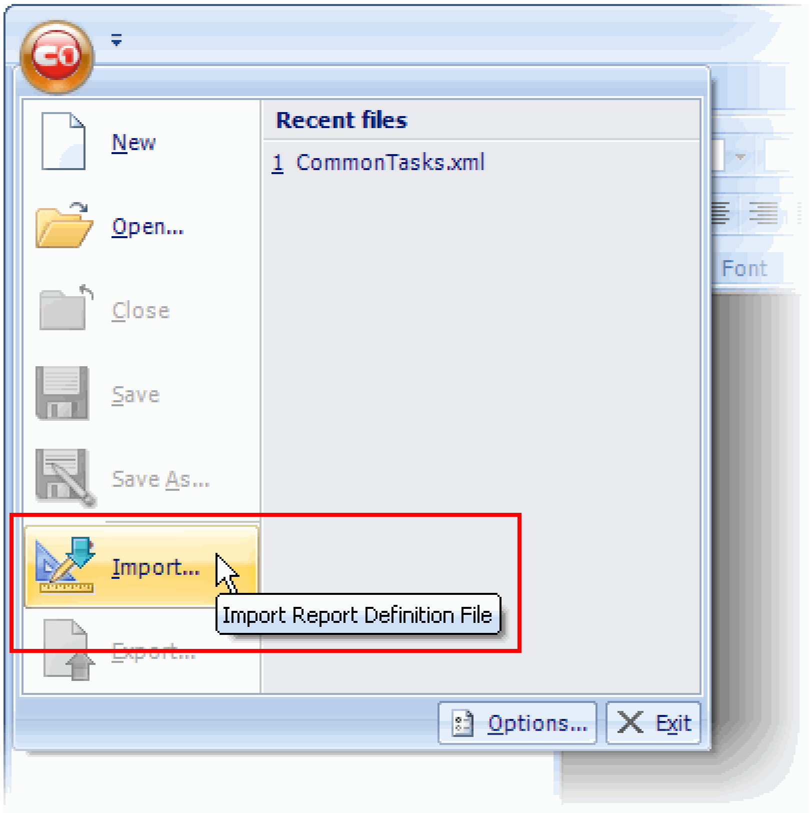
13. **Summary of Sales by Year**

    No action required.

Summing up the information on the table, out of the 13 reports imported from the NorthWind database: eight did not require any editing, three required some code translation, one required changes to the SQL statement, and one had a chart control that was not replaced.

## Importing Crystal Reports

The **C1ReportDesigner** application can also import Crystal report definition files (.rpt files).



To import reports from a Crystal report definition file:

1. Click the **Application** button and select **Import** from the menu.

   A dialog box prompts you for the name of the file you want to import.

2. Select a Crystal report definition file (RPT) and the Designer will convert the report into the C1Report format.

The import process handles most elements of the source reports, with a few exceptions for elements that are not exposed by the Crystal object model or not supported by C1Report. The exceptions include image fields, charts, and cross-tab fields.

## Charting in Reports for WinForms

Aggregate charting is a powerful, yet simple and easy-to-use feature that was added in the 2009 v3 release of **ComponentOne Reports for WinForms**.

**ComponentOne Reports for WinForms** has always supported chart fields using its extensible custom field architecture. The **Chart** field is implemented as a custom field in the C1.Win.C1Report.CustomFields.2.dll assembly, which is installed with the report designer application and is also included as a sample with full source code (**CustomFields**). In the following topics, you'll see how you can customize chart fields in reports using the **C1ReportDesigner** application. The **C1ReportDesigner** application is installed with both **ComponentOne Reports for WinForms** and **ComponentOne Reports for WPF**.

### *Charts in Flat Reports*

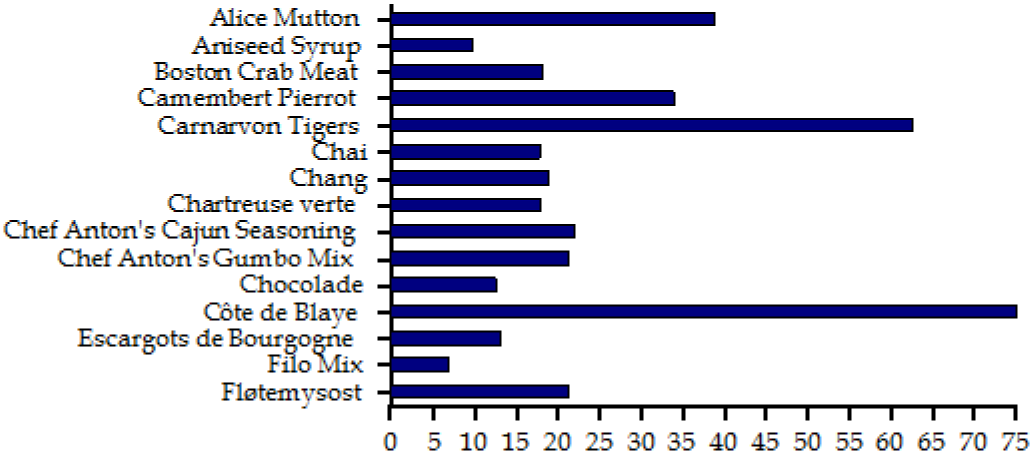
Creating simple charts is very easy. The following steps required to create a simple chart:

1. Open the **C1ReportDesigner** application and create or open a report definition file.

2. Add a **Chart** field to the report, then select it to show its properties in the designer's property window.
3. Set the chart's **DataX** property to the name of the field whose values should be displayed in the X axis (chart categories).
4. Set the chart's **DataY** property to the name of the field whose values should be displayed in the Y axis (chart values).
5. Optionally set additional properties such as **ChartType** and **DataColor**.

For example, the chart below was created based on the NorthWind Products table. In this case, the following properties were set:

**DataX** = "ProductName"
**DataY** = "UnitPrice"



Note that for this chart type (Bar), the value axis (where the **DataY** field is displayed) is the horizontal one, and the category axis is the vertical one.

In this case, a filter was applied to the data in order to limit the number of values shown. Without the filter, the chart would contain too many values and the vertical axis would not be readable.

### *Other Useful Chart Properties*

In addition to the **DataX** and **DataY** properties mentioned above, the Chart object provides a few other properties that are commonly used:

- **ChartType**: This property allows you to select the type of chart to display. There are six options: **Bar** (horizontal bars), **Column** (vertical columns), **Scatter** (X-Y values), **Line**, **Area**, and **Pie**.
- **DataColor**: This property selects the color used to draw the bars, columns, areas, scatter symbols, and pie slices. If the chart contains multiple series, then the **Chart** field automatically generates different shades of the selected color for each series. If you want to select specific colors for each series, use the **Palette** property instead, and set its value to a semi-colon separated list containing the colors to use (for example "Red;Green;Blue").
- **FormatY, FormatX**: These properties determine the format used to display the values along each axis. For example, setting **FormatY** to "c" causes the **Chart** field to format the values along the Y axis as currency values. This is analogous to the **Format** property in regular report fields.
- **XMin, XMax, YMin, YMax**: These properties allow you to specify ranges for each axis. Setting any of them to -1 cause the **Chart** to calculate the range automatically. For example, if you set the **YMax** property to 100, then any values higher than 100 will be truncated and won't appear on the chart.

These properties apply to all chart types. There are a few additional properties that only apply to **Pie** charts:

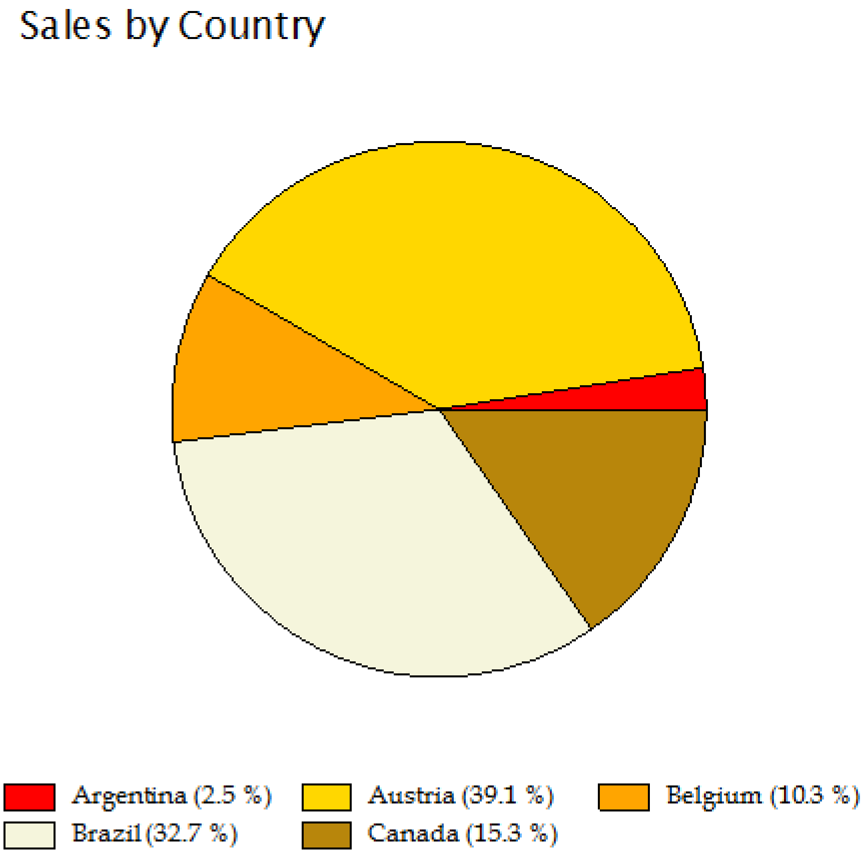
- **ShowPercentages**: Each pie slice has a legend that shows the X value for the slice. If the **ShowPercentages** property is set to true, the legend will also include a percentage value that indicates the size of the slice with respect to the pie. The percentage is formatted using the value specified by the **FormatY** property. For example, if you set **FormatY** to "p2", then the legends will include the X value and the percentage with two decimal points (for example "North Region (15.23%)").
- **RadialLabels**: This property specifies that instead of showing a legend on the right side of the chart, labels with connecting lines should be attached to each slice. This works well for pies with few slices (up to about ten).

The **Chart** field is actually a wrapper for a **C1Chart** control, which provides all the charting services and has an extremely rich object model of its own. If you want to customize the **Chart** field even further, you can use the **ChartControl** property to access the inner **C1Chart** object using scripts.

For example, the **Chart** field does not have a property to control the position of the legend. But the **C1Chart** control does, and you can access this property through the **ChartControl** property. For example, the script below causes the chart legend to be positioned below the chart instead of on the right:

```
' place legend below the chart
chartField.ChartControl.Legend.Compass = "South"
```

If you assign this script to the report's **OnLoad** property, the chart will look like the image below:



The other properties used to create these chart are as follows:

**ChartType** = Pie
**FormatY** = "p1"

**ShowPercentage** = true
**Palette** = "Red;Gold;Orange;Beige;DarkGoldenrod;Goldenrod;"

### Charts with Multiple Series

To create charts with multiple series, simply set the **DataY** property to a string that contains the names of each data field you want to chart, separated by semi-colons.

For example, to create a chart showing product prices and discounts you would set the **DataY** property as shown below:

**DataY** = "UnitPrice;Discount"

If you want to specify the color used to display each series, set the **Palette** property to a list of colors separated by semi-colons. For example, the value displayed below would cause the chart to show the UnitPrice" series in red and the "Discount" series in blue:

**Palette** = "Red;Blue"

### Series with Calculated Values

The **DataY** property is not restricted to field names. The strings that specify the series are actually treated as full expressions, and are calculated like any regular field in the report.

For example, to create a chart showing the actual price of each field you could set the **DataY** property to the value shown below:

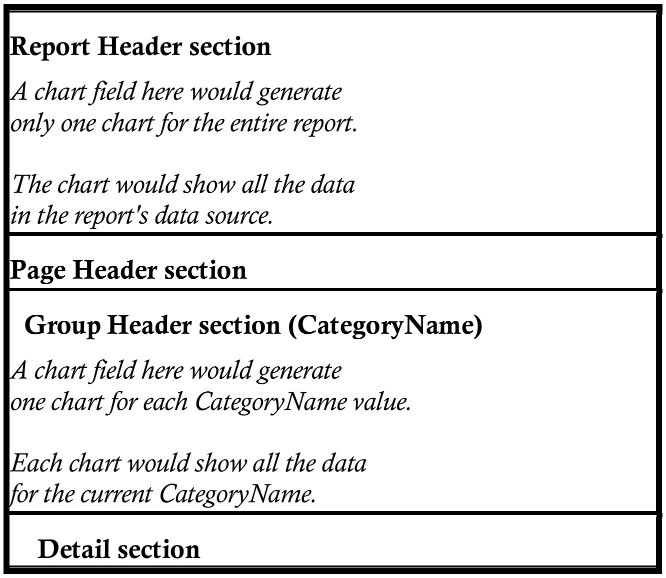
**DataY** = "UnitPrice * (1 - Discount)"

### Charts in Grouped Reports

**Reports for WinForms** allows you to create reports with multiple groups. For example, instead of listing all products in a single flat report, you could group products by category. Each group has a header and a footer section that allow you to display information about the group, including titles and subtotals, for example.

If you add a chart to a group header, the chart will display only the data for the current group. By contrast, adding a chart to the report header or footer would include all the data in the report.

To illustrate this, here is a diagram depicting a report definition as shown in the report designer and showing the effect of adding a **Chart** field to the report header and to a group header:



**Report Header section**

*A chart field here would generate*
*only one chart for the entire report.*

*The chart would show all the data*
*in the report's data source.*

**Page Header section**

**Group Header section (CategoryName)**

*A chart field here would generate*
*one chart for each CategoryName value.*

*Each chart would show all the data*
*for the current CategoryName.*

**Detail section**

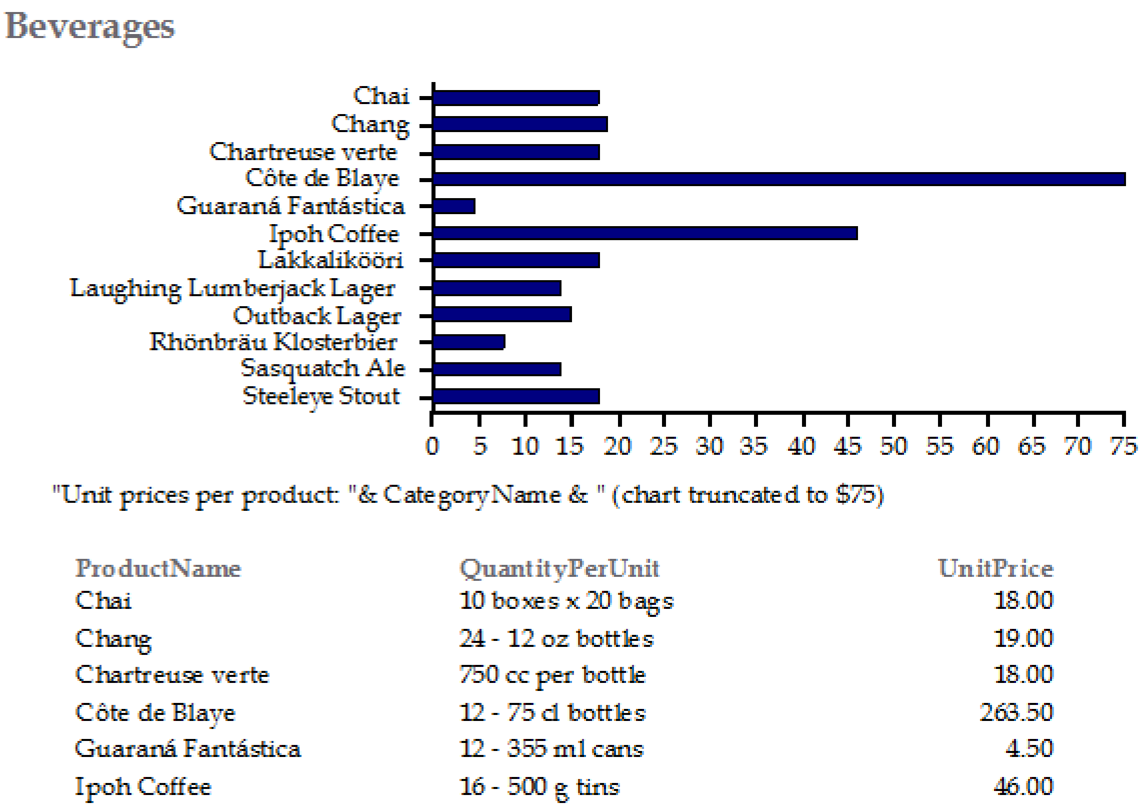
**Group Footer section (CategoryName)**

**Page Footer section**

**Report Footer section**

Continuing with the example mentioned above, if you added a chart to the group header and set the **DataX** property to "ProductName" and the **DataY** property to "UnitPrice", the final report would contain one chart for each category, and each chart would display the unit prices for the products in that category.

The images below show screenshots of the report described above with the group headers, the charts they contain, and a few detail records to illustrate:

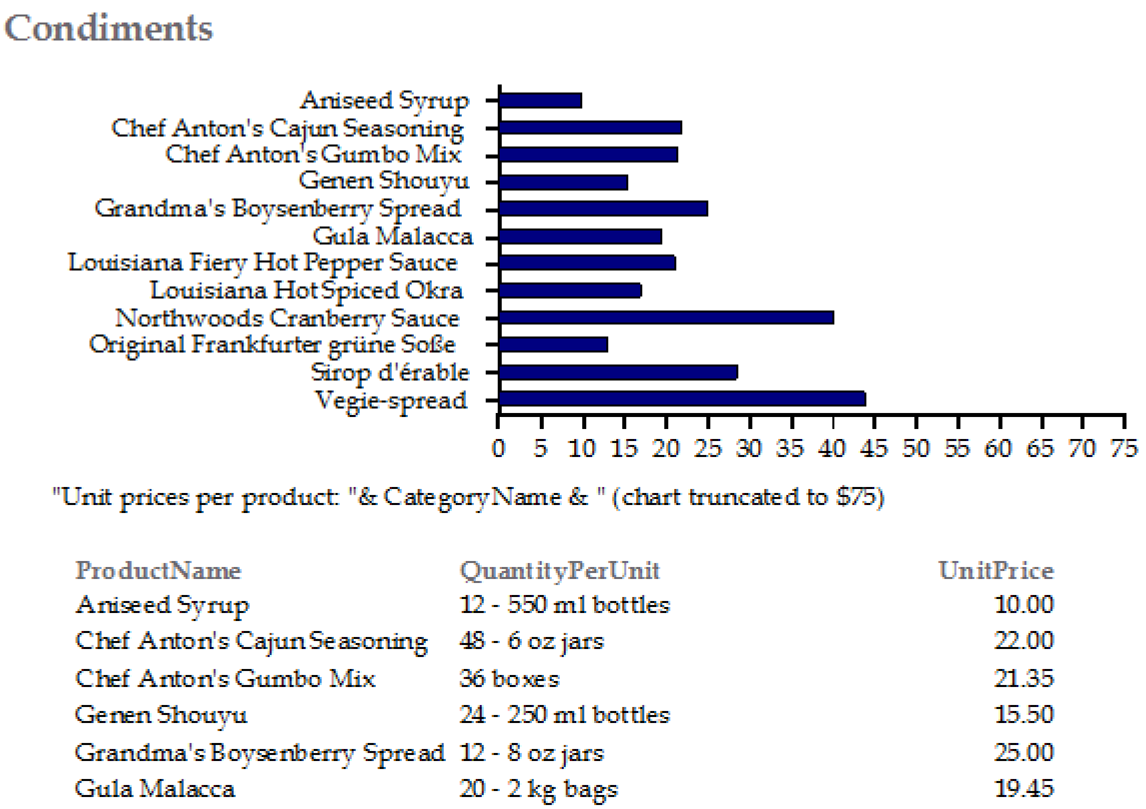


Beverages

"Unit prices per product: "& CategoryName & " (chart truncated to $75)

| ProductName | QuantityPerUnit | UnitPrice |
|---|---|---|
| Chai | 10 boxes x 20 bags | 18.00 |
| Chang | 24 - 12 oz bottles | 19.00 |
| Chartreuse verte | 750 cc per bottle | 18.00 |
| Côte de Blaye | 12 - 75 cl bottles | 263.50 |
| Guaraná Fantástica | 12 - 355 ml cans | 4.50 |
| Ipoh Coffee | 16 - 500 g tins | 46.00 |

The above chart shows unit prices for products in the "Beverages" category. The below chart shows unit prices for products in the "Condiments" category.



Condiments

Aniseed Syrup
Chef Anton's Cajun Seasoning
Chef Anton's Gumbo Mix
Genen Shouyu
Grandma's Boysenberry Spread
Gula Malacca
Louisiana Fiery Hot Pepper Sauce
Louisiana Hot Spiced Okra
Northwoods Cranberry Sauce
Original Frankfurter grüne Soße
Sirop d'érable
Vegie-spread
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75

"Unit prices per product: "& CategoryName & " (chart truncated to $75)

| ProductName | QuantityPerUnit | UnitPrice |
|---|---|---|
| Aniseed Syrup | 12 - 550 ml bottles | 10.00 |
| Chef Anton's Cajun Seasoning | 48 - 6 oz jars | 22.00 |
| Chef Anton's Gumbo Mix | 36 boxes | 21.35 |
| Genen Shouyu | 24 - 250 ml bottles | 15.50 |
| Grandma's Boysenberry Spread | 12 - 8 oz jars | 25.00 |
| Gula Malacca | 20 - 2 kg bags | 19.45 |

**DataX** = "Product Name"
**DataY** = "Unit Price"

Because the chart automatically selects the data based on the scope of the section that contains it, creating charts in grouped reports is very easy.

### *Aggregate Charts*

The **Chart** field included with the 2009 v3 release of **Reports for WinForms** has a powerful new feature called "aggregated charting". This feature allows you to create charts that automatically aggregate data values (**DataY**) that have the same category (**DataX**) using an aggregate function of your choice (sum, average, standard deviation, and so on).

To illustrate this feature, consider an "Invoices" report that groups data by country, customer, and order ID. The general outline for the report is as follows:

| |
|---|
| **Report Header section** |
| **Page Header section** |
| **Group Header section (Country)** |
| **Group Header section (Customer)** |
| **Group Header section (OrderID)** |
| **Detail section** |
| **Group Footer section (OrderID)** |

| |
|---|
| Group Footer section (Customer) |
| Group Footer section (Country) |
| Page Footer section |
| Report Footer section |

Now imagine that you would like to add a chart to each **Country** header displaying the total value of all orders placed by each customer in the current country.

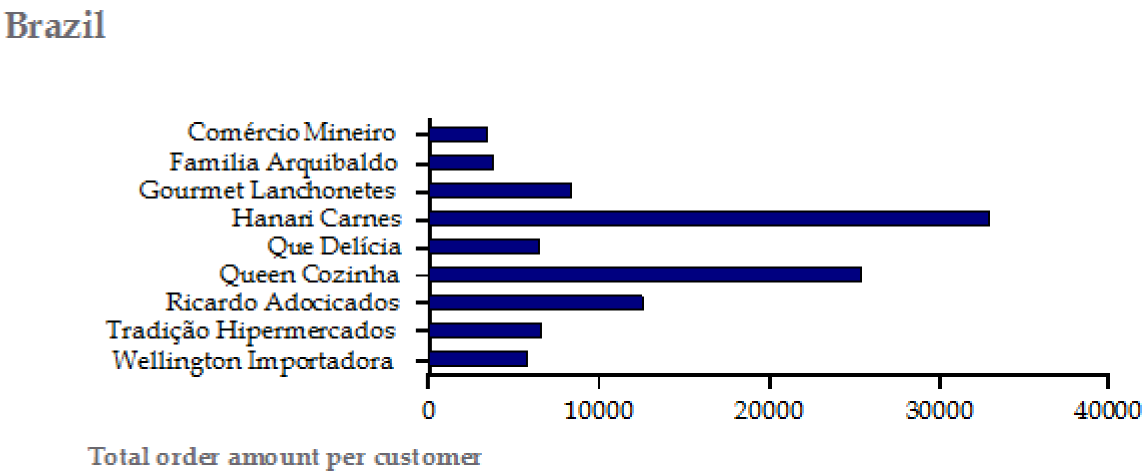
You would start you adding a **Chart** field to the "Country" header section and set the **DataX** and **DataY** properties as follows:

**DataX** = "CustomerName"
**DataY** = "ExtendedPrice"

This would not work. The data for each country usually includes several records for each customer, and the chart would create one data point for each record. The chart would not be able to guess you really want to add the values for each customer into a single data point.

To address this scenario, we added an **Aggregate** property to the **Chart** field. This property tells the chart how to aggregate values that have the same category into a single point in the chart. The **Aggregate** property can be set to perform any of the common aggregation functions on the data: sum, average, count, maximum, minimum, standard deviation, and variance.

Continuing with our example, we can now simply set the chart's **Aggregate** property to "Sum". This will cause the chart to add all "ExtendedPrice" values for records that belong to the same customer into a single data point. The result is shown below:



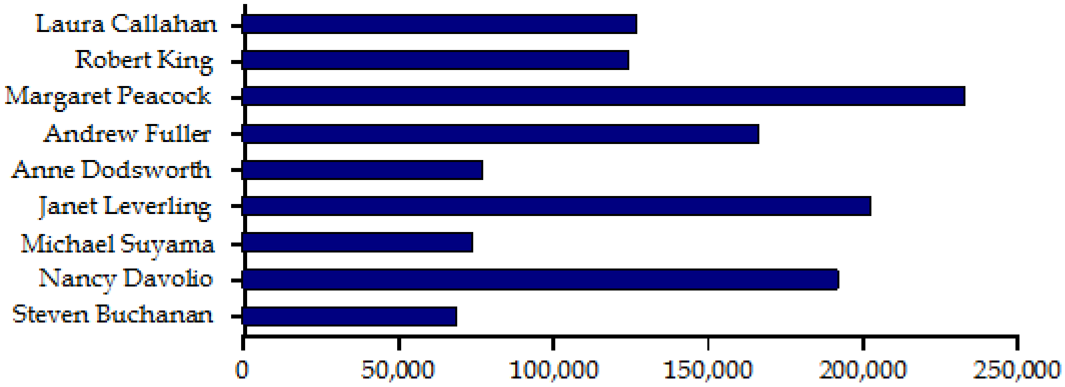
Notice how each customer appears only once. The values shown on the chart correspond to the sum of the "ExtendedPrice" values for all fields with the same "Customer".

Because the chart appears in the "Country" header field, it is repeated for each country, showing all the customers in that country.

If you place the chart in the report header section, it will aggregate data over the entire report. For example, suppose you want to start the "Invoices" report with a chart that shows the total amount ordered by each salesperson. To accomplish this, you would add a **Chart** field to the report header section and would set the following properties:

**DataX** = "Salesperson"
**DataY** = "ExtendedPrice"
**Aggregate** = "Sum"

The image below shows the resulting chart:



Total order amount per Salesperson

Since the chart is placed in the report header section, the values displayed include all countries and all customers. If you moved the chart field from the report header to the "Country" group header, you would obtain a similar chart for each country, showing the total amounts sold by each salesperson in that country.

# Working with C1ReportsScheduler

The following topics contain important information about **C1ReportsScheduler**, a stand-alone application used to schedule report creation to run in the background. Using the **C1ReportsScheduler** application, you can choose what reports to export or print, formats to export the report to, and the schedule and frequency for creating reports.

## About C1ReportsScheduler

The **C1ReportsScheduler** application is designed to run scheduled **ComponentOne Reports for WinForms** tasks in the background. The following report task types are supported:

- An XML report definition loaded into a **C1Report** component.
- An XML report definition imported into a **C1PrintDocument** component.
- A data bound **C1PrintDocument** loaded from a C1D/C1DX file.
- An executable user program generating and exporting or printing a report.

For each task, multiple actions can be specified. For example, a report can be exported to PDF and Excel files and also printed. Each task also has an associated schedule that specifies when the task needs to be executed.

The **C1ReportsScheduler** application consists of two related interacting parts:

- The **C1ReportsScheduler** frontend application (**C1ReportsScheduler.exe**).
- The **C1ReportsScheduler** Windows service (**C1ReportsSchedulerService.exe**).

In the recommended mode of operation, the service (**C1ReportsSchedulerService.exe**) runs in the background, executing specified tasks according to their schedules. The frontend (**C1ReportsScheduler.exe**) can be used to view or edit the task list, start or stop schedules, and control the service. While the frontend is used to install and setup the service, it is not needed for the service to run, so normally it would be started to adjust the list of scheduled tasks as needed, and exited. (If the service is installed on a machine, the frontend connects to it automatically when started.)

While not recommended, standalone mode of operation is also possible. In this mode, the frontend is also used to actually run the scheduled tasks. In that case, of course, terminating the frontend application will also stop all schedules.

Note that all service specific operations (installation, setup, and un-installation) can be performed from the frontend application.

## Installation and Setup

By default, the **C1ReportsScheduler** application and service (**C1ReportsScheduler.exe** and **C1ReportsSchedulerService.exe**) will be installed in the **C:\Program Files\ComponentOne\Studio for WinForms\C1Report\Scheduler** directory. To install and setup the application and service, complete the following steps:

1. Navigate to the installation directory and double-click the **C1ReportsScheduler.exe** application to open it.

   When the **C1ReportsScheduler.exe** application is run for the first time, a dialog box will appear asking whether you would like to install the **C1ReportsScheduler** service.

2. Click **Yes** to install the service (recommended).

   A form should appear which will allow you to set up the service parameters such as the WCF address used by the frontend to communicate with the service, the path of the service configuration file (.c1rsconf), service startup type (manual or automatic), and log options.

3. Adjust the parameters if necessary (defaults should normally work), and click **OK** to install the service.

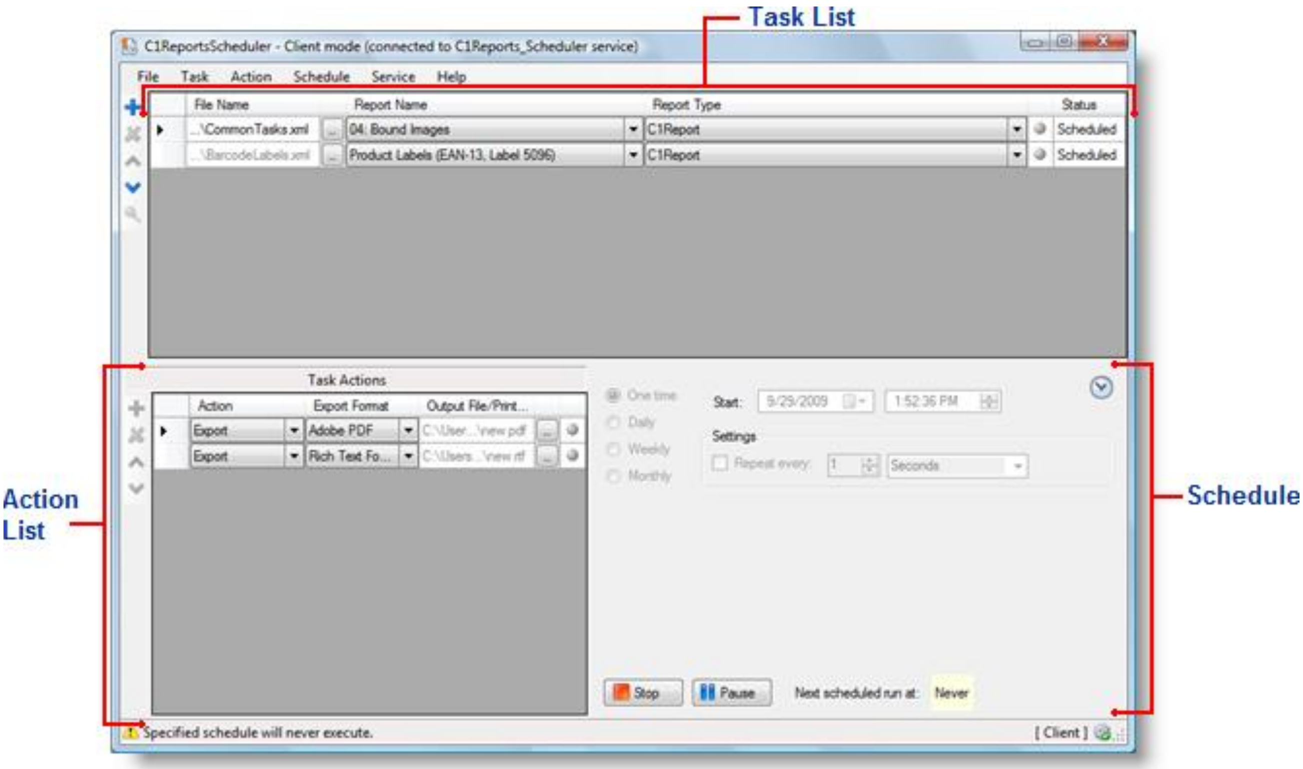
   The progress window should appear, and when it closes the frontend should be running in client mode. This is indicated by the words "client mode" in the form's caption, and "[ Client ]" in the status line (which should also contain an icon showing a gear with a green checkmark).

If you clicked **No** when asked whether to install the service, the frontend will start in standalone mode, as indicated in the form's caption and status line. You can still add tasks, specify their actions and schedules and start them. The only difference is that in standalone mode you will need to keep the frontend running for the tasks to run on schedules.

Note that when in standalone mode, you can install the service at any time and transfer your task list to it. To install the service, open the frontend application, and select **Install Service** from the **Service** menu. To uninstall the service, select **Uninstall Service** from the **Service** menu.

## User Interface

When you open the **C1ReportsScheduler** application, it appears similar to the following image:



C1ReportsScheduler - Client mode (connected to C1Reports_Scheduler service)

| | File Name | Report Name | Report Type | Status |
|---|---|---|---|---|
| ▸ | ...\CommonTasks.xml | 04: Bound Images | C1Report | Scheduled |
| | ...\BarcodeLabels.xml | Product Labels (EAN-13, Label 5096) | C1Report | Scheduled |

Task Actions

| | Action | Export Format | Output File/Print... |
|---|---|---|---|
| ▸ | Export | Adobe PDF | C:\User...\new.pdf |
| | Export | Rich Text Fo... | C:\Users...\new.rtf |

● One time
○ Daily
○ Weekly
○ Monthly

Start: 9/29/2009 1:52:36 PM

Settings

□ Repeat every: 1 Seconds

Stop Pause Next scheduled run at: Never

Specified schedule will never execute. [ Client ]

The main **C1ReportsScheduler** window is divided into three areas:

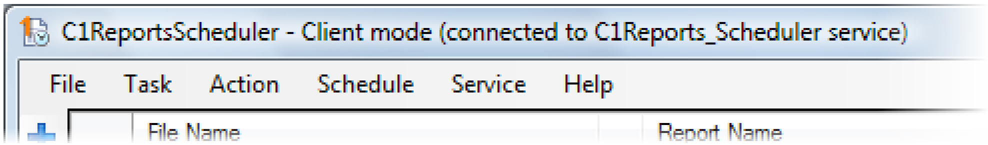
- The upper part of the main window shows the task list. Each task defines a report or document scheduled for generation. For more information, see Task List.
- The lower left part of the main window shows the list of actions defined for the current task. For more information, see Action List.
- The lower right part of the main window shows the schedule for the current task. For more information, see Schedule.

The following topics detail the **C1ReportsScheduler** application's user interface.

## Caption and Status Bar

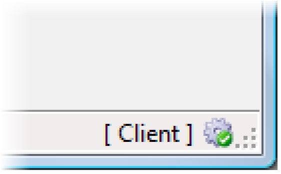
When working with the C1ReportsScheduler application, you may notice that the caption and status bars provide various indicators and information.

The form caption is used to show the current mode (client or standalone). For example, the caption bar in the image below indicates that the application is running in client mode and lists the service it is connected to:



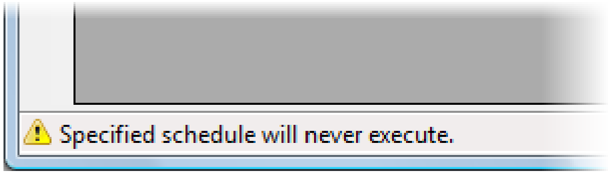
C1ReportsScheduler - Client mode (connected to C1Reports_Scheduler service)

The status bar has two areas. The right side of the status bar displays an icon and a brief text description of the current mode. The two available modes are Client and Standalone mode. For example, in the image below, the status bar indicates it is in Client mode:



[ Client ]

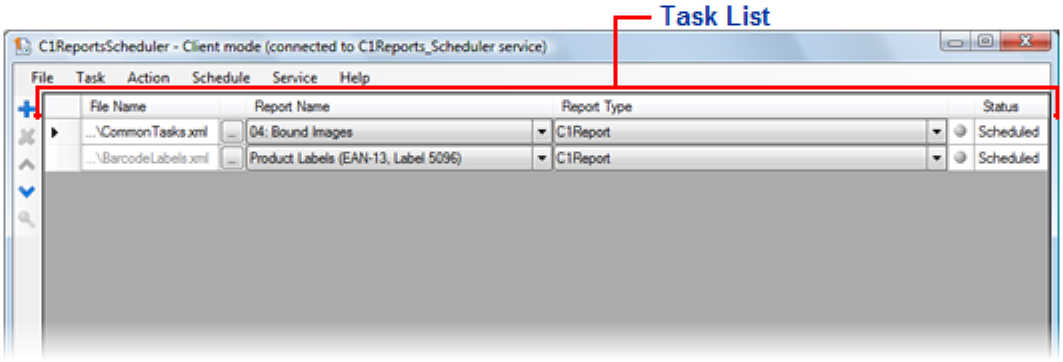
The left side of the status bar displays any warnings or error messages. If the current task or action has errors, the main status area shows a related icon and the error description (for example, the report definition file is not found). For example, in the image below the status bar displays a warning:



Specified schedule will never execute.

## Task List

The task list appears at upper part of the main window and consists of a grid with several columns where you can add various tasks to complete. Each task defines a report or document scheduled for generation. The task list appears similar to the following image:


Task List
C1ReportsScheduler - Client mode (connected to C1Reports_Scheduler service)
File
Task
Action
Schedule
Service
Help

| File Name | Report Name | Report Type | Status |
|---|---|---|---|
| ...\CommonTasks.xml | 04: Bound Images | C1Report | Scheduled |
| ...\BarcodeLabels.xml | Product Labels (EAN-13, Label 5096) | C1Report | Scheduled |

The task list is represented by a grid with the following columns:

- **File Name**

  The **File Name** column lists the name of the **C1Report**, **C1PrintDocument**, or executable file on which to schedule an action. For **C1Report**/Imported **C1Report** type of tasks, this is the name of the report definition file, for **C1PrintDocument** type of tasks, this is the name of C1D/C1DX file containing the document, and for external executable type of tasks, this is the name of the executable file to run. To select a file, click the ellipsis button to the right of the file name text box.

- **Report Name**

  Used only for **C1Report**/Imported **C1Report** tasks, this column specifies the name of the report. It's a combo box: when a report definition file is selected in the first column, the combo drop-down box is automatically filled with available reports' names.

- **Report Type**

  This column specifies the type of the current task. The following report task types are supported:

| States | Description |
|---|---|
| C1Report | For tasks of the **C1Report** type, an instance of **C1Report** component is used to load the report definition and generate the report. |
| Imported C1Report | For tasks of the **Imported C1Report** type, an instance of **C1PrintDocument** component is used to import the report definition and generate the report. |
| C1PrintDocument | For tasks of the **C1PrintDocument** type, an instance of **C1PrintDocument** is used to load and generate the document. |
| External executable | Tasks of the **External executable** type are represented by external programs. The intention is to run applications that rely on code when generating reports. |

- **Task state**

  This column shows a small image representing the current state of the task. Note that this column does not have a caption. Image indicators used in the column include:

| | Description |
|---|---|

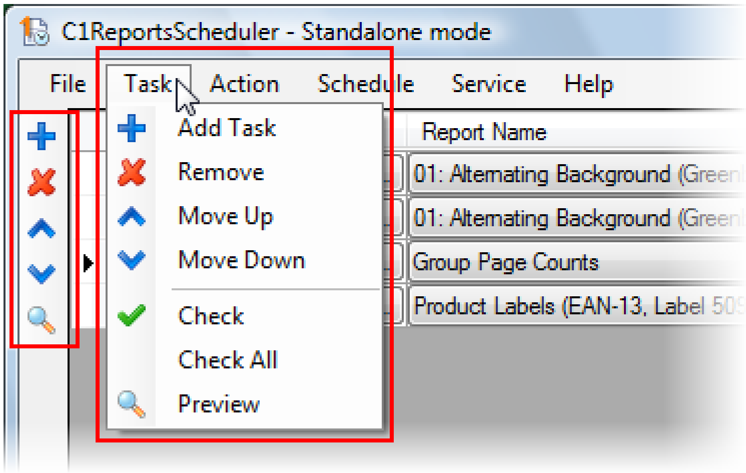
| | |
|---|---|
| ⚪ | A gray ball representing an unchecked task. |
| 🟢 | A green ball representing a successfully checked task that is currently not running. |
| 🟡 | A yellow ball representing a successfully checked task that is currently running. |
| ⚠ | A yellow triangle with an exclamation mark representing a task that has errors. |

- **Status**

  This column shows the current state of the task. States include:

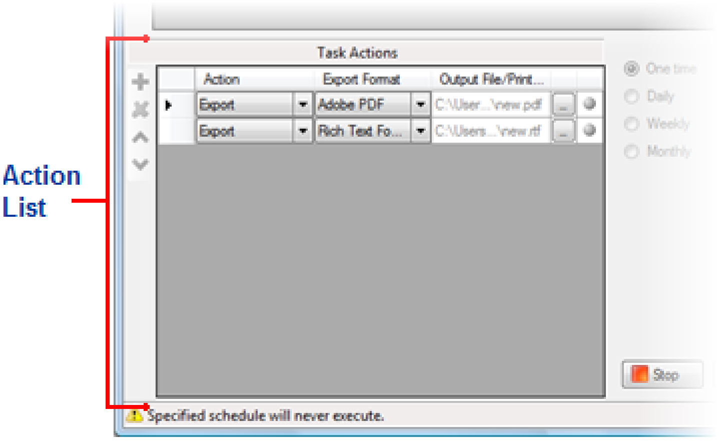
| States | Description |
|---|---|
| Ready | Task is ready but is not scheduled. This is the only status allowing to edit the task. |
| Scheduled | Task is scheduled for execution. Changes are not allowed to a task with this status. |
| Busy | A scheduled task that is currently running. Changes are not allowed to a task with this status. |
| Paused | Task is scheduled for execution but the schedule is paused. Changes are not allowed to a task with this status. |

To manipulate the task list, use the **Task** menu or the toolbar on the left of the task grid:


C1ReportsScheduler - Standalone mode
File
Task
Action
Schedule
Service
Help
Add Task
Remove
Move Up
Move Down
Check
Check All
Preview
Report Name
01: Alternating Background (Green
01: Alternating Background (Green
Group Page Counts
Product Labels (EAN-13, Label 50

## *Action List*

The task action list appears in the bottom-left of the screen and represents the list of actions associated with the current task (the task selected in the Task List). The action list appears like the following image:



| | Action | Export Format | Output File/Print... | | |
|---|---|---|---|---|---|
| ▸ | Export | Adobe PDF | C:\User...\new.pdf | ... | ● |
| | Export | Rich Text Fo... | C:\Users...\new.rtf | ... | ● |

⚠ Specified schedule will never execute.

The action list is represented by a grid with the following columns:

- **Action**

  This is the type of action. The following action types are supported:

| Type | Description |
|---|---|
| Export | Exports the report or document represented by the current task to one of the supported external formats.<br>This action type is not allowed for External executable type of tasks. |
| Print | Prints the report or document represented by the current task.<br>This action type is not allowed for External executable type of tasks. |
| Run | Runs the executable. This action type is only allowed for External executable type of tasks. |

- **Export Format**

  For export actions, this column specifies the export format. Note that different sets of export formats are available for tasks using C1Report and C1PrintDocument components (this is similar to the way export of those components is handled by the preview controls).

- **Output File/Printer Name**

  Specifies either the name of the exported file, or the name of the printer used.
  Click the button to the right of this textbox to select the file or printer name (depending on the action type).

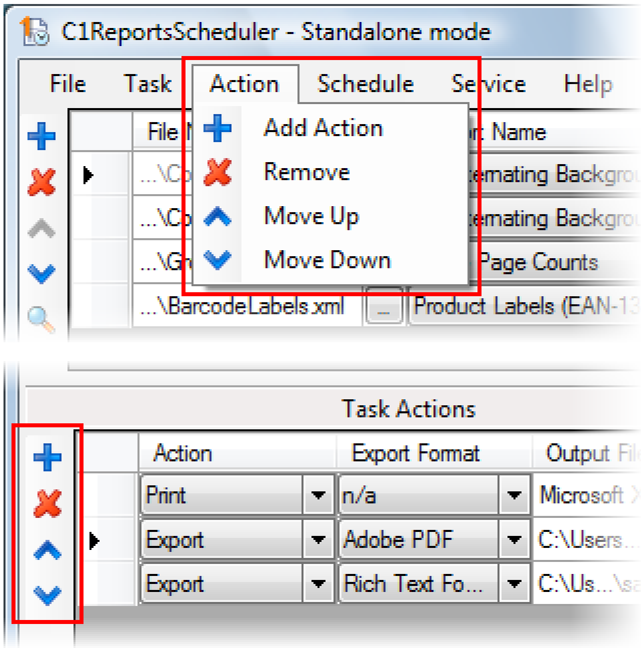
- **Current Status**

  The last column (without a title) is used to show a small image representing the current status of the action:

| | Description |
|---|---|
| ● | A gray ball representing an unchecked task. |
| 🟢 | A green ball representing a successfully checked task that is |

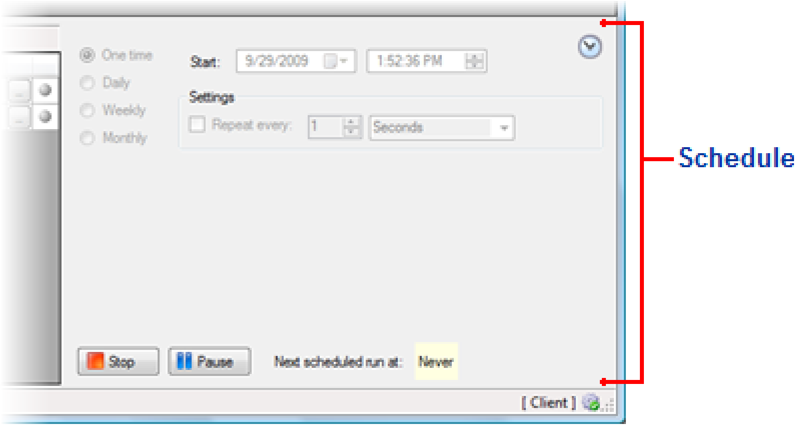
| | |
|---|---|
| | currently not running. |
| ● | A yellow ball representing a successfully checked task that is currently running. |
| ⚠ | A yellow triangle with an exclamation mark representing a task that has errors. |

To manipulate the action list, use the **Action** menu or the toolbar on the left of the action grid:



## *Schedule*

The scheduling panel displays the schedule associated with the current task and allows you to schedule and run the task. The scheduling area appears similar to the following image:



The scheduling section of the application includes the following options:

- **Frequency**

  In the upper left part of the panel there are four radio buttons specifying how often the task will run: **One time**, **Daily**, **Weekly**, or **Monthly**. The first option, **One time**, allows the task to repeat every specified number of seconds, minutes or hours. The other options allow recurrence on a daily, weekly, or monthly basis. As each radio button is selected, the schedule panel shows different scheduling options.

- **Start Date and Time**

  The start date and time do what you'd expect – set the date and time that the scheduling action should begin.

- **Start, Stop, and Pause**

  In the bottom part of the panel, there are two buttons allowing you to start, stop, pause or resume the current task. The buttons change and become available depending on the status of the current task.

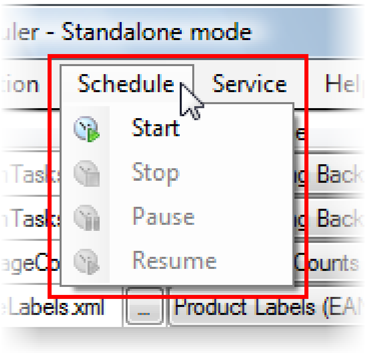
- **Next scheduled run**

  To the right of the buttons, the time of the next scheduled run of the task is shown.

- **Recurrence**

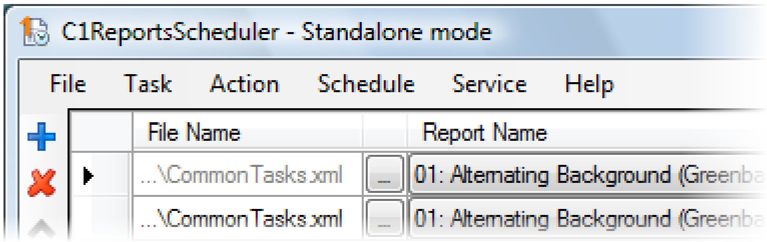
  This section changes depending on the frequency radio button selected. For example, the **Daily** option includes a numeric box allowing you to choose the number of days between the schedule tasks, the **Monthly** option allows you to select the months to run the task and the date each month, and so on.

To start, stop, pause, or resume a scheduled task, you can also use the **Schedule** menu:



### *Menu System*

The **C1ReportsScheduler** application includes several menu options, and the application menu includes **File**, **Task**, **Action**, **Schedule**, **Service**, and **Help** options which appear similar to the following:



| File Name | | Report Name |
|---|---|---|
| ...\CommonTasks.xml | ... | 01: Alternating Background (Greenba |
| ...\CommonTasks.xml | ... | 01: Alternating Background (Greenba |

This topic describes each of the available menu options.

**File**

The **File** menu includes the following options:

- **New**

  Clears the current task list.

- **Open**

  Opens an existing C1Reports Scheduler configuration file (.c1rsconf).

- **Save**

  Saves the current task list.

- **Save As**

  Saves the current task list in a C1Reports Scheduler configuration file (.c1rsconf).

- **Exit**

  Closes the program.

### Task

The **Task** menu includes the following options:

- **Add Task**

  Adds a new task to the task list. The task is added to the end of the list, but can be moved up and down in the list using appropriate commands.

- **Remove**

  Removes the current task from the list.

- **Move Up**

  Moves the current task up in the list.

- **Move Down**

  Moves the current task down in the list.

- **Check**

  Checks the validity of the current task and all its actions' specification. Items that are checked include the existence and validity of report definition, correctness of output file names, and so on. A task is checked automatically when it is scheduled. A successfully checked task has a green ball icon in the "State" column. If a check failed, the task cannot be scheduled, and an exclamation mark icon is displayed. Hover the mouse over that icon to see the error message (it is duplicated in the status line when the task is current).

- **Check All**

  Checks the validity of all tasks in the list.

- **Preview**

  Generates the current task's report or document, and shows it in a print preview dialog. Note that this command is disabled when the task is scheduled.

### Action

The **Action** menu includes the following options:

- **Add Action**

  Adds a new action to the action list of the current task. The action is added to the end of the list, but can be moved up and down in the list using appropriate commands.

- **Remove**

  Removes the current action from the list.

- **Move Up**

  Moves the current action up in the list.

- **Move Down**

  Moves the current action down in the list.

### Schedule

The **Schedule** menu includes the following options:

- **Start**

  Starts the current task's schedule. When the schedule is started, the task or its actions cannot be edited.

- **Stop**

  Stops the current task's schedule.

- **Pause**

  Pauses the current task's schedule.

- **Resume**

  Resumes the current task's schedule if it has been paused.

### Service

The **Service** menu includes the following options:

- **Connect**

  Connects to the C1Reports Scheduler service. This command is only available when the service is running.

- **Disconnect**

  Disconnects from the C1Reports Scheduler service.

- **Transfer Tasks**

  Transfers the current task list to the C1Reports Scheduler service. This command is available when the service is running but the frontend application is disconnected from the service and contains its own task list.

- **Start**

  Starts the C1Reports Scheduler service. This command is available when the service is installed on the machine but is not running.

- **Stop**

  Stops the C1Reports Scheduler service. This command is available when the service is installed on the machine and is running.

- **Pause**

  Pauses the C1Reports Scheduler service. This command is available when the service is installed on the machine and is running.

- **Resume**

  Resumes the C1Reports Scheduler service. This command is available when the service is installed on the machine and is paused.

- **Service Setup**

  Launches the C1Reports Scheduler service setup dialog. That dialog allows to adjust the service parameters, and will restart the service when OK is pressed in the dialog. This command is available if the service is installed on the machine.

- **Install Service**

  Installs the C1Reports Scheduler service on the machine. This command is available when the service is not installed on the machine.

- **Uninstall Service**

  Uninstalls the C1Reports Scheduler service. This command is available when the service is installed on the machine.

- **Service Log**

  Shows or shows the window with the C1Reports Scheduler service log.

### Help

The **Help** menu includes the following options:

- **Contents**

  Shows the help file.

- **About**

  Shows the **About** box which includes information about the application, as well as links to online resources.

# Working with C1PrintDocument

The **C1PrintDocument** component allows you to create complex documents that can be printed, previewed, persisted in a disc file, or exported to a number of external formats including PDF (Portable Document Format) and RTF (Rich Text File).

**C1PrintDocument** provides a number of unique features, including:

- Consistent transparent hierarchical document structure.
- Easy to use and efficient styles.
- Documents that can be changed and re-rendered to accommodate the changes and/or different page settings.
- Documents' preview, printing, persisting, and export to external formats.
- Documents that are input forms (supported by the preview components).
- Complete tables support, including nested tables.
- Support for multi-style text, including inline images.
- True type fonts embedding.
- Hyperlinks.
- Auto-generated TOC.
- And more!

The default namespace used by **C1PrintDocument** is C1.C1Preview (the Windows Forms controls for previewing documents, uses the default namespace C1.Win.C1Preview).

The whole document is represented by the C1PrintDocument class, which inherits from Component.

The main parts of a C1PrintDocument are:

- **Body**

  The actual content of the document – text, images and so on. The body represents the logical structure of the document (see also the page collection below).

- **Pages**

  The collection of pages which were generated based on the content (body) and a particular page setup. Normally the page collection can be regenerated without loss of information (for example, for a different paper size).

- **Style**

  The root style of the document. Styles control most of the visual properties of the document elements (such as fonts, colors, line styles and so on).

- **Dictionary**

  Images used in multiple places in a document can be put in the dictionary and reused to improve performance and reduce memory footprint.

- **EmbeddedFonts**

  The collection of embedded true type fonts used by the document.

- **Tags**

The collection of user defined tags that can be inserted in the document to be replaced by their values when the document is generated.

- **DataSchema**

  Contains the data schema built-in document.

# Render Objects

The following sections discuss the hierarchy of render objects, as well as containment, positioning, and stacking rules.

## *Render Objects Hierarchy*

All content of a C1PrintDocument is represented by **render objects**. A rich hierarchy of render objects (based on the RenderObject class) is provided to represent different types of content. Below is the hierarchy of render object types, with a brief description for each class (note that italics indicate abstract classes):

| **Render Object Type** | | | | **Description** |
|---|---|---|---|---|
| *RenderObject* | | | | The base class for all render objects. |
| | RenderArea | | | Represents a general-purpose container for render objects. |
| | | RenderToc | | Represents a table of contents. |
| | | RenderReport | | Represents a sub-report (a C1Report contained within a RenderField and specified by its SubReport property). |
| | | RenderSection | | Represents a section of an imported C1Report. |
| | | RenderC1Printable | | Represents an external object that can be seamlessly rendered in a C1PrintDocument. (The object must support the IC1Printable interface.) |
| | RenderEmpty | | | Represents an empty object. Provides a convenient placeholder for things like page breaks and so on where no content needs to be rendered. |
| | RenderGraphics | | | Represents a drawing on the .NET Graphics object. |
| | RenderImage | | | Represents an image. |
| | *RenderInputBase* | | | The abstract base class for all Preview Forms' input controls. Derived types represent active UI elements embedded in the document when the document is shown by the preview. |
| | | *RenderInputButtonBase* | | The abstract base class for button-like input controls (button, check box, radio button). |
| | | | RenderInputButton | Represents a push button. |
| | | | RenderInputCheckBox | Represents a checkbox. |
| | | | RenderInputRadioButton | Represents a radio button. |
| | | RenderInputComboBox | | Represents a combo box (text input control with a dropdown list). |
| | | RenderInputText | | Represents a textbox control. |
| | RenderRichText | | | Represents RTF text. |
| | *RenderShapeBase* | | | The abstract base class for classes representing shapes (lines, polygons and so on). |

| | | | | |
|---|---|---|---|---|
| | | *RenderLineBase* | | The abstract base class for lines and polygons. |
| | | | RenderLine | Represents a line. |
| | | | RenderPolygon | Represents a closed or open polygon. |
| | | RenderRectangle | | Represents a rectangle. |
| | | | RenderEllipse | Represents an ellipse. |
| | | | RenderRoundRectangle | Represents a rectangle with rounded corners. |
| | RenderTable | | | Represents a table. |
| | *RenderTextBase* | | | The abstract base class for classes representing text and paragraph objects. |
| | | RenderParagraph | | Represents a paragraph (a run of text fragments in different styles, and inline images). |
| | | | RenderTocItem | Represents an entry in the table of contents (RenderToc). |
| | | RenderText | | Represents a piece of text rendered using a single style. |
| | RenderField | | | Represents a field of a C1Report. Objects of this type are created when a C1Report is imported into a C1PrintDocument. |
| | RenderBarCode | | | Represents a barcode. |

## Render Objects Containment, Positioning, and Stacking Rules

All visible content of a C1PrintDocument is represented by a tree of render objects (instances of types derived from RenderObject, as described above), with the root of the tree being the **Body** of the document. So in order to add a render object to the document, it must be either added to the **Children** collection of the document's **Body**, or to the **Children** collection of another object already in the hierarchy. For example, the render text in the following code is added to the document's **Body.Children** collection:

- Visual Basic

```
Dim doc As New C1PrintDocument()
Dim rt As New RenderText()
rt.Text = "This is a text."
doc.Body.Children.Add(rt)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText();
rt.Text = "This is a text.";
doc.Body.Children.Add(rt);
```

A document's **Body.Children** collection contains all top-level render objects of the document. Each render object in its turn also has a **Children** collection, which contains render objects within it, and so on. (This is very similar to the way Windows Forms' controls can be nested inside each other – any control has the collection of contained controls, and so on.)

In addition to the document's body, there are two other places for render objects in a document – the page header and footer, accessible via the PageHeader and PageFooter properties.

### Render Areas

Although any render object can contain other render objects as its children, there is one render object designed specifically as a container for other objects – RenderArea. The primary difference between a render area and other render objects (such a render text) is that for a render area, specifying either of its width or height as **Auto** means that the corresponding dimension is determined by the size of the children, while for other types of objects, auto size is determined by the object's own content (text size for a RenderText, image size for a RenderImage, and so on).

By default, when a new render area is created, its **Width** is equal to the width of its parent (so a top-level render area stretches across the whole page – or across the current column for multi-column layouts). The **Height** of a render area, on the other hand, is by default set to Auto (**Unit.Auto**), and is determined by the combined height of the render area's children. So the default behavior of a top-level render area is to take up the whole page width, and stretch down as needed (possibly spanning multiple pages) to accommodate all its content. You can set the **Width** of a render area to auto (**Unit.Auto**), in which case it will adjust to accommodate the combined widths of the area's children. In this case, if the combined width of the area's children exceeds the width of the page, **horizontal** page breaks will occur, adding extension pages to the right of the current page. To prevent horizontal page breaks (clipping the area on the right if necessary), set the area's CanSplitHorz property to **False** (it is **True** by default).

### *Stacking*

Within their container (parent object or document body), render objects by default are placed according to the stacking rules, determined by the value of the **Stacking** property of the container (document for top-level objects). This value can be one of the following  StackingRulesEnum enumeration members:

- **BlockTopToBottom**

  Objects are placed one beneath the other within the container. When the bottom edge of the current page is reached, a new page is added. This is the default.

- **BlockLeftToRight**

  Objects are placed one next to another, from left to right. When the right edge of the current page is reached, a new "horizontal" page is added (a horizontal page logically extends the preceding page to the right; **C1PreviewPane** respects this location by default, showing such pages arranged in a row).

- **InlineLeftToRight**

  Objects are placed inline, one next to another, from left to right. When the right edge of the current page is reached, the sequence wraps to the next line. A new page is added when the bottom of the current page is reached.

Stacking rules do not propagate down into the contained objects (children). In other words, if you define a render area and set its stacking to the (non-default) value **BlockLeftToRight**, and then add another render area inside the first one – its stacking will be the default (**BlockTopToBottom**) unless you explicitly change it.

You may also use the X and Y properties of a render object to set its position explicitly (see the next section for details). In this case that render object does not participate in the stacking order at all – that is, its position neither affects the positioning of its siblings nor is affected by their positions.

### *Specifying Render Objects' Size and Location*

The four properties controlling the size and location of a render object are:

- X  – specifies the X coordinate of the object.
- Y – specifies the Y coordinate of the object.
- Width – specifies the width of the object.
- Height – specifies the height of the object.

All those properties have the value type **C1.C1Preview.**Unit. The default value for **X** and **Y** is **Auto** (represented by the static field **Unit.Auto**), which means that the object is positioned according to the stacking rules provided by its parent (see Stacking above for more information). Width and height have different defaults depending on the type of the render object.

The following table lists the default sizes (width and height) for all render objects, as well as the rules used to calculate auto sizes:

| | Width | Height | Auto Size |
|---|---|---|---|
| **RenderArea**<br>**RenderToc**<br>**RenderReport**<br>**RenderSection**<br>**RenderC1Printable** | Parent width | Auto | Determined by the combined size of the children |
| **RenderEmpty** | Auto | Auto | 0 |
| **RenderGraphics** | Auto | Auto | Determined by the size of the content |
| **RenderImage** | Auto | Auto | Determined by the size of the image |
| **RenderInputButton**<br>**RenderInputCheckBox**<br>**RenderInputRadioButton**<br>**RenderInputComboBox**<br>**RenderInputText** | Auto | Auto | Determined by the size of the content |
| **RenderRichText** | Parent width (auto width is not supported). | Auto (determined by the text size). | -- |
| **RenderLine**<br>**RenderPolygon**<br>**RenderEllipse**<br>**RenderArc**<br>**RenderPie**<br>**RenderRectangle**<br>**RenderRoundRectangle** | Auto | Auto | Determined by the size of the shape |
| **RenderTable** | Parent width (auto width is calculated as the sum of columns' widths). | Auto | Determined by the total width of all columns for width, and by the total height of all rows for height |
| **RenderParagraph**<br>**RenderText**<br>**RenderTocItem** | Parent width | Auto | Determined by the size of the text |
| **RenderField** | Parent width | Auto | Determined by the size of the content |
| **RenderBarCode** | Auto | Auto | Determined by the size of the content |

You can override the default values for any of those properties with custom values (specifying anything but **Auto** as the value for X or Y coordinates excludes the object from the stacking flow; see Stacking for more information). The size and location properties can be set in any of the following ways (note that **ro** indicates a render object in the samples below):

- As auto (semantics depend on the render object type):

  ```
  ro.Width = Unit.Auto;
  ro.Height = "auto";
  ```

- As absolute value:

  ```
  ro.X = new Unit(8, UnitTypeEnum.Mm);
  ```

**ro.Y = 8;** (**C1PrintDocument.DefaultUnit** is used for the units);

**ro.Width = "28mm";**

- As percentage of a parent's size (using this for coordinates is not meaningful and will yield 0):

  **ro.Height = new Unit(50, DimensionEnum.Width);**

  **ro.Width = "100%";**

As a reference to a size or position of another object. The object can be identified by any of the following key words:

**self** – the current object (the default value, may be omitted);

**parent** – the object's parent;

**prev** – the previous sibling;

**next** – the next sibling;

**page** – the current page;

**column** – the current column;

**page<N>** – page with the specified number (note: the page must already exist, that is forward references to future pages are not supported);

**column<M>** – a column (specified by number) on the current page;

**page<N>.column<N>** – a column (specified by number) on the specified page;

**<object name>** – the object with the specified name (the value of the **Name** property; the object is searched first among the siblings of the current object, then among its children).

Sizes and locations of the referenced objects are identified by the following key words: **left**, **top**, **right**, **bottom**, **width**, **height** (coordinates are relative to the object's parent).

Some examples:

**ro.Height = "next.height";** – sets the object's height to the height of its next sibling;

**ro.Width = "page1.width";** – sets the object's width to the width of the first page;

**ro.Height = "width";** – sets the object's height to its own width;

**ro.Y = "prev.bottom";** – sets the object's Y coordinate to the bottom of its previous sibling;

**ro.Width = "prev.width";** – sets the object's width to the width of its previous sibling.

- Using functions "Max" and "Min". For example:

  **ro.Width = "Max(prev.width,6cm)";** – sets the object's width to the maximum of 6 cm and the object's previous sibling's width;

- As an expression. Expressions can use any of the ways described above to reference size and position of another object, combined using the operators **+**, **-**, *****, **/**, **%**, functions **Max** and **Min**, and parentheses **(** and **)**. For example:

  **ro.Width = "prev.width + 50%prev.width";** – sets the object's width to the width and a half of its previous sibling;

  **ro.Width = "150%prev";** – same as above (when referencing the same dimension as the one being set, the dimension – "width" in this case – can be omitted).

  **ro.Width = "prev*1.5";** – again, same as above but using multiplication rather than percentage.

In all of the above samples where a size or location is set to a string, the syntax using the **Unit(string)** constructor can also be used, for example:

- Visual Basic

```
ro.Width = New C1.C1Preview.Unit("150%prev")
```

- C#

```
ro.Width = new Unit("150%prev");
```

Case is not important in strings, so "prev.width", "PrEv.WidTh", and "PREV.WIDTH" are equivalent.

### *Examples of relative positioning of render objects*

Below are some examples showing the use of relative positioning of objects to arrange an image and a text ("`myImage`" in those samples is an object of type System.Drawing.Image declared elsewhere in code):

This code places the text below the image simply adding one object after another into the regular block flow:

- Visual Basic

```
Dim doc as New C1PrintDocument
Dim rt as New RenderText("test")
Dim ri as New RenderImage(myImage)
Dim ra As New RenderArea()
ra.Children.Add(ri)
ra.Children.Add(rt)
doc.Body.Children.Add(ra)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("test");
RenderImage ri = new RenderImage(myImage);
RenderArea ra = new RenderArea();
ra.Children.Add(ri);
ra.Children.Add(rt);
doc.Body.Children.Add(ra);
```

This code produces equivalent result (text below image) while the children are added to the area in inverse order (because both objects have non-auto coordinates specified explicitly, neither is inserted into the block flow):

- Visual Basic

```
Dim doc as New C1PrintDocument
Dim rt as New RenderText("test")
Dim ri as New RenderImage(myImage)
Dim ra As New RenderArea()
' place image at the top of the parent:
ri.Y = 0
' place text below next sibling:
rt.Y = "next.bottom"
' auto-size text width:
rt.Width = Unit.Auto
ra.Children.Add(ri)
ra.Children.Add(rt)
doc.Body.Children.Add(ra)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("test");
RenderImage ri = new RenderImage(myImage);
RenderArea ra = new RenderArea();
// place image at the top of the parent:
ri.Y = 0;
// place text below next sibling:
rt.Y = "next.bottom";
```

```
// auto-size text width:
rt.Width = Unit.Auto;
ra.Children.Add(rt);
ra.Children.Add(ri);
doc.Body.Children.Add(ra);
```

The following code inserts the image into the regular block flow, while putting the text to the right of the image, centering it vertically relative to the image:

- Visual Basic

```
Dim doc as New C1PrintDocument
Dim rt as New RenderText("test")
Dim ri as New RenderImage(myImage)
Dim ra As New RenderArea()
ra.Children.Add(ri)
rt.Width = Unit.Auto
' add text after the image:
ra.Children.Add(rt)
rt.X = "prev.right"
rt.Y = "prev.height/2-self.height/2"
doc.Body.Children.Add(ra)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("test");
RenderImage ri = new RenderImage(myImage);
RenderArea ra = new RenderArea();
ra.Children.Add(ri);
rt.Width = Unit.Auto;
// add text after the image:
ra.Children.Add(rt);
rt.X = "prev.right";
rt.Y = "prev.height/2-self.height/2";
doc.Body.Children.Add(ra);
```

This code also places the text to the right of the image, centered vertically – but uses the **RenderObject.Name** in the positioning expressions rather than relative id "prev", Also, the text is shifted 2mm to the right, demonstrating the use of absolute lengths in expressions:

- Visual Basic

```
Dim doc as New C1PrintDocument
Dim rt as New RenderText("test")
Dim ri as New RenderImage(myImage)
Dim ra As New RenderArea()
ri.Name = "myImage"
rt.Width = "auto"
rt.X = "myImage.right+2mm"
rt.Y = "myImage.height/2-self.height/2"
ra.Children.Add(ri)
ra.Children.Add(rt)
doc.Body.Children.Add(ra)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("test");
RenderImage ri = new RenderImage(myImage);
RenderArea ra = new RenderArea();
ri.Name = "myImage";
rt.Width = "auto";
rt.X = "myImage.right+2mm";
rt.Y = "myImage.height/2-self.height/2";
ra.Children.Add(ri);
ra.Children.Add(rt);
doc.Body.Children.Add(ra);
```

The code below modifies the same example so that the text is shifted to the right at least 6cm, using the built-in Max functions:

- Visual Basic

```
Dim doc as New C1PrintDocument
Dim rt as New RenderText("test")
Dim ri as New RenderImage(myImage)
Dim ra As New RenderArea()
ri.Name = "myImage"
rt.Width = "auto"
rt.X = "Max(myImage.right+2mm,6cm)"
rt.Y = "myImage.height/2-self.height/2"
ra.Children.Add(ri)
ra.Children.Add(rt)
doc.Body.Children.Add(ra)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("test");
RenderImage ri = new RenderImage(myImage);
RenderArea ra = new RenderArea();
ri.Name = "myImage";
rt.Width = "auto";
rt.X = "Max(myImage.right+2mm,6cm)";
rt.Y = "myImage.height/2-self.height/2";
ra.Children.Add(ri);
ra.Children.Add(rt);
doc.Body.Children.Add(ra);
```

The following code snippet aligns image to the right side of the page (utilizing the default value for the width of a render area – parent width), while the text is left-aligned, and centered vertically relative to the image:

- Visual Basic

```
Dim doc as New C1PrintDocument
Dim rt as New RenderText("test")
Dim ri as New RenderImage(myImage)
Dim ra As New RenderArea()
ri.Name = "myImage"
' right-align image:
ri.X = "parent.right-width"
' left-align text:
rt.X = "0"
rt.Y = "myImage.height/2-height/2"
ra.Children.Add(ri)
ra.Children.Add(rt)
doc.Body.Children.Add(ra)
```

- C#

  C1PrintDocument doc = new C1PrintDocument();

```
RenderText rt = new RenderText("test");
RenderImage ri = new RenderImage(myImage);
RenderArea ra = new RenderArea();
ri.Name = "myImage";
```

```
// right-align image:
ri.X = "parent.right-width";
// left-align text:
rt.X = "0";
rt.Y = "myImage.height/2-height/2";
ra.Children.Add(ri);
ra.Children.Add(rt);
doc.Body.Children.Add(ra);
```

## Render Object Shadows

In the 2009 v3 release of **Reports for WinForms**, support was added for shadows cast by render objects. The new public interface IShadow, is implemented by a public structure Shadow, and exposed by a non-ambient public property Shadow.

It includes the following sub-properties:

| **Property** | **Description** |
|---|---|
| Transparency | Gets or sets the transparency of the shadow, in percent. A value of 0 defines a solid (non-transparent) shadow, a value of 100 (which is the default) defines a fully transparent (invisible) shadow. |
| Size | Gets or sets the size of the shadow relative to the size of the object, in percent. A value of 100 (which is the default) indicates that the shadow has the same size as the object. |
| Distance | Gets or sets the distance that the shadow's center is offset from the the object's center. Note that only absolute Unit values (such as "0.5in" or "4mm") can be assigned to this property. The default is 2mm. |
| Angle | Gets or sets the angle, in degrees, of the shadow. The angle is measured relative to the three o'clock position clockwise. The default is 45. |
| Color | Gets or sets the color of the shadow. The default is Black. |

The following sample code defines a shadow on a render object:

- Visual Basic

```
Dim doc As C1PrintDocument = C1PrintDocument1
Dim rt As New RenderText("Sample Shadow")
rt.Width = Unit.Auto
rt.Style.Shadow.Transparency = 20
rt.Style.Shadow.Color = Color.BurlyWood
doc.Body.Children.Add(rt)
```

- C#

```
C1PrintDocument doc = c1PrintDocument1;
RenderText rt = new RenderText("Sample Shadow");
rt.Width = Unit.Auto;
rt.Style.Shadow.Transparency = 20;
rt.Style.Shadow.Color = Color.BurlyWood;
doc.Body.Children.Add(rt);
```

Note that while you do not need to create a Shadow object when setting shadow properties, you may choose to do so, for example, like this:

- Visual Basic

```
Dim doc As C1PrintDocument = C1PrintDocument1
Dim rt As New RenderText("Sample Shadow")
rt.Width = Unit.Auto
rt.Style.Shadow = New Shadow(20, 100, "1mm", 45, Color.CadetBlue)
doc.Body.Children.Add(rt)
```

- C#

```
C1PrintDocument doc = c1PrintDocument1;
RenderText rt = new RenderText("Sample Shadow");
rt.Width = Unit.Auto;
rt.Style.Shadow = new Shadow(20, 100, "1mm", 45, Color.CadetBlue);
doc.Body.Children.Add(rt);
```

> **Note:** Shadows do NOT affect the objects' sizes for layout purposes.

## Object Borders

In the 2010 v1 release of **Reports for WinForms**, support was added for a new way of laying out and positioning borders. This method of laying out and positioning borders was primarilyadded for RDL compatibility but can be useful on its own. For example, now optionally borders can be centered over an object's bounds, without affecting either the object's size or the surrounding objects' positions.

The following public type facilitates this feature:

| Type | Description |
|---|---|
| BordersModeEnum | Specifies the various modes of accounting for border thickness when laying out the objects in a document. |

The BordersModeEnum includes the following members:

- **Default**: The whole border is considered to be part of the object. This is the default behavior of objects in C1PrintDocument.
- **C1Report**: The inner 1/2 of border thickness is considered to be part of the object, the outer 1/2 of border is considered to be outside of the object's space. This is the default behavior of objects in C1Report (same as in MS Access).
- **Rdl**: Border thickness is not taking into account at all when calculating objects' sizes and layout. Borders are drawn centered on objects' bounds.

## Styles

Most of the visual aspects of a **C1PrintDocument** are controlled by styles, which are an integral part of the document. The following sections describe styles in detail.

### *Classes Exposing the Style Property*

All objects having their own visual representation in the document have a style (of the type **C1.C1Preview.Style**) associated with them. Specifically, the following classes expose the Style property:

- The whole document (C1PrintDocument).

- Render objects (RenderObject and all derived classes).
- Paragraph objects (ParagraphText and ParagraphImage, derived from ParagraphObject).
- Table cells (TableCell).
- User-defined groups of cells in tables (UserCellGroup).
- Table rows and columns (TableRow and TableCol, derived from TableVector).
- Groups of table rows and columns such as table headers/footers (TableVectorGroup).

### *Inline and Non-Inline Styles*

In C1PrintDocument, there are two kinds of styles – inline and non-inline. If an object has the Style property, that property refers to the inline style of the object, which is an integral part of the object itself. An inline style cannot be removed or set – it's a read-only property that refers to the style instance which always lives together with the object. Thus, style properties can be considered to be the properties of the object itself. But, due to inheritance, styles are much more flexible and memory-efficient (for example, if none of an object's style properties have been modified from their default values, they consume almost no memory, referencing base style properties instead).

Additionally, each Style contains a collection of styles (called Children, and empty by default) which are not directly attached to any objects. Instead, those (non-inline) styles can be used as parent styles (see style properties Parent and AmbientParent) to provide values for inherited properties of other styles (including, of course, inline styles).

A style object cannot be created by itself – it is always either an inline style attached directly to a render object or other element of the document, or a member of the Children collection of another style.

So, for instance, this code will not compile:

- Visual Basic

```
Dim doc As New C1PrintDocument()
Style s = new Style() ' will not compile
s.Borders.All = New LineDef("1mm", Color.Red)
Dim rt As New RenderText("My text.")
rt.Style = s
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
Style s = new Style(); // will not compile
s.Borders.All = new LineDef("1mm", Color.Red);
RenderText rt = new RenderText("My text.");
rt.Style = s;
```

While this code will compile and achieve the desired result:

- Visual Basic

```
Dim doc As New C1PrintDocument()
Dim s As doc.Style.Children.Add()
s.Borders.All = New LineDef("1mm", Color.Red)
RenderText rt = New RenderText("My text.")
rt.Style.Parent = s
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
Style s = doc.Style.Children.Add();
s.Borders.All = new LineDef("1mm", Color.Red);
RenderText rt = new RenderText("My text.");
rt.Style.Parent = s;
```

For more about ambient and non-ambient style attributes and parents, see Ambient and Non-Ambient Style Properties.

### Ambient and Non-Ambient Style Properties

All style properties (font, colors, borders, and so on) can be classified into two groups according to their semantics: ambient and non-ambient. Ambient properties are those that affect the content of an object (for example, the text font) whereas non-ambient properties are those used to paint the object's adornments, or decorations (for example, the borders around the object). This distinction is natural and useful, as in most cases the desired behaviors (especially the inheritance rules) are different for the two groups.

For ambient properties, it is usually desirable to propagate them down the hierarchy of objects, so that setting of an ambient property on a container object affects all objects contained within (consider a table – if you set the font on the table itself, you would normally want that font to be applied to texts in all cells of that table, unless explicitly overridden at a lower level – for example, for a specific cell). This is markedly different for non-ambient properties. For instance, if you wanted to add white space before and after that table, you would set the **Spacing** property on that table – but you would not want that spacing to propagate to all cells. In C1PrintDocument, this division of style properties into ambient and non-ambient groups is built in. Basically, this means that usually just setting the property on the style of an object (without thinking too much about ambience) will do what you expect.

The complete list of all style properties, indicating which of those properties are ambient and which are not, is shown in a separate section below (see Style Properties and Their Default Values). But the following general rule applies:

**Ambient** style properties control the display of the object's content (such as text font). By default, ambient properties propagate down the objects' containment hierarchy – i.e. an ambient property set on the style of a container object is applied to all objects within that container.

**Non-ambient** style properties control the object's decorations (such as borders). By default, non-ambient properties do **not** propagate via objects' containment, but can propagate across styles via the **Style.Parent** property.

### Style Inheritance, Parent and AmbientParent

All style properties affecting the appearance of the object to which the style is applied (such as **Font**, **BackColor**, and so on) may be in one of two states: set or not set. Initially, all properties of a newly created style are not set. Their values can be queried, but are obtained from another style or object via inheritance (see below). If a property is set though, the set value is stored in the style itself, and is no longer affected by other styles via inheritance.

A style has two special properties to support style inheritance: Parent and AmbientParent. The Parent property gets or sets the style that provides values for non-ambient properties that are not set on the current style. The AmbientParent property gets or sets the style that provides values for ambient properties that are not set on the current style. By default, both parents of a newly created inline style are empty (null values in C#, Nothing in VB), whereas for styles created on the Children collection of a style, the Parent is set to the style containing that collection, while the AmbientParent is empty.

If the Parent of a style is not specified, the values of non-ambient properties are taken from static defaults (see the table below). If the AmbientParent of a style is not specified, the values of ambient properties are taken from the object which contains the object the current style applies to.

For example, if a RenderText **rt** is contained in a RenderArea **ra**, then the following rules will be used to retrieve the font (that is, an ambient style property) used to render the text:

- If **rt.Style.Font** is set, it will be used.
- Otherwise, if **rt.Style.AmbientParent** is not null, **rt.Style.AmbientParent.Font** will be used, because Font is an ambient property.
- Otherwise (if **rt.Style.AmbientParent** is null, which is the default), the font of the containing object will be used, that is **ra.Style.Font**.

In the same example, the following rules will be used to retrieve the **BackgroundImage** (a non-ambient style property) for the RenderText object:

- If **rt.Style.BackgroundImage** is set, it will be used.

- Otherwise, if **rt.Style.Parent** is not null, **rt.Style.Parent.BackgroundImage** will be used.
- Otherwise the default value for background image (that is, no image) will be used.

Please note that although by default styles in the **Children** collection of a style have their **Parent** set to the style which is the owner of that **Children** collection, this can be changed. For example, the parent of one style in the **Children** collection may be set to another style in the same collection. The **Children** collection is just a convenient place to group/store related styles, but does not really impose any limitations on the styles hierarchy.

## *Style Properties and Their Default Values*

The following table lists all style properties affecting the display of objects, indicates which of those are ambient, and specifies the default values:

| **Property name** | **Ambient** | **Default value** |
|---|---|---|
| ActiveHyperlinkAttrs | Yes | |
| BackColor | | Empty |
| BackgroundImage | | None |
| BackgroundImageAlign | | Align left/top, stretch horizontally/vertically, keep aspect ratio |
| BackgroundImageName | | None |
| Borders | | All empty |
| Brush | | None |
| CharSpacing | Yes | 0 |
| CharWidth | Yes | 100 |
| ClientAreaOnly | | False |
| FlowAlign | | Default flow alignment |
| FlowAlignChildren | | Near alignment |
| Font | Yes | Arial, 10pt |
| FontBold | Yes | False |
| FontItalic | Yes | False |
| FontName | Yes | Arial |
| FontSize | Yes | 10 |
| FontStrikeout | Yes | False |
| FontUnderline | Yes | False |
| GridLines | | None |
| HoverHyperlinkAttrs | Yes | |
| HyperlinkAttrs | Yes | Blue |
| ImageAlign | Yes | Align left/top, stretch horizontally/vertically, keep aspect ratio |
| JustifyEndOfLines | Yes | True |
| JustifyLastLine | Yes | False |
| LineSpacing | Yes | 100% |
| MeasureTrailingSpaces | Yes | False |
| MinOrphanLines | | 0 |

| | | |
|---|---|---|
| Padding | | All zeroes |
| ShapeFillBrush | | None |
| ShapeFillColor | | Transparent |
| ShapeLine | | Black, 0.5pt |
| Spacing | | All zeroes |
| TextAlignHorz | Yes | Left |
| TextAlignVert | Yes | Top |
| TextAngle | | 0 |
| TextColor | Yes | Black |
| TextIndent | | 0 |
| TextPosition | | Normal |
| VisitedHyperlinkAttrs | Yes | Magenta |
| WordWrap | Yes | True |

## *Sub-Properties of Complex Style Properties*

Some of the properties in the table in the Style Properties and Their Default Values topic contain sub-properties which can be individually set. For instance, the BackgroundImageAlign property has **AlignHorz**, **AlignVert**, and several other sub-properties. With the exception of read-only sub-properties (as is the case with fonts, which are immutable and whose individual sub-properties cannot be set), each sub-property can be set or inherited individually.

While the sub-properties of a font can not be modified, each of those sub-properties is represented by a separate root level property on a style – **FontBold**, **FontItalic**, and so on. Each of those properties can be set individually, and follows the general style inheritance rules. There is a nuance though that must be taken into consideration: if both **Font** and one of the separate font-related properties (**FontBold**, **FontItalic**, and so on.) are set, the result depends on the order in which the two properties are set. If the **Font** is set first, and then a font-related property is modified (that is, **FontItalic** is set to **True**), that modification affects the result. If, on the other hand, **FontItalic** is set to **True** first, and then **Font** is assigned to a non-italic font, the change to **FontItalic** is lost.

## *Calculated Style Properties*

In the 2009 v3 release of **Reports for WinForms**, support was added for calculated style properties. For each style property, a matching string property was been added with the same name with "Expr" appended. For example, the BackColorExpr and TextColorExpr properties are matched to the BackColor and TextColor properties, and so on for all properties.

Sub-properties of complex properties (such as ImageAlign, Borders, and so on.)  also have matching expression sub-properties. For example, the LeftExpr property is matched to Left property and so on.

**Style Expressions**

The following objects can be used in style expressions:

- RenderObject: the current style's owner render object.
- Document: the current document.
- Page (and other page related objects such as **PageNo**): the page containing the object (but see note below).
- RenderFragment: the current fragment.
- Aggregates.

- Fields, DataBinding: reference the current data source; if the style's owner is within a table, and data sources have been specified for both rows and columns, this will reference the data source defined for the columns.
- RowNumber: row number in the associated data source.
- ColFields, ColDataBinding: only accessible if the style is used within a table, references the data source defined for the columns.
- RowFields, RowDataBinding: only accessible if the style is used within a table, references the data source defined for the rows.

### Converted Types

When a calculated style property value is assigned to the real value that will be used to render the object, the types are converted according to the following rules:

- If the target property is numeric (int, float, and so on), the calculated value is converted to the required numeric type (converted from string, rounded, and so on) as necessary.
- If the target property is a Unit (for example, spacings), and the expression yields a number, the unit is created using the following constructor: new Unit(Document.DefaultUnitType, value). If the expression yields a string, that string is parsed using the normal Unit rules.
- In all other cases, an attempt is made to convert the value to the target type using the **TypeConverter**.
- Finally, if the expression yielded null, the parent style value is used as if the property has not been specified on the current style at all (such as the default behavior for unspecified style properties).

### Page References

The following is an important note related to page references. Style expressions may reference the current page, for example:

```
ro.Style.BackColor = "iif(PageCount < 3, Color.Red, Color.Blue)";
```

Such expressions cannot be calculated when the document is generated. Thus, during generation such expressions are ignored (default values are used), and the values are only calculated when the actual page that contains the object is being rendered (for example, for drawing in the preview, exporting and so on).

As the result, style expressions that depend on pagination AND would affect the layout of the document may yield unexpected and undesirable results. For example, if the following expression is used for font size:

```
ro.Style.FontSize = "iif(PageCount < 3, 20, 30)";
```

The above expression would be ignored during generation, as the result the rendered text will most probably be too large for the calculated object's size and clipping will occur.

## *Paragraph Object Styles*

The RenderParagraph object (as all render objects) has the **Style** property, which can be used to set style properties applied to the paragraph as a whole. Individual paragraph objects (ParagraphText and ParagraphImage) also have styles, but the set of style properties that are applied to paragraph objects is limited to the following properties:

- BackColor
- Brush (only if it is a solid brush)
- Font (and font-related properties such as FontBold, and so on.)
- HoverHyperlinkAttrs
- TextColor
- TextPosition
- VisitedHyperlinkAttrs

### *Table Styles*

In tables, the number of styles affecting the display of objects increases dramatically. In addition to the normal containment (a table, as all other elements of a document, is always contained either within another render object, or within the body of the document at the top level), object in tables also belong to at least a cell, a row and a column, all of which have their own styles. Additionally, an object can belong to a number of table element groups, which complicates the things even more. How styles in tables work is described in more detail in the Styles in Tables topic.

## Tables

Tables are represented by instances of the RenderTable class. To create a table, just invoke its constructor, for example like this:

- Visual Basic

```
Dim rt1 As New C1.C1Preview.RenderTable()
```

- C#

```
RenderTable rt1 = new RenderTable();
```

C1PrintDocument tables follow the model of Microsoft Excel. Though a newly created table is physically empty (that is, it does not take much space in memory), logically it is infinite: you can access any element (cell, row, or column) of a table without first adding rows or columns, and writing to that element will logically create all elements preceding it. For instance, setting the text of the cell of an empty table at row index 9 and column index 3 will make the table grow to 10 rows and 4 columns.

To add content to a table, you must fill the cells with data. This can be done in one of the following ways:

- By setting the cell's RenderObject property to an arbitrary render object. This will insert the specified render object into that cell. Any render object can be added to a cell, including another table, thus allowing nested tables.
- By setting the cell's Text property to a string. This is actually a handy shortcut to create text-only tables, which internally creates a new RenderText object, sets the cell's RenderObject property to that RenderText, and sets that object's Text property to the specified string.

So, for example, the following code snippet will create a table with 10 rows and 4 columns:

- Visual Basic

```
Dim rt1 As New C1.C1Preview.RenderTable()
Dim row As Integer = 0
Do While (row < 10)
    Dim col As Integer = 0
    Do While (col < 4)
        rt1.Cells(row, col).Text = String.Format( _
          "Text in cell({0}, {1})", row, col)
        col += 1
    Loop
    row += 1
Loop
```

- C#

```
RenderTable rt1 = new RenderTable();
for (int row = 0; row < 10; ++row)
{
    for (int col = 0; col < 4; ++col)
        rt1.Cells[row, col].Text = string.Format(
          "Text in cell({0}, {1})", row, col);
}
```

At any time, you can find out the actual current size of a table by querying the values of properties **Cols.Count** (which returns the current number of columns) and **Rows.Count** (which returns the current number of rows).

### Accessing Cells, Columns and Rows

As can be seen from the sample code in the Tables topic, all cells in a table are represented by the Cells collection, which has the type TableCellCollection. Elements in this collection representing individual cells have the type TableCell. To access any cell in a table, the Cells collection can be indexed by the cell's row and column like this:

- Visual Basic

```
Dim rt As New C1.C1Preview.RenderTable()
…
' get cell at row 10, column 4:
Dim tc as TableCell = rt.Cells(10, 4)
```

- C#

```
RenderTable rt = new RenderTable();
…
// get cell at row 10, column 4:
TableCell tc = rt.Cells[10, 4];
```

Table columns are accessed via the Cols collection, which has the type TableColCollection, and contains elements of the type TableCol. As with cells, just "touching" a column will create it. For instance, if you set a property of a column's Style, that column will be created if it did not exist already.

Table rows are accessed via the Rows collection, which has the type TableRowCollection, and contains elements of the type TableRow. As with cells and columns, just "touching" a row will create it. For instance, if you set the Height of a row it (and all rows before it) will be automatically created.

Please note, though, that all table rows that do not contain cells with some actual content will have a zero height, so will not be visible when the table is rendered.

### Table and Column Width, Row Height

Both rows and columns in **C1PrintDocument** tables can be auto-sized but the default behavior is different for rows and columns. By default, rows have auto-height (calculated based on the content of cells in the row), while columns' widths are fixed. The default width of a RenderTable is equal to the width of its parent, and that width is shared equally between all columns. So the following code will create a page-wide table, with 3 equally wide columns, and 10 rows with the heights that auto-adjust to the height of the cells' content:

- Visual Basic

```
Dim rt As New C1.C1Preview.RenderTable()
rt.Style.GridLines.All = LineDef.Default
Dim row As Integer = 0
Do While (row < 10)
  Dim col As Integer = 0
    Do While (col < 3)
      rt.Cells(row, col).Text = String.Format( _
          "Cell({0},{1})", row, col)
      col += 1
    Loop
  row += 1
Loop
doc.Body.Children.Add(rt)
```

- C#

```
RenderTable rt = new RenderTable();
rt.Style.GridLines.All = LineDef.Default;
for (int row = 0; row < 10; ++row)
  for (int col = 0; col < 3; ++col)
```

```
    rt.Cells[row, col].Text = string.Format(
        "Cell({0}, {1})", row, col);
doc.Body.Children.Add(rt);
```

To make a fully auto-sized table, as compared to the default settings, two things must be done:

- The width of the whole table must be set to **Auto** (either the string "auto", or the static field **Unit.Auto**) and
- The table's RenderTable.ColumnSizingMode must be set to TableSizingModeEnum.Auto.

Here's the modified code:

- Visual Basic

```
Dim rt As New C1.C1Preview.RenderTable()
rt.Style.GridLines.All = LineDef.Default
Dim row As Integer = 0
Do While (row < 10)
  Dim col As Integer = 0
    Do While (col < 3)
      rt.Cells(row, col).Text = String.Format( _
          "Cell({0},{1})", row, col)
      col += 1
    Loop
  row += 1
Loop
rt.Width = Unit.Auto
rt.ColumnSizingMode = TableSizingModeEnum.Auto
doc.Body.Children.Add(rt)
```

- C#

```
RenderTable rt = new RenderTable();
rt.Style.GridLines.All = LineDef.Default;
for (int row = 0; row < 10; ++row)
  for (int col = 0; col < 3; ++col)
    rt.Cells[row, col].Text = string.Format(
        "Cell({0}, {1})", row, col);
rt.Width = Unit.Auto;
rt.ColumnSizingMode = TableSizingModeEnum.Auto;
doc.Body.Children.Add(rt);
```

The modified code makes each column of the table only as wide as needed to accommodate all text within cells.

## *Groups of Rows and Columns, Headers and Footers*

Element groups are a very powerful feature of tables. Groups allow accessing several elements of a table as a whole (for example, the style can be set on a group as if it were a single element). Supported are groups of columns, groups of rows, and groups of cells.

To access a group of rows, use the collection RowGroups (which has the type TableVectorGroupCollection). Elements of that collection have the type TableVectorGroup, with some useful properties defined on that type. One of the more interesting of those properties is ColumnHeader. That property allows assigning a group of rows to be a table header, to be repeated at the top of each new page or page column. A related property is ColumnFooter, which allows to assign a group of rows to be a table footer, again repeated either at the end of each page or page column.

The following line of code shows how to assign the first 2 rows of a table to be the table header, repeated after each page or column break (rt1 here is a RenderTable object):

- Visual Basic
  ```
  rt1.RowGroups(0, 2).Header = C1.C1Preview.TableHeaderEnum.Page
  ```
- C#
  ```
  rt1.RowGroups[0, 2].Header = C1.C1Preview.TableHeaderEnum.Page;
  ```

As seen above, the indexer on the TableVectorGroupCollection class accepts two integers. The first value is the index of the first row included in the group (0 in the code example above). The second value is the count of rows in the group (2 in the code example above).

To access a group of columns, the collection ColGroups should be used. It has the same type as the row groups' collection (TableVectorGroupCollection), and provides the same functionality. Of particular interest is the ability to assign a group of columns to be a **vertical** table header or footer. C1PrintDocument supports "horizontal" (or "extension") pages, which allow wide objects to span several pages horizontally. To allow an object (for example, a table) to span several pages horizontally, set its SplitHorzBehavior to a value other than SplitBehaviorEnum.**Never**. If the object's width is wider than the page width, it will be split into several horizontal pages. In particular, a wide table can be split in this way. To make a group of columns repeat along the left edge of each page, set the group's ColumnHeader property to **True**. To make a group of columns repeat along the right edge of each page, set the group's ColumnFooter property to **True**.

**Note:** Although any group of rows (or columns) of a table can be assigned to be the footer, normally you would want to include only the last rows (or columns) of the table into the footer group. This will ensure that the footer behaves as a normal footer – that is appears only at the bottom (or right edge) of pages, and also appears at the end of the table. (If, for example, you assign the first row of a table to be the footer, it will still appear at the beginning of the table, and also will not print at the end of the table.)

Here is an example of code that creates a table with 100 rows and 10 columns, sets the width of the table to **Auto**, explicitly sets the width of each column to **1 inch**, and also assigns horizontal and vertical table headers and footers:

- Visual Basic
  ```
  ' Create and fill the table.
  Dim rt1 As C1.C1Preview.RenderTable = New C1.C1Preview.RenderTable()
  Dim row As Integer = 0
  Dim col As Integer
  Do While (row < 100)
      col = 0
      Do While (col < 6)
          rt1.Cells(row, col).Text = String.Format("Text in cell({0}, {1})",
  row, col)
          col += 1
      Loop
      row += 1
  Loop

  ' Set the table and columns' widths.
  rt1.Width = C1.C1Preview.Unit.Auto
  col = 0
  Do While (col < 6)
      rt1.Cols(col).Width = "1in"
      col += 1
  Loop

  ' Assign the first 2 rows as the header and set the background.
  rt1.RowGroups(0, 2).PageHeader = True
  rt1.RowGroups(0, 2).Style.BackColor = Color.Red

  ' Assign the last 2 rows as the footer and set the background.
  ```

```
rt1.RowGroups(98, 2).PageFooter = True
rt1.RowGroups(98, 2).Style.BackColor = Color.Blue

' Assign the first column as the header.
rt1.ColGroups(0, 1).PageHeader = True
rt1.ColGroups(0, 1).Style.BackColor = Color.BlueViolet

' Assign the last column as the footer.
rt1.ColGroups(5, 1).PageFooter = True
rt1.ColGroups(5, 1).Style.BackColor = Color.BurlyWood
```

- C#

```
// Create and fill the table.
RenderTable rt1 = new RenderTable();
for (int row = 0; row < 100; ++row)
{
    for (int col = 0; col < 6; ++col)
    {
        rt1.Cells[row, col].Text = string.Format("Text in cell({0}, {1})", 
row, col);
    }
}

// Set the table and columns' widths.
rt1.Width = Unit.Auto;
for (int col = 0; col < 6; ++col)
{
    rt1.Cols[col].Width = "1in";
}

// Assign the first 2 rows as the header and set the background.
rt1.RowGroups[0, 2].PageHeader = true;
rt1.RowGroups[0, 2].Style.BackColor = Color.Red;

// Assign the last 2 rows as the footer and set the background.
rt1.RowGroups[98, 2].PageFooter = true;
rt1.RowGroups[98, 2]. Style.BackColor = Color.Blue;

// Assign the first column as the header.
rt1.ColGroups[0, 1].PageHeader = true;
rt1.ColGroups[0, 1].Style.BackColor = Color.BlueViolet;

// Assign the last column as the footer.
rt1.ColGroups[5, 1].PageFooter = true;
rt1.ColGroups[5, 1].Style.BackColor = Color.BurlyWood;
```

In this sample, background color is used to highlight row and column groups.

### User Cell Groups

Cells, even cells that are not adjacent to each other in the table, can be united into groups. You can then set styles on all cells in a group with a single command. To define a user cell group:

1. Create an object of the type UserCellGroup. This class has several overloaded constructors which allow specifying the coordinates of cells to be included in the group (all cells must be added to the group in the constructor).
2. Add the created UserCellGroup object to the UserCellGroups collection of the table.

3. Now you can set the style on the group. This will affect all cells in the group.

### Styles in Tables

Though table cells, columns, and rows are not render objects (they do not derive from the RenderObject class), they do have some properties similar to those of render objects. In particular, they all have the **Style** property.

Manipulating styles affects the corresponding element and all its content. Setting the style of a row will affect all cells in that row. Setting the style of a column will affect all cells in that column. The style of the cell at the intersection of that row and column will be a combination of the styles specified for the row and the column. If the same style property is set on both the row and the column, the column will take precedence.

Additionally, groups (groups of rows, groups of columns, and user cell groups) all have their own styles, which also affect the display of data in cells, and the display of table rows and columns.

The following rules govern the application of styles in tables:

Ambient style properties propagate down through the table elements (the whole table, row and column groups, cell groups, individual rows and columns, and individual cells) based on the "geometric" containment, similar to how ambient style properties propagate down render objects' containment outside of tables.

Ambient properties affect the cells' content, without affecting those container elements. For instance, setting the font on the style of a whole table affects all text within that table unless its font was explicitly set at a lower level. Similarly, setting the font on the style of a row within that table affects the font of all cells within that row.

When a specific ambient property is changed on two or more of the table elements involved, the following order of precedence is used to calculate the effective value of the property used to draw the cell:

- Cell's own style (has the highest priority)
- UserCellGroup style, if the cell is included in such
- Column style
- Columns group style, if any
- Row style
- Rows group style, if any
- Table style (has the least priority)

Non-ambient properties set on styles of table elements listed above (whole table, row, column and cell groups, rows, columns and cells) are applied to those elements themselves, without affecting the content of the cells, even though such elements (with the exception of the whole table) are not render objects. For instance, to draw a border around a row in a table, set the **Style.Borders** on the row to the desired value.

To set a non-ambient style property on all cells in a table, use **RenderTable.CellStyle**. If specified, that style is effectively used as the parent for the style of render object within cells.

The **CellStyle** property is also defined on rows, columns, and groups of table elements, and if specified all those styles will affect non-ambient properties of the object within the cell. For instance, to set the background image for all cells in a table, set the table's **CellStyle.BackgroundImage**. This will repeat that image in all cells in a table, while assigning the same image to the table's **Style.BackgroundImage** will make that image the background for the whole table (the difference is apparent if the image is stretched in both cases).

## Anchors and Hyperlinks

**Reports for WinForms** hyperlinks. Hyperlinks can be attached to render objects (RenderObject and derived classes), and paragraph objects (ParagraphObject and derived classes), and can be linked to:

- An anchor within the current document.
- An anchor within another C1PrintDocument.

- A location within the current document.
- An external file.
- A page within the current document.
- A user event.

Hyperlinks are supported by the preview controls included in C1.Win.C1Preview assembly (C1PreviewPane, C1PrintPreviewControl and C1PrintPreviewDialog). When a document with hyperlinks in it is previewed, and the mouse hovers over a hyperlink, the cursor changes to a hand. Clicking the hyperlink will, depending on the target of the link, either jump to another location within the document, open a different document and jump to a location in it, open an external file, or invoke the user event.

> **Note:** The sample code fragments in the following topics assume that the "using C1.C1Preview" directive (in C# syntax; or an equivalent for other languages) has been inserted into the file, so that instead of fully qualified type names (such as C1.C1Preview.RenderText) we can use just the class name part (RenderText).

### Adding a Hyperlink to an Anchor within the Same Document

To link one part of your document to another, you have to do two things:

- Mark the location (called an **anchor**) where you want the link to point.
- Add a link to that location (a **hyperlink**) to another part of the document (you can have several hyperlinks pointing to the same anchor, of course).

To create an anchor on a render object, you can add an element (of the type C1Anchor) to the Anchors collection of that render object. For example, if **rt** is a RenderTable you can write:

- Visual Basic
```
rt.Anchors.Add(New C1.C1Preview.C1Anchor("anchor1"))
```
- C#
```
rt.Anchors.Add(new C1Anchor("anchor1"));
```

This will define an anchor with the name **anchor1** (the name used to reference the anchor) on the render table.

To link another render object, for example a RenderText, to that anchor, you can write:

- Visual Basic
```
Dim rtxt As New C1.C1Preview.RenderText()
rtxt.Text = "Link to anchor1"
rtxt.Hyperlink = New C1.C1Preview.C1Hyperlink("anchor1")
```
- C#
```
RenderText rtxt = new RenderText();
rtxt.Text = "Link to anchor1";
rtxt.Hyperlink = new C1Hyperlink("anchor1");
```

Of course, you must add both involved render objects (the one containing the anchor, and the one with the hyperlink) to the document.

Hyperlink is a property of the RenderObject class, which is the base class for all render objects, so in exactly the same manner as shown above, any render object may be turned into a hyperlink by setting that property.

### Adding a Hyperlink to an Anchor in a Different C1PrintDocument

To link a location in one document to a location in another, you must do the following:

- As described above, add an anchor to the target document, generate that document and save it as a C1D file on your disk. You can save the document using the **Save** button of the preview control, or in code using the **Save** method on the document itself.

- Add a link pointing to that anchor to another document, in a way very similar to how an internal link is added. The only difference is that in addition to the target' anchor name you must also provide the name of the file containing the document.

Here is the text of a complete program that creates a document with an anchor in it, and saves it in a disk file (myDocument1.c1d). It then creates another document, adds a link to the anchor in the first document to it, and shows the second document in a preview dialog box:

- Visual Basic

```
' Make target document with an anchor.
Dim targetDoc As New C1.C1Preview.C1PrintDocument
Dim rt1 As New C1.C1Preview.RenderText("This is anchor1 in myDocument1.")
rt1.Anchors.Add(New C1.C1Preview.C1Anchor("anchor1"))
targetDoc.Body.Children.Add(rt1)
targetDoc.Generate()
targetDoc.Save("c:\myDocument1.c1d")

' Make document with a hyperlink to the anchor.
Dim sourceDoc As New C1.C1Preview.C1PrintDocument
Dim rt2 As New C1.C1Preview.RenderText("This is hyperlink to
myDocument1.")
Dim linkTarget As C1.C1Preview.C1LinkTarget = New
C1.C1Preview.C1LinkTargetExternalAnchor("c:\myDocument1.c1d", "anchor1")
rt2.Hyperlink = New C1.C1Preview.C1Hyperlink(linkTarget)
sourceDoc.Body.Children.Add(rt2)
sourceDoc.Generate()

' Show document with hyperlink in preview.
Dim preview As New C1.Win.C1Preview.C1PrintPreviewDialog()
preview.Document = sourceDoc
preview.ShowDialog()
```

- C#

```
// Make target document with an anchor.
C1PrintDocument targetDoc = new C1PrintDocument();
RenderText rt1 = new RenderText("This is anchor1 in myDocument1.");
rt1.Anchors.Add(new C1Anchor("anchor1"));
targetDoc.Body.Children.Add(rt1);
targetDoc.Generate();
targetDoc.Save(@"c:\myDocument1.c1d");

// Make document with a hyperlink to the anchor.
C1PrintDocument sourceDoc = new C1PrintDocument();
RenderText rt2 = new RenderText("This is hyperlink to myDocument1.");
C1LinkTarget linkTarget = new
C1LinkTargetExternalAnchor(@"c:\myDocument1.c1d", "anchor1");
rt2.Hyperlink = new C1Hyperlink(linkTarget);
sourceDoc.Body.Children.Add(rt2);
sourceDoc.Generate();

// Show document with hyperlink in preview.
C1PrintPreviewDialog preview = new C1PrintPreviewDialog();
preview.Document = sourceDoc;
preview.ShowDialog();
```

Note the following:

- The anchor is created in exactly the same manner as for links within the same document. In fact, there is no difference; the same anchor may be the target of links both from the same document and from other documents.
- To save the document, the **Save** method is used. This method saves the document in native **C1PrintDocument** format, the default extension for which is **C1D**. Files saved in that format can be later loaded into a **C1PrintDocument** object for further processing, or previewed using the ComponentOne print preview control.
- Before creating the hyperlink, a link target object must be created which is then passed to the hyperlink constructor. Several link target types are provided, derived from the C1LinkTarget base class. For external anchors, the C1LinkTargetExternalAnchor type is used. The link target contains information needed to process the jump to the link, in this case the document filename and the name of the anchor in it.

### *Adding a Hyperlink to a Location Within the Current Document*

You can add a link to an object within the current document without creating an anchor. Instead, you can use the C1LinkTargetDocumentLocation link target created directly on a render object, like this, where **ro1** is an arbitrary render object in the current document:

- Visual Basic
```
Dim linkTarget = New C1.C1Preview.C1LinkTargetDocumentLocation(ro1)
```
- C#
```
C1LinkTarget linkTarget = new C1LinkTargetDocumentLocation(ro1);
```

Setting this link target on a hyperlink will make that hyperlink jump to the specified render object when the object owning the hyperlink is clicked. If, for example, **ro2** is a render object that you want to turn into a hyperlink, the following code will link it to the location of **ro1** on which the **linkTarget** was created as shown in the code snippet above:

- Visual Basic
```
rt2.Hyperlink = New C1.C1Preview.C1Hyperlink()
rt2.Hyperlink.LinkTarget = linkTarget
```
- C#
```
rt2.Hyperlink = new C1Hyperlink();
rt2.Hyperlink.LinkTarget = linkTarget;
```

Note that in this example, the LinkTarget property of the hyperlink was set after the hyperlink has been created.

### *Adding a Hyperlink to an External File*

A hyperlink to an external file differs from a link to an external anchor by the link target. The link target class for an external file link is called C1LinkTargetFile. Clicking such a link will use the Windows shell to open that file. For instance, if in the sample from the preceding section, you replace the line creating the external anchor link target with the following line:

- Visual Basic
```
Dim linkTarget = New C1.C1Preview.C1LinkTargetFile("c:\")
```
- C#
```
C1LinkTarget linkTarget = new C1LinkTargetFile(@"c:\");
```

Clicking on that link will open the Windows Explorer on the root directory of the C: drive.

Again, here is a complete program:

- Visual Basic
```
' Make document with a hyperlink to external file.
Dim doc As New C1.C1Preview.C1PrintDocument
Dim rt As New C1.C1Preview.RenderText("Explore drive C:...")
Dim linkTarget As C1.C1Preview.C1LinkTarget = New
C1.C1Preview.C1LinkTargetFile("c:\")
rt.Hyperlink = New C1.C1Preview.C1Hyperlink(linkTarget)
doc.Body.Children.Add(rt)
doc.Generate()

' Show document with hyperlink in preview.
Dim preview As New C1.Win.C1Preview.C1PrintPreviewDialog()
preview.Document = doc
preview.ShowDialog()
```

- C#

```
// Make document with a hyperlink to external file.
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("Explore drive C:...");
C1LinkTarget linkTarget = new C1LinkTargetFile(@"c:\");
rt.Hyperlink = new C1Hyperlink(linkTarget);
doc.Body.Children.Add(rt);
doc.Generate();

// Show document with hyperlink in preview.
C1PrintPreviewDialog preview = new C1PrintPreviewDialog();
preview.Document = doc;
preview.ShowDialog();
```

### *Adding a Hyperlink to a Page in the Same Document*

You can add hyperlinks to other pages in the same document without defining anchors, using the C1LinkTargetPage link target. The following page jumps are supported:

- To the first page of the document.
- To the last page of the document.
- To the previous page.
- To the next page.
- To a page specified by its absolute page number.
- To a page specified by an offset relative to the current page.

For instance, to make a link target jumping to the first page in the document, the following line of code may be used:

- Visual Basic

```
Dim linkTarget = New
C1.C1Preview.C1LinkTargetPage(C1.C1Preview.PageJumpTypeEnum.First)
```

- C#

```
C1LinkTarget linkTarget = new C1LinkTargetPage(PageJumpTypeEnum.First);
```

Here, **PageJumpTypeEnum** specifies the type of the page jump (of course, for jumps requiring an absolute or a relative page number, you must use a variant of the constructor accepting that).

This feature allows you to provide simple means of navigating the document in the document itself. For example, you can add DVD player-like controls (go to first page, go to previous page, go to next page, go to last page) to the document footer.

### Adding a Hyperlink to a User Event

Finally, you may add a hyperlink that will fire an event on the C1PreviewPane, to be handled by your code. For this, C1LinkTargetUser should be used. Here is a complete example illustrating the concept:

- Visual Basic

```
Private Sub UserLinkSetup()

    ' Make document with a user hyperlink.
    Dim doc As New C1.C1Preview.C1PrintDocument
    Dim rt As New C1.C1Preview.RenderText("Click this to show message 
box...")
    Dim linkTarget As C1.C1Preview.C1LinkTarget = New 
C1.C1Preview.C1LinkTargetUser
    rt.Hyperlink = New C1.C1Preview.C1Hyperlink(linkTarget)
    rt.Hyperlink.UserData = "My hyperlnk user data"
    doc.Body.Children.Add(rt)
    doc.Generate()

    ' Create the preview.
    Dim preview As New C1.Win.C1Preview.C1PrintPreviewDialog()

    ' Attach an event handler to the UserHyperlinkJump event.
    AddHandler preview.PreviewPane.UserHyperlinkJump, New 
C1.Win.C1Preview.HyperlinkEventHandler(AddressOf 
Me.C1PreviewPane1_UserHyperlinkJump)

    ' Preview the document.
    preview.Document = doc
    preview.ShowDialog()End Sub

Private Sub C1PreviewPane1_UserHyperlinkJump(ByVal sender As Object, ByVal 
e As C1.Win.C1Preview.HyperlinkEventArgs) Handles 
C1PreviewPane1.UserHyperlinkJump
    MessageBox.Show(e.Hyperlink.UserData.ToString())
End Sub
```

- C#

```
private void UserLinkSetup()
{

    // Make document with a user hyperlink.
    C1PrintDocument doc = new C1PrintDocument();
    RenderText rt = new RenderText("Click this to show message box...");
    C1LinkTarget linkTarget = new C1LinkTargetUser();
    rt.Hyperlink = new C1Hyperlink(linkTarget);
    rt.Hyperlink.UserData = "My hyperlnk user data";
    doc.Body.Children.Add(rt);
    doc.Generate();

    // Create the preview.
    C1PrintPreviewDialog preview = new C1PrintPreviewDialog();

    // Attach an event handler to the UserHyperlinkJump event.
    preview.PreviewPane.UserHyperlinkJump += new 
HyperlinkEventHandler(PreviewPane_UserHyperlinkJump);

    // Preview the document.
    preview.Document = doc;
    preview.ShowDialog();
}

private void PreviewPane_UserHyperlinkJump(object sender,
HyperlinkEventArgs e)
{
    MessageBox.Show(e.Hyperlink.UserData.ToString());
}
```

This example will show the message box with the string that was assigned to the UserData property of the hyperlink when the hyperlink is clicked (in this case, "My hyperlink user data").

### *Link Target Classes Hierarchy*

To conclude the section on hyperlinks, here is the hierarchy of link target classes:

| Class | | | Description |
|---|---|---|---|
| *C1LinkTarget* | | | The base class for the whole hierarchy. |
| | C1LinkTargetAnchor | | Describes a target which is an anchor in the current document. |
| | | C1LinkTargetExternalAnchor | Describes a target which is an anchor in a different document. |
| | C1LinkTargetDocumentLocation | | Describes a target which is a render object. |
| | C1LinkTargetFile | | Describes a target which is an external file to be opened by the OS shell. |
| | C1LinkTargetPage | | Describes a target which is a page in the current document. |
| | C1LinkTargetUser | | Describes a target which invokes a user event handler on the **C1PreviewPane**. |

## Expressions, Scripts, Tags

Expressions (or scripts – the two terms are used interchangeably here) can be used in various places throughout the document. Mark an expression by surrounding it by square brackets, as in the following code:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.Body.Children.Add(New RenderText("2 + 2 = [2+2]"))
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.Body.Children.Add(new RenderText("2 + 2 = [2+2]"));
```

This code will produce the following text in the generated document:

```
2 + 2 = 4
```

In this case, the "`[2+2]`" is interpreted as an expression, and calculated to produce the resulting text.

**Note:** If the expression parser cannot parse the text included in square brackets, such text is not interpreted as an expression, and is inserted in the document "as is".

### Tags

Tags are closely related to expressions. In fact, tags are variables that can be used in expressions, or simply as expressions. In the simplest case, tags allow you to use a placeholder where you want to insert a certain string but do not know the value of that string yet. The typical example of a tag is the page number – you want to print the page number but do not know yet what it is going to be or realize it may change when the document is regenerated.

A tag has two main properties: **Name** and **Value**. The name is used to identify the tag, while the value is what the tag is replaced with.

C1PrintDocument provides two kinds of tags: predefined and custom. Predefined tags are:

- [PageNo] – replaced with the current page number.
- [PageCount] – replaced with the total number of pages.
- [PageX] – replaced with the current horizontal page number.
- [PageXCount] – replaced with the total number of horizontal pages.
- [PageY] – replaced with the current vertical page number (if there are no horizontal pages, this is equivalent to [PageNo]).
- [PageYCount] – replaced with the total number of vertical pages (if there are no horizontal pages, this is equivalent to [PageCount]).

Custom tags are stored in the **Tags** collection of the document. To add a tag to that collection, you may use the following code:

- Visual Basic
```
doc.Tags.Add(New C1.C1Preview.Tag("tag1", "tag value"))
```
- C#
```
doc.Tags.Add(new C1.C1Preview.Tag("tag1", "tag value"));
```

The value of the tag may be left unspecified when the tag is created, and may be specified at some point later (even after that tag was used in the document).

To use a tag, insert its name in square brackets in the text where you want the tag value to appear, for example, like this:

- Visual Basic
```
Dim rt As New C1.C1Preview.RenderText()
rt.Text = "The value of tag1 will appear here: [tag1]."
```
- C#
```
RenderText rt = new RenderText();
Rt.Text = "The value of tag1 will appear here: [tag1].";
```

### Tags/expressions syntax

You can change the square brackets used to include tags or scripts in text to arbitrary strings, if desired, via the document's **TagOpenParen** and **TagCloseParen** properties. This may be a good idea if your document contains a lot of square brackets – by default, they will all trigger expression parsing which can consume a lot of resources even if not affecting the result visually. So, for instance this:

- Visual Basic
```
doc.TagOpenParen = "@@@["
doc.TagCloseParen = "@@@]"
```
- C#
```
doc.TagOpenParen = "@@@[";
doc.TagCloseParen = "@@@]";
```

will ensure that only strings surrounded by "@@@[" and "@@@]" are interpreted as expressions.

The expression brackets can also be escaped by preceding them with the backslash symbol, for instance this code:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.Body.Children.Add(new RenderText("2 + 2 = \[2+2\]"))
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.Body.Children.Add(new RenderText("2 + 2 = \[2+2\]"));
```

will produce the following text in the generated document:

```
2 + 2 = [2+2]
```

because the brackets were escaped.

The property **TagEscapeString** on the document can be used to change the escape symbol to an arbitrary string.

### *Editing Tag Values at Run Time*

You can show a form for the end-user to view and edit tag values for tags contained in the Tags collection of a C1PrintDocument component. Several new members support this option allowing you to create a customized form with specific tags displayed.

You have several options with the **Tags** form that you create. You can:

- Allow the user to edit every tag each time a C1PrintDocument is generated. For more information, see Displaying All Tags.
- Let the user edit some but not all tags, set the **Flags** property to **None** on tags you do not want to be edited by the user.
- Finally, you can choose when the end user is presented with the **Tags** dialog box. set ShowTagsInputDialog to **False** on the document, and call the EditTags() method on the document in your own code when you want the user to be presented with the tags input dialog.

## Displaying All Tags

By default the ShowTagsInputDialog property is set to **False** and the **Tags** dialog box is not displayed. To allow the user to input all tags each time a C1PrintDocument is generated, set the ShowTagsInputDialog property to **True** on the document. Any tags you've added to the document's Tags collection will then be automatically presented in a dialog box to the user each time the document is about to be generated. This will give the end-user the opportunity to edit the tags' values in the **Tags** dialog box.

For example, the following code in the **Form_Load** event adds three tags to the document and text values for those tags:

- Visual Basic

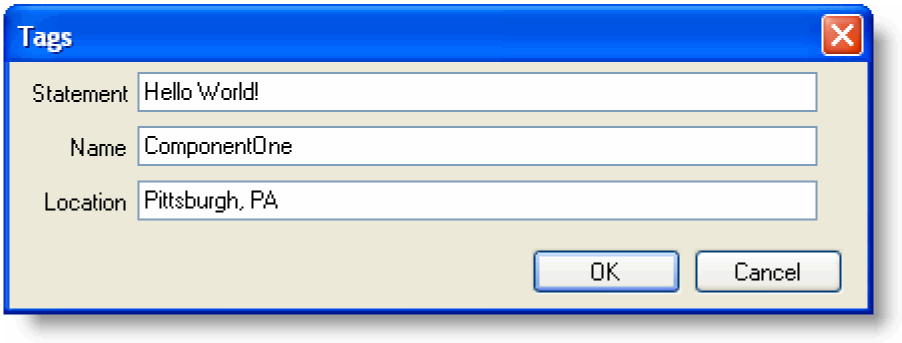
```
Dim doc As New C1PrintDocument()
Me.C1PrintPreviewControl1.Document = doc
' Show the Tags dialog box on document generation.
doc.ShowTagsInputDialog = True
' Create tags that will be shown in the Tags dialog box.
doc.Tags.Add(New C1.C1Preview.Tag("Statement", "Hello World!"))
doc.Tags.Add(New C1.C1Preview.Tag("Name", "ComponentOne"))
doc.Tags.Add(New C1.C1Preview.Tag("Location", "Pittsburgh, PA"))
' Add tags to the document and generate.
Dim rt As New C1.C1Preview.RenderText()
rt.Text = "[Statement] My name is [Name] and my current location is
[Location]."
```

```
doc.Body.Children.Add(rt)
doc.Generate()
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
this.c1PrintPreviewControl1.Document = doc;
// Show the Tags dialog box on document generation.
doc.ShowTagsInputDialog = true;
// Create tags that will be shown in the Tags dialog box.
doc.Tags.Add(new C1.C1Preview.Tag("Statement", "Hello World!"));
doc.Tags.Add(new C1.C1Preview.Tag("Name", "ComponentOne"));
doc.Tags.Add(new C1.C1Preview.Tag("Location", "Pittsburgh, PA"));
// Add tags to the document and generate.
C1.C1Preview.RenderText rt = new C1.C1Preview.RenderText();
rt.Text = "[Statement] My name is [Name] and my current location is
[Location].";
doc.Body.Children.Add(rt);
doc.Generate();
```

When the application is run, the following dialog box is displayed before the document is generated:



Statement: Hello World!

Name: ComponentOne

Location: Pittsburgh, PA

Changing the text in any of the textboxes in the **Tags** dialog box will change the text that appears in the generated document. If the default text is left, the following will produce the following text in the generated document:

```
Hello World! My name is ComponentOne and I'm currently located in
Pittsburgh, PA.
```

## Displaying Specific Tags

When the ShowTagsInputDialog property is set to **True** all tags are displayed by default in the **Tags** dialog box. You can prevent users from editing specific tags by using the **Tag.ShowInDialog** property. To let users edit some but not all tags, set the **Flags** property to **None** on tags you do not want to be edited.

For example, the following code in the **Form_Load** event adds three tags to the document, one of which cannot be edited:

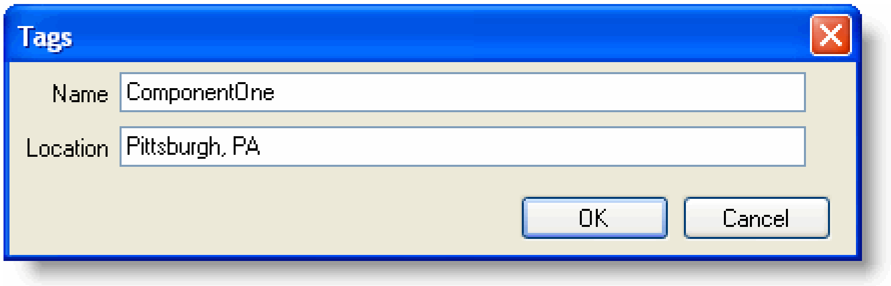
- Visual Basic

```
Dim doc As New C1PrintDocument()
Me.C1PrintPreviewControl1.Document = doc
' Show the Tags dialog box on document generation.
doc.ShowTagsInputDialog = True
' Create a tag but do not show it in the Tags dialog box.
doc.Tags.Add(New C1.C1Preview.Tag("Statement", "Hello World!"))
doc.Tags("Statement").ShowInDialog = False
' Create tags that will be shown.
doc.Tags.Add(New C1.C1Preview.Tag("Name", "ComponentOne"))
doc.Tags.Add(New C1.C1Preview.Tag("Location", "Pittsburgh, PA"))
' Add tags to the document and generate.
Dim rt As New C1.C1Preview.RenderText()
rt.Text = "[Statement] My name is [Name] and my current location is
[Location]."
doc.Body.Children.Add(rt)
doc.Generate()
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
this.c1PrintPreviewControl1.Document = doc;
// Show the Tags dialog box on document generation.
doc.ShowTagsInputDialog = true;
// Create a tag but do not show it in the Tags dialog box.
doc.Tags.Add(new C1.C1Preview.Tag("Statement", "Hello World!"));
doc.Tags["Statement"].ShowInDialog = false;
// Create tags that will be shown.
doc.Tags.Add(new C1.C1Preview.Tag("Name", "ComponentOne"));
doc.Tags.Add(new C1.C1Preview.Tag("Location", "Pittsburgh, PA"));
// Add tags to the document and generate.
C1.C1Preview.RenderText rt = new C1.C1Preview.RenderText();
rt.Text = "[Statement] My name is [Name] and my current location is
[Location].";
doc.Body.Children.Add(rt);
doc.Generate();
```

When the application is run, the following dialog box is displayed before the document is generated:



Name: ComponentOne

Location: Pittsburgh, PA

Changing the text in any of the textboxes in the **Tags** dialog box will change the text that appears in the generated document. Note that the **Statement** tag is not displayed, and can not be changed from the dialog box. If the default text is left, the following will produce the following text in the generated document:

Hello World! My name is ComponentOne and I'm currently located in Pittsburgh, PA.

## Specifying When the Tags Dialog Box is Shown

When the ShowTagsInputDialog property is set to **True,** the **Tags** dialog box is shown just before the document is generated. You can programmatically show that dialog whenever you want (and independently of the value of the ShowTagsInputDialog property) by calling the EditTags method.

For example, the following code shows the tags input dialog box when a button is clicked:

- Visual Basic

```
Public Class Form1
    Dim doc As New C1PrintDocument()
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
```

```
        Me.C1PrintPreviewControl1.Document = doc
        ' Create tags to be shown.
        doc.Tags.Add(New C1.C1Preview.Tag("Statement", "Hello World!"))
        doc.Tags("Statement").ShowInDialog = True
        doc.Tags.Add(New C1.C1Preview.Tag("Name", "ComponentOne"))
        doc.Tags.Add(New C1.C1Preview.Tag("Location", "Pittsburgh, PA"))
        ' Add tags to the document.
        Dim rt As New C1.C1Preview.RenderText()
        rt.Text = "[Statement] My name is [Name] and my current location
is [Location]."
        doc.Body.Children.Add(rt)
    End Sub
    Private Sub EditTagsNow_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles EditTagsNow.Click
        ' Show the Tags dialog box on button click.
        doc.ShowTagsInputDialog = True
        doc.EditTags()
    End Sub
    Private Sub GenerateDocNow_Click(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles GenerateDocNow.Click
        doc.ShowTagsInputDialog = False
        ' Generate the document on button click.
        doc.Generate()
    End Sub
End Class
```

- C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    C1PrintDocument doc = new C1PrintDocument();
    private void Form1_Load(object sender, EventArgs e)
    {
        this.c1PrintPreviewControl1.Document = doc;
        // Create tags to be shown.
        doc.Tags.Add(new C1.C1Preview.Tag("Statement", "Hello World!"));
        doc.Tags["Statement"].ShowInDialog = true;
        doc.Tags.Add(new C1.C1Preview.Tag("Name", "ComponentOne"));
        doc.Tags.Add(new C1.C1Preview.Tag("Location", "Pittsburgh, PA"));
        // Add tags to the document.
        C1.C1Preview.RenderText rt = new C1.C1Preview.RenderText();
        rt.Text = "[Statement] My name is [Name] and my current location
is [Location].";
        doc.Body.Children.Add(rt);
    }
    private void EditTagsNow_Click(object sender, EventArgs e)
    {
        // Show the Tags dialog box on button click.
        doc.ShowTagsInputDialog = true;
        doc.EditTags();
    }
    private void GenerateDoc_Click(object sender, EventArgs e)
    {
        doc.ShowTagsInputDialog = false;
        // Generate the document on button click.
        doc.Generate();
    }
}
```

In the example above, the **Tags** dialog box will appear when the **EditTagsNow** button is clicked.

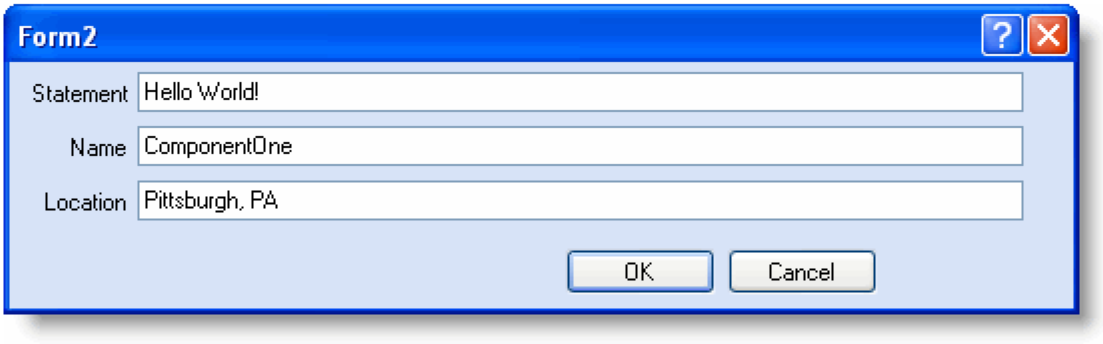
## Define the Default Tags Dialog Box

You can easily customize the **Tags** dialog box by adding an inherited form to your project that is based on either TagsInputForm or TagsInputFormBase. The difference in what approach you follow depends on whether you plan to make a small change to the form or if you want to completely redesign the form.

**Making a small change**

If you only want to make a small change to the default form (for example, add a Help button), you can add an inherited form to your project based on TagsInputForm, adjust it as needed, and assign that form's type name to the TagsInputDialogClassName property on the document.

For example, in the following form a **Help** button was added to the caption bar and the background color of the form was changed:



Statement Hello World!

Name ComponentOne

Location Pittsburgh, PA

**Completely changing the form**

If you choose, you can completely change the default form. For example, you can provide your own controls or entering tags' values, and so on. To do so, base your inherited form on TagsInputFormBase, change it as needed, and override the EditTags method.

### *Scripting/Expression Language*

The language used in expressions is determined by the value of the property **C1PrintDocument.ScriptingOptions.Language**. That property can have one of two possible values:

- **VB**. This is the default value, and indicates that the standard VB.NET will be used as the scripting language.
- **C1Report**. This value indicates that the **C1Report** scripting language will be used. While that language is similar to VB, there are subtle differences. This option is primarily provided for backwards compatibility.
- **CSharp**. This value  indicates that the standard C# will be used as the scripting language.

If **VB** is used as the expression language, when the document is generated a separate assembly is built internally for each expression containing a single class derived from ScriptExpressionBase. This class contains the protected properties that can be used in the expression. The expression itself is implemented as a function of that class.

For example:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.ScriptingOptions.Language = ScriptLanguageEnum.CSharp
Dim rt As New RenderText("[PageNo == 1 ? ""First"" : ""Not first""]")
doc.Body.Children.Add(rt)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.ScriptingOptions.Language = ScriptLanguageEnum.CSharp;
RenderText rt = new RenderText("[PageNo == 1 ? \"First\" : \"Not
first\"]");
doc.Body.Children.Add(rt);
```

### Assemblies and Namespaces

By default, the following assemblies are available (referenced) for scripts:

- System
- System.Drawing

To add another (system or custom) assembly to the list of assemblies referenced in scripts, add it to the **C1PrintDocument.ScriptingOptions.ExternalAssemblies** collection on the document. For instance, the following will add a reference to the System.Data assembly to the document's scripts:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.ScriptingOptions.ExternalAssemblies.Add("System.Data.dll")
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.ScriptingOptions.ExternalAssemblies.Add("System.Data.dll");
```

The following namespaces are by default available (imported) for use in scripts:

- System
- System.Collections
- System.Collections.Generic
- System.Text
- Microsoft.VisualBasic
- System.Drawing

To add another namespace, add it to the **C1PrintDocument.ScriptingOptions.Namespaces** collection on the document. For instance, this will allow the use of types declared in System.Data namespace in the document's scripts without fully qualifying them with the namespace:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.ScriptingOptions. Namespaces.Add("System.Data")
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.ScriptingOptions. Namespaces.Add("System.Data");
```

### IDs Accessible in Text Expressions

As mentioned above, expressions in brackets can be used within the **Text** property of **RenderText** and **ParagraphText** objects. In those expressions, the following object IDs are available:

- Document (type C1PrintDocument)

This variable references the document being generated. This can be used in a number of ways, for instance the following code will print the author of the current document:

  - Visual Basic

    ```
    Dim doc As New C1PrintDocument()
    Dim rt As New RenderText("Author:[Document.DocumentInfo.Author]")
    doc.Body.Children.Add(rt)
    ```

  - C#

    ```
    C1PrintDocument doc = new C1PrintDocument();
    RenderText rt =
      new RenderText("Author: [Document.DocumentInfo.Author]");
    doc.Body.Children.Add(rt);
    ```

- RenderObject (type RenderObject)

  This variable references the current render object.  For instance, the following code will print the name of the current render object:

  - Visual Basic

    ```
    Dim doc As New C1PrintDocument()
    Dim rt As New RenderText( _
      "The object's name is [RenderObject.Name]")
    rt.Name = "MyRenderText"
    doc.Body.Children.Add(rt)
    ```

  - C#

    ```
    C1PrintDocument doc = new C1PrintDocument();
    RenderText rt = new RenderText(
      "The object's name is [RenderObject.Name]");
    rt.Name = "MyRenderText";
    doc.Body.Children.Add(rt);
    ```

- Page (type C1Page)

  This variable references the current page (object of type **C1Page**). While the most commonly used in scripts members of the page object are accessible directly (see **PageNo**, **PageCount** and so on below), there is other data that can be accessed via the Page variable, such as the current page settings. For instance, the following code will print "Landscape is TRUE" if the current page layout has landscape orientation, and "Landscape is FALSE" otherwise:

  - Visual Basic

    ```
    Dim doc As New C1PrintDocument()
    Dim rt As New RenderText("Landscape is " + _
      "[Iif(Page.PageSettings.Landscape,\"TRUE\",\"FALSE\")].")
    doc.Body.Children.Add(rt)
    ```

  - C#

    ```
    C1PrintDocument doc = new C1PrintDocument();
    RenderText rt = new RenderText("Landscape is " +
      "[Iif(Page.PageSettings.Landscape,\"TRUE\",\"FALSE\")].");
    doc.Body.Children.Add(rt);
    ```

- PageNo (type Integer)

  This name resolves to the current 1-based page number. Equivalent to **Page.PageNo**.

- PageCount (type Integer)

  This name resolves to the total page count for the document. Equivalent to **Page.PageCount**. For instance, the following code can be used to generate the common "Page X of Y" page header:

  - Visual Basic

```
Dim doc As New C1PrintDocument()
doc.PageLayout.PageHeader = New RenderText( _
  "Page [PageNo] of [PageCount]")
```

  - C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.PageLayout.PageHeader = new RenderText(
  "Page [PageNo] of [PageCount]");
```

- PageX (type Integer)

  This name resolves to the current 1-based horizontal page number. (For documents without horizontal page breaks, will return 1.)

- PageY (type Integer)

  This name resolves to the current 1-based vertical page number. (For documents without horizontal page breaks, will be equivalent to **PageNo**.)

- PageXCount (type Integer)

  This name resolves to the total page count for the document. (For documents without horizontal page breaks, will return 1.)

- PageYCount (type Integer)

  This name resolves to the total page count for the document. (For documents without horizontal page breaks, will be equivalent to **PageCount**.)

  It is important to note that any of page numbering-related variables described here can be used anywhere in a document – not necessarily in page headers or footers.

- Fields (type FieldCollection)

  This variable references the collection of available database fields, and has the type **C1.C1Preview.DataBinding.FieldCollection**. It can only be used in data-bound documents. For instance, the following code will print the list of product names contained in the Products table of the NWIND database:

  - Visual Basic

```
Dim doc As New C1PrintDocument()
Dim dSrc As New DataSource()
dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB
dSrc.ConnectionProperties.ConnectString = _
  "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB"
Dim dSet1 As New C1.C1Preview.DataBinding.DataSet( _
  dSrc, "select * from Products")
doc.DataSchema.DataSources.Add(dSrc)
doc.DataSchema.DataSets.Add(dSet1)
Dim rt As New RenderText()
rt.DataBinding.DataSource = dSet1
rt.Text = "[Fields!ProductName.Value]"
doc.Body.Children.Add(rt)
```

  - C#

```
C1PrintDocument doc = new C1PrintDocument();
DataSource dSrc = new DataSource();
dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB;
dSrc.ConnectionProperties.ConnectString =
  @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB";
C1.C1Preview.DataBinding.DataSet dSet1 =
  new C1.C1Preview.DataBinding.DataSet(dSrc,
  "select * from Products");
```

```
doc.DataSchema.DataSources.Add(dSrc);
doc.DataSchema.DataSets.Add(dSet1);
RenderText rt = new RenderText();
doc.Body.Children.Add(rt);
rt.DataBinding.DataSource = dSet1;
rt.Text = "[Fields!ProductName.Value]";
```

Note the use of "!" to access an element of the fields array in the last line. Alternatively, you can write:

- Visual Basic

```
rt.Text = "[Fields(\"ProductName\").Value]"
```

- C#

```
rt.Text = "[Fields(\"ProductName\").Value]";
```

but the notation using "!" is shorter and easier to read.

- Aggregates (type AggregateCollection)

  This variable allows access to the collection of aggregates defined on the document. That collection is of the type **C1.C1Preview.DataBinding.AggregateCollection**, its elements have the type **C1.C1Preview.DataBinding.Aggregate**. For instance, the following code will print the average unit price after the list of products:

- Visual Basic

```
Dim doc As New C1PrintDocument()
Dim dSrc As New DataSource()
dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB
dSrc.ConnectionProperties.ConnectString = _
  "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB"
C1.C1Preview.DataBinding.DataSet dSet1 = _
  new C1.C1Preview.DataBinding.DataSet(dSrc, _
  "select * from Products")
doc.DataSchema.DataSources.Add(dSrc)
doc.DataSchema.DataSets.Add(dSet1)
Dim rt As New RenderText()
doc.Body.Children.Add(rt)
rt.DataBinding.DataSource = dSet1
rt.Text = "[Fields!ProductName.Value]"
doc.DataSchema.Aggregates.Add(new Aggregate( _
  "AveragePrice", "Fields!UnitPrice.Value", _
  rt.DataBinding, RunningEnum.Document, _
  AggregateFuncEnum.Average))
doc.Body.Children.Add(new RenderText( _
  "Average price: [Aggregates!AveragePrice.Value]"))
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
DataSource dSrc = new DataSource();
dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB;
dSrc.ConnectionProperties.ConnectString =
  @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB";
C1.C1Preview.DataBinding.DataSet dSet1 =
  new C1.C1Preview.DataBinding.DataSet(dSrc,  "select * from
Products");
doc.DataSchema.DataSources.Add(dSrc);
doc.DataSchema.DataSets.Add(dSet1);
RenderText rt = new RenderText();
doc.Body.Children.Add(rt);
rt.DataBinding.DataSource = dSet1;
rt.Text = "[Fields!ProductName.Value]";
doc.DataSchema.Aggregates.Add(new Aggregate(
  "AveragePrice", "Fields!UnitPrice.Value",
  rt.DataBinding, RunningEnum.Document,
  AggregateFuncEnum.Average));
doc.Body.Children.Add(new RenderText(
  "Average price: [Aggregates!AveragePrice.Value]"));
```

- DataBinding (type C1DataBinding)

  This variable allows accessing the **DataBinding** property of the current render object, of the type **C1.C1Preview.DataBinding.C1DataBinding**. For instance, the following code (modified from the sample showing the use of Fields variable) will produce a numbered list of products using the **RowNumber** member of the render object's **DataBinding** property in the text expression:

  - Visual Basic

    ```
    Dim doc As New C1PrintDocument()
    Dim dSrc As New DataSource()
    dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB
    dSrc.ConnectionProperties.ConnectString = _
      "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB"
    C1.C1Preview.DataBinding.DataSet dSet1 = _
      new C1.C1Preview.DataBinding.DataSet(dSrc, _
      "select * from Products")
    doc.DataSchema.DataSources.Add(dSrc)
    doc.DataSchema.DataSets.Add(dSet1)
    Dim rt As New RenderText()
    rt.DataBinding.DataSource = dSet1
    rt.Text = "[DataBinding.RowNumber]: [Fields!ProductName.Value]"
    doc.Body.Children.Add(rt)
    ```

  - C#

    ```
    C1PrintDocument doc = new C1PrintDocument();
    DataSource dSrc = new DataSource();
    dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB;
    dSrc.ConnectionProperties.ConnectString =
      @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB";
    C1.C1Preview.DataBinding.DataSet dSet1 =
      new C1.C1Preview.DataBinding.DataSet(dSrc,
      "select * from Products");
    doc.DataSchema.DataSources.Add(dSrc);
    doc.DataSchema.DataSets.Add(dSet1);
    RenderText rt = new RenderText();
    rt.DataBinding.DataSource = dSet1;
    rt.Text = "[DataBinding.RowNumber]: [Fields!ProductName.Value]";
    doc.Body.Children.Add(rt);
    ```

### IDs accessible in expressions within Filter, Grouping and Sorting

In filter, grouping and sorting expressions, the following subset of IDs are accessible:

- Document (type C1PrintDocument)
- DataBinding (type C1DataBinding)
- Fields (FieldCollection)

For details on those object types, see IDs Accessible in Text Expressions.

### IDs accessible in expressions used to specify calculated fields in a DataSet

In expressions used to specify calculated fields in data sets, the following subset of IDs are accessible:

- Document (type C1PrintDocument)
- Fields (FieldCollection)

For details on those object types, see IDs Accessible in Text Expressions.

## Data Binding

In addition to creating a C1PrintDocument fully (including data) in code, a C1PrintDocument may be data bound. In that case, the actual document is produced when the document is filled with data from the database during generation.

The main property facilitating data binding is the **DataBinding** property on RenderObject , of the type C1DataBinding, which allows to specify the data source for data shown by the render object. Additionally, the data binding can indicate that the render object must be repeated for all records in the data source, in which case the render object becomes similar to a "band" from a banded report generator. This is similar to the RDL definition from Microsoft.

Thus for data bound documents, document generation involves two stages:

- All data bound render objects are selected, and used (as templates) to create the "real" render objects based on data.
- The resulting document is paginated as a non-data bound document.

The document can contain the database schema (represented by the class C1DataSchema, and including the data base connection info, SQL queries, and so on) inside. The C1DataBinding objects within the document can reference properties of that schema. If all data bound objects in a document reference only the properties of the C1DataSchema of the document itself, the document becomes "data reflowable" – that is, the document can be regenerated independently of the program used to create it, with data completely updated from the database.

Also, the C1DataBinding may reference existing data sources (DataTable and so on) which were created on the form or elsewhere in the program that created that C1PrintDocument. In this case, of course, the document can only be regenerated with updating the data only in the context of that program, and saving and later loading that document (as a C1D or C1DX file) will break any connection with the data.

### Data Binding in Render Objects

When a render object is created, the data binding for it is not created initially. It is created when the **DataBinding** property is referenced in user code. For example:

- Visual Basic

```
Dim rt As RenderText = New RenderText
' ...
If Not (rt.DataBinding Is Nothing) Then
    MessageBox.Show("Data binding defined.")
End If
```

- C#

```
RenderText rt = new RenderText();
// ...
if (rt.DataBinding != null)
{
    MessageBox.Show("Data binding defined.");
}
```

The condition in the previous code will **always** evaluate to **True**. Thus if you only want to check whether data binding exists on a particular render object, you should use the DataBindingDefined property instead:

- Visual Basic

```
Dim rt As RenderText = New RenderText
' ...
If rt.DataBindingDefined Then
    MessageBox.Show("Data binding defined.")
End If
```

- C#

```
RenderText rt = new RenderText();
// ...
if  (rt.DataBindingDefined)
{
    MessageBox.Show("Data binding defined.");
}
```

> **Note:** This is similar to the **Handle** and **IsHandleCreated** properties of the WinForms Control class.

During document generation the **RenderObjectsList** collection is formed. Three different situations are possible as a result:

- The **Copies** property on the render object is null. This means that the object is not data bound, and is processed as usual. The Fragments property of the object can be used to access the actual rendered fragments of the object on the generated pages.
- The **Copies** property on the render object is not null, but Copies.Count is 0. This means that the object is data bound, but the data set is empty (contains no records). In such situations the object is not show in the document at all, that is, no fragments are generated for the object. For an example see **Binding to the Empty List** in the **DataBinding** sample.
- The **Copies** property on the render object is not null and Copies.Count is greater than 0. This means that the object is data bound, and the data source is not empty (contains records). During the document generation, several copies of the render object will be created and placed in this collection. Those copies will then be processed and fragments will be generated for each copy as usual.

### *Data bound tables*

Using the **DataBinding** property in the TableVectorGroup, which is the base class for table row and column groups, a RenderTable can be data bound.

> **Sample Project Available**
>
> For examples of binding to a RenderTable, see the **DataBinding** sample located on the ComponentOne HelpCentral Sample page.

Note that not only groups of rows, but also groups of columns can data bound. That is, a table can grow not only down, but also sideways.

Grouping will work, but note that group hierarchy is based on the hierarchy of TableVectorGroup objects, as shown in the following code:

- Visual Basic

```
Dim rt As C1.C1Preview.RenderTable = New C1.C1Preview.RenderTable
rt.Style.GridLines.All = C1.C1Preview.LineDef.Default

' Table header:
Dim c As C1.C1Preview.TableCell = rt.Cells(0, 0)
c.SpanCols = 3
c.Text = "Header"
```

```
' Group header:
c = rt.Cells(1, 0)
c.Text = "GroupId = [Fields!GroupId.Value]"
c.SpanCols = 3
c.Style.TextAlignHorz = C1.C1Preview.AlignHorzEnum.Center
c.Style.TextAlignVert = C1.C1Preview.AlignVertEnum.Center

' Sub-group header:
c = rt.Cells(2, 0)
c.Text = "GroupId = [Fields!GroupId.Value] SubGroupId =
[Fields!SubGroupId.Value]"
c.SpanCols = 3
c.Style.TextAlignHorz = C1.C1Preview.AlignHorzEnum.Center
c.Style.TextAlignVert = C1.C1Preview.AlignVertEnum.Center

' Sub-group data:
rt.Cells(3, 0).Text = "GroupId=[Fields!GroupId.Value]"
rt.Cells(3, 1).Text = "SubGroupId=[Fields!SubGroupId.Value]"
rt.Cells(3, 2).Text = "IntValue=[Fields!IntValue.Value]"

' Create a group of data bound lines, grouped by the GroupId field:
Dim g As C1.C1Preview.TableVectorGroup = rt.RowGroups(1, 3)
g.CanSplit = True
g.DataBinding.DataSource = MyData.Generate(20, 0)
g.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value")
g.Style.BackColor = Color.LightCyan

' Create a nested group, grouped by SubGroupId:
Dim ng As C1.C1Preview.TableVectorGroup = rt.RowGroups(2, 2)
ng.CanSplit = True
ng.DataBinding.DataSource = g.DataBinding.DataSource
ng.DataBinding.Grouping.Expressions.Add("Fields!SubGroupId.Value")
ng.Style.BackColor = Color.LightPink

' Create yet deeper nested data bound group:
Dim ng2 As C1.C1Preview.TableVectorGroup = rt.RowGroups(3, 1)
ng2.DataBinding.DataSource = g.DataBinding.DataSource
ng2.Style.BackColor = Color.LightSteelBlue
```

- C#

```
RenderTable rt = new RenderTable();
rt.Style.GridLines.All = LineDef.Default;

// Table header:
TableCell c = rt.Cells[0, 0];
c.SpanCols = 3;
c.Text = "Header";

// Group header:
c = rt.Cells[1, 0];
c.Text = "GroupId = [Fields!GroupId.Value]";
c.SpanCols = 3;
c.Style.TextAlignHorz = AlignHorzEnum.Center;
c.Style.TextAlignVert = AlignVertEnum.Center;

// Sub-group header:
c = rt.Cells[2, 0];
```

```
c.Text = "GroupId = [Fields!GroupId.Value] SubGroupId =
[Fields!SubGroupId.Value]";
c.SpanCols = 3;
c.Style.TextAlignHorz = AlignHorzEnum.Center;
c.Style.TextAlignVert = AlignVertEnum.Center;

// Sub-group data:
rt.Cells[3, 0].Text = "GroupId=[Fields!GroupId.Value]";
rt.Cells[3, 1].Text = "SubGroupId=[Fields!SubGroupId.Value]";
rt.Cells[3, 2].Text = "IntValue=[Fields!IntValue.Value]";

// Create a group of data bound lines, grouped by the GroupId field:
TableVectorGroup g = rt.RowGroups[1, 3];
g.CanSplit = true;
g.DataBinding.DataSource = MyData.Generate(20, 0);
g.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value");
g.Style.BackColor = Color.LightCyan;

// Create a nested group, grouped by SubGroupId:
TableVectorGroup ng = rt.RowGroups[2, 2];
ng.CanSplit = true;
ng.DataBinding.DataSource = g.DataBinding.DataSource;
ng.DataBinding.Grouping.Expressions.Add("Fields!SubGroupId.Value");
ng.Style.BackColor = Color.LightPink;

// Create yet deeper nested data bound group:
TableVectorGroup ng2 = rt.RowGroups[3, 1];
ng2.DataBinding.DataSource = g.DataBinding.DataSource;
ng2.Style.BackColor = Color.LightSteelBlue;
```

The above code can be illustrated by the following table:

<table>
<tr><td></td><td></td><td></td><td colspan="3">Header</td></tr>
<tr><td rowspan="3">Group 1, 3</td><td></td><td></td><td colspan="3">GroupId = [Fields!GroupId.Value]</td></tr>
<tr><td rowspan="2">Group 2, 2</td><td></td><td colspan="3">GroupId = [Fields!GroupId.Value] SubGroupId = [Fields!SubGroupId.Value]</td></tr>
<tr><td>Group 3, 1</td><td>GroupId=[Fields!GroupId.Value]</td><td>SubGroupId=[Fields!SubGroupId.Value]</td><td>IntValue=[Fields!IntValue.Value]</td></tr>
</table>

### *Data Binding Examples*

The **DataBinding** sample, available on HelpCentral, contains several examples of data bound documents. Some of the issues from that sample are discussed in the following topics.

## Working With Groups

A typical use of grouping is demonstrated by the following code:

- Visual Basic

```
' A RenderArea is created that is to be repeated for each group.
Dim ra As C1.C1Preview.RenderArea = New C1.C1Preview.RenderArea
```

```
ra.Style.Borders.All = New C1.C1Preview.LineDef("2mm", Color.Blue)

' MyData array of objects is used as the data source:
ra.DataBinding.DataSource = MyData.Generate(100, 0)

' Data is grouped by the GroupId field:
ra.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value")

' Create a RenderText that will serve as the group header; In a general
case, the header can be complex and itself be data bound:
Dim rt As C1.C1Preview.RenderText = New C1.C1Preview.RenderText

' The group header will look like "GroupId = XXX", where XXX is the value
of the GroupId field in the group:
rt.Text = "GroupId: [Fields!GroupId.Value]"
rt.Style.BackColor = Color.Yellow

' Add the header to the group area:
ra.Children.Add(rt)

' This RenderText will print records within each group:
rt = New C1.C1Preview.RenderText

' The text to print for each record:
rt.Text = "GroupId: [Fields!GroupId.Value]" &
Microsoft.VisualBasic.Chr(13) & "IntValue: [Fields!IntValue.Value]"
rt.Style.Borders.Bottom = C1.C1Preview.LineDef.Default
rt.Style.BackColor = Color.FromArgb(200, 210, 220)

' Set the text's data source to the data source of the containing
RenderArea - this indicates that the render object is bound to the current
group in the specified object:
rt.DataBinding.DataSource = ra.DataBinding.DataSource

' Add the text to the area:
ra.Children.Add(rt)
```

- C#

```
// A RenderArea is created that is to be repeated for each group.
RenderArea ra = new RenderArea();
ra.Style.Borders.All = new LineDef("2mm", Color.Blue);

// MyData array of objects is used as the data source:
ra.DataBinding.DataSource = MyData.Generate(100, 0);

// Data is grouped by the GroupId field:
ra.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value");

// Create a RenderText that will serve as the group header; In a general
case, the header can be complex and itself be data bound:
RenderText rt = new RenderText();

// The group header will look like "GroupId = XXX", where XXX is the value
of the GroupId field in the group:
rt.Text = "GroupId: [Fields!GroupId.Value]";
rt.Style.BackColor = Color.Yellow;
```

```
// Add the header to the group area:
ra.Children.Add(rt);

// This RenderText will print records within each group:
rt = new RenderText();

// The text to print for each record:
rt.Text = "GroupId: [Fields!GroupId.Value]\rIntValue:
[Fields!IntValue.Value]";
rt.Style.Borders.Bottom = LineDef.Default;
rt.Style.BackColor = Color.FromArgb(200, 210, 220);

// Set the text's data source to the data source of the containing
RenderArea - this indicates that the render object is bound to the current
group in the specified object:
rt.DataBinding.DataSource = ra.DataBinding.DataSource;

// Add the text to the area:
ra.Children.Add(rt);
```

## Using Aggregates

The code below expands on the previous example by introducing aggregates in groups and the document as a whole:

- Visual Basic

```
' Create a Render area to be repeated for each group:
Dim ra As C1.C1Preview.RenderArea = New C1.C1Preview.RenderArea
ra.Style.Borders.All = New C1.C1Preview.LineDef("2mm", Color.Blue)
ra.DataBinding.DataSource = MyData.Generate(20, 0, True)
ra.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value")

' Make an aggregate that will calc the sum of IntValue fields within each
group:
Dim agg As C1.C1Preview.DataBinding.Aggregate = New
C1.C1Preview.DataBinding.Aggregate("Group_IntValue")

' Define the expression that will calc the sum:
agg.ExpressionText = "Fields!IntValue.Value"

' Specify that aggregate should have group scope:
agg.Running = C1.C1Preview.DataBinding.RunningEnum.Group

' Specify the data source for the aggregate:
agg.DataBinding = ra.DataBinding

' Add the aggregate to the document:
doc.DataSchema.Aggregates.Add(agg)

' Make an aggregate that will calc the sum of IntValue fields over the
whole document:
agg = New C1.C1Preview.DataBinding.Aggregate("Total_IntValue")

' Define the expression to calc the sum:
agg.ExpressionText = "Fields!IntValue.Value"

' Specify that aggregate should have document scope:
```

```
agg.Running = C1.C1Preview.DataBinding.RunningEnum.All

' Specify the data source for the aggregate:
agg.DataBinding = ra.DataBinding

' Add the aggregate to the document:
doc.DataSchema.Aggregates.Add(agg)

' Make the group header:
Dim rt As C1.C1Preview.RenderText = New C1.C1Preview.RenderText
rt.Text = "GroupId: [Fields!GroupId.Value]"
rt.Style.BackColor = Color.Yellow
ra.Children.Add(rt)

' This render text will print group records; as can be seen, group
aggregate values can be referenced not only in group footer but also in
group header and in group detail:
rt = New C1.C1Preview.RenderText
rt.Text = "GroupId:
[Fields!GroupId.Value]\rIntValue:[Fields!IntValue.Value]\rGroup_IntValue:[
Aggregates!Group_IntValue.Value]\rTotal_IntValue:[Aggregates!Total_IntValu
e.Value]\rTatalNested_IntValue:[Aggregates!TatalNested_IntValue.Value]"
rt.Style.Borders.Bottom = C1.C1Preview.LineDef.Default
rt.Style.BackColor = Color.FromArgb(200, 210, 220)
rt.DataBinding.DataSource = ra.DataBinding.DataSource
ra.Children.Add(rt)

' This aggregate is also calculated over the group, but is connected to
the data binding of the nested object:
agg = New C1.C1Preview.DataBinding.Aggregate("TotalNested_IntValue")
agg.ExpressionText = "Fields!IntValue.Value"
agg.Running = RunningEnum.All
agg.DataBinding = rt.DataBinding
doc.DataSchema.Aggregates.Add(agg)

' Add the area to the document:
doc.Body.Children.Add(ra)
```

- C#

```
// Create a Render area to be repeated for each group:
RenderArea ra = new RenderArea();
ra.Style.Borders.All = new LineDef("2mm", Color.Blue);
ra.DataBinding.DataSource = MyData.Generate(20, 0, true);
ra.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value");

// Make an aggregate that will calc the sum of IntValue fields within each
group:
Aggregate agg = new Aggregate("Group_IntValue");

// Define the expression that will calc the sum:
agg.ExpressionText = "Fields!IntValue.Value";

// Specify that aggregate should have group scope:
agg.Running = RunningEnum.Group;

// Specify the data source for the aggregate:
agg.DataBinding = ra.DataBinding;
```

```
// Add the aggregate to the document:
doc.DataSchema.Aggregates.Add(agg);

// Make an aggregate that will calc the sum of IntValue fields over the
whole document:
agg = new Aggregate("Total_IntValue");

// Define the expression to calc the sum:
agg.ExpressionText = "Fields!IntValue.Value";

// Specify that aggregate should have document scope:
agg.Running = RunningEnum.All;

// Specify the data source for the aggregate:
agg.DataBinding = ra.DataBinding;

// Add the aggregate to the document:
doc.DataSchema.Aggregates.Add(agg);

// Make the group header:
RenderText rt = new RenderText();
rt.Text = "GroupId: [Fields!GroupId.Value]";
rt.Style.BackColor = Color.Yellow;
ra.Children.Add(rt);

// This render text will print group records; as can be seen, group
aggregate values can be referenced not only in group footer but also in
group header and in group detail:
rt = new RenderText();
rt.Text = "GroupId:
[Fields!GroupId.Value]\rIntValue:[Fields!IntValue.Value]\rGroup_IntValue:[
Aggregates!Group_IntValue.Value]\rTotal_IntValue:[Aggregates!Total_IntValu
e.Value]\rTatalNested_IntValue:[Aggregates!TatalNested_IntValue.Value]";
rt.Style.Borders.Bottom = LineDef.Default;
rt.Style.BackColor = Color.FromArgb(200, 210, 220);
rt.DataBinding.DataSource = ra.DataBinding.DataSource;
ra.Children.Add(rt);

// This aggregate is also calculated over the group, but is connected to
the data binding of the nested object:
agg = new Aggregate("TotalNested_IntValue");
agg.ExpressionText = "Fields!IntValue.Value";
agg.Running = RunningEnum.All;
agg.DataBinding = rt.DataBinding;
doc.DataSchema.Aggregates.Add(agg);

// Add the area to the document:
doc.Body.Children.Add(ra);
```

Note that there are also aggregate types that can be used in data-bound C1PrintDocuments without declaring them in the document's aggregates collection (Aggregates). For more details and an example, see the Data Aggregates topic.

### Data Aggregates

In the 2010 v1 release, new aggregates were added to **Reports for WinForms**. These aggregate types can be used in data-bound C1PrintDocuments without the need to declare them in the document's aggregates collection (Aggregates).

For instance, if "Balance" is a data field in a data-bound document, the following RenderText can be used to print the total balance for the dataset:

- Visual Basic
  ```
  Dim rt As New RenderText("[Sum(""Fields!Balance.Value"")]")
  ```
- C#
  ```
  RenderText rt = new RenderText("[Sum(\"Fields!Balance.Value\")]");
  ```

The following new properties and methods were added to the DataSet and C1DataBinding types to support this feature:

| Class | Member | Description |
|---|---|---|
| C1DataBinding | Name property | Gets or sets the name of the current C1DataBinding. That name can be used in aggregate functions to indicate which data binding the aggregate refers to. |
| DataSet | Name property | Gets or sets the name of the current DataSet. That name can be used in aggregate functions to indicate which data set the aggregate refers to. |

All aggregate functions have the follwing format:

```
AggFunc(expression, scope)
```

where:

- expression is a string defining an expression calculated for each row group or dataset row.
- scope is a string identifying the set of data for which the aggregate is calculated. If omitted, the aggregate is calculated for the current set of data (such as for the current group, dataset, and so on). If specified, should be the name of the target group or dataset.

For example, if a dataset has the following fields, *ID, GroupID, SubGroupID, NAME, Q*, and records are grouped by *GroupID* and *SubGroupID*, the following document can be created:

- Visual Basic
  ```
  Dim doc As New C1PrintDocument()
  Dim raGroupId As New RenderArea()
  ' ...set up raGroupId properties as desired...
  raGroupID.DataBinding.DataSource = dataSet
  raGroupID.DataBinding.Name = "GroupID"
  raGroupID.DataBinding.Grouping.Expressions.Add("Fields!GroupID.Value")

  Dim raSubGroupID As New RenderArea()
  ' ...set up raSubGroupID properties as desired...
  raSubGroupID.DataBinding.DataSource = dataSet
  raSubGroupID.DataBinding.Grouping.Expressions.Add("Fields!SubGroupID.Value
  ")
  raGroupID.Children.Add(raSubGroupID)

  Dim raDetail As New RenderArea()
  ' ...set up raDetail properties as desired...
  ```

```
raDetail.DataBinding.DataSource = dataSet
raSubGroupID.Children.Add(raDetail)

' show value of Q field:
Dim rtQ As New RenderText()
rtQ.Text = "[Fields!Q.Value]"
raDetail.Children.Add(rtQ)

' show sum of Q field for nested group (SubGroupID):
Dim rtSumQ1 As New RenderText()
rtSumQ1.Text = "[Sum(""Fields!Q.Value"")]"
raDetail.Children.Add(rtSumQ1)
' show sum of Q field for GroupID:
Dim rtSumQ2 As New RenderText()
rtSumQ2.Text = "[Sum(\"Fields!Q.Value\", "\"GroupID\"")]"
raDetail.Children.Add(rtSumQ2)
' show TOTAL sum of Q field for the entire dataset:
Dim rtSumQ3 As New RenderText()
rtSumQ3.Text = "[Sum(\"Fields!Q.Value\", "\"DataSet\"")]"
raDetail.Children.Add(rtSumQ3)
doc.Body.Children.Add(raGroupId)
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderArea raGroupId = new RenderArea();
// ...set up raGroupId properties as desired...
raGroupID.DataBinding.DataSource = dataSet;
raGroupID.DataBinding.Name = "GroupID";
raGroupID.DataBinding.Grouping.Expressions.Add("Fields!GroupID.Value");

RenderArea raSubGroupID = new RenderArea();
// ...set up raSubGroupID properties as desired...
raSubGroupID.DataBinding.DataSource = dataSet;
raSubGroupID.DataBinding.Grouping.Expressions.Add("Fields!SubGroupID.Value
");
raGroupID.Children.Add(raSubGroupID);

RenderArea raDetail = new RenderArea();
// ...set up raDetail properties as desired...
raDetail.DataBinding.DataSource = dataSet;
raSubGroupID.Children.Add(raDetail);

// show value of Q field:
RenderText rtQ = new RenderText();
rtQ.Text = "[Fields!Q.Value]";
raDetail.Children.Add(rtQ);
// show sum of Q field for nested group (SubGroupID):
RenderText rtSumQ1 = new RenderText();
rtSumQ1.Text = "[Sum(\"Fields!Q.Value\")]";
raDetail.Children.Add(rtSumQ1);
// show sum of Q field for GroupID:
RenderText rtSumQ2 = new RenderText();
rtSumQ2.Text = "[Sum(\"Fields!Q.Value\", "\"GroupID\"")]";
raDetail.Children.Add(rtSumQ2);
// show TOTAL sum of Q field for the entire dataset:
RenderText rtSumQ3 = new RenderText();
rtSumQ3.Text = "[Sum(\"Fields!Q.Value\", "\"DataSet\"")]";
```

```
raDetail.Children.Add(rtSumQ3);

doc.Body.Children.Add(raGroupId);
```

When the above document is generated, each instance of the *raDetail* group will show four values as follows:

- the current value of "Q" field
- the sum of the "Q" field over the current SubGroupID
- the sum of the "Q" field over the current GroupID
- the sum of the "Q" field over the whole document

## Table of Contents

C1PrintDocument supports automatic generation of table of contents (TOC). The table of contents itself is represented by a dedicated render object, RenderToc, which is derived from RenderArea and adds TOC-specific features. Individual items within the TOC are represented by RenderTocItem (derived from RenderParagraph). Each TOC item holds a hyperlink (the **RenderTocItem.Hyperlink** property) pointing to a location in the document (represented by an anchor). So, the same mechanism is used for connecting TOC items to document content as for hyperlinks. Convenient methods are provided to create the TOC, see below.

To add a table of contents to your document, do the following:

1. Create an instance of the RenderToc class and add it to your document at the point where you want the TOC to appear.
2. Add individual items (of the type RenderTocItem) to the RenderToc instance. Any of the following approaches (or a mix of them) may be used for this:
    - You may add the TOC item to the TOC using any of the overloads of the **RenderToc.AddItem** method, providing the text of the item, the location it should point to, and optionally the level in the TOC.
    - You may create an instance of the RenderTocItem class in your code, set properties on it, and add it to the Children collection of the TOC render object.
    - An overload of the RenderTocItem constructor is provided which accepts an instance of the RenderToc as the argument. When that constructor is used, the newly created TOC item is added to the TOC within the constructor, so you do not need to add it to the **Children** collection of the TOC object manually.

Sample Available

For a complete sample of how to create the table of contents for a document using the dedicated RenderToc render object, see the **RenderTOC** sample located on the ComponentOne HelpCentral Sample page.

## Word Index

You can now automatically generate indexes using C1PrintDocument. Each index (there can be several indexes in a document) consists of an alphabetized sorted list of letter headings with entries followed by lists of page numbers where the entry occurs.

An index is represented by an instance of the RenderIndex class, a render object derived from RenderArea. You can add that object to the document as you would any other render object, but with one important limitation: the index must appear in the document after all occurrences of entries (terms) contained in it; so, like the traditional index, the index here would be best served to appear at the end of the document.

Terms are words and word combinations that occur in the document and should appear as entries in the index. When the document is created, these terms are added to the RenderIndex object together with information about locations where they occur (usually the RenderText or RenderParagraph containing the term). Then, when the document generates, the RenderIndex object produces the actual index.

## Classes Supporting the Index Feature

The following specialized classes support indexing:

- RenderIndex: This class is derived from RenderArea, and produces the index when inserted in a C1PrintDocument and that document generates.
  - The RenderIndex must appear in the document **after** all occurrences of the index entries. The reason for this limitation is that the actual content of the index (and hence, the amount of space occupied by it) may vary significantly depending on the occurrences of the entries.
- IndexEntry: This class is used to describe an index entry (term) in the index.
  - Each entry can have multiple occurrences (locations in the document where the term is described or referenced) associated with it. The collection of all occurrences of an entry is exposed via the Occurrences property on the IndexEntry.
  - Each occurrence will produce a hyperlinked page number in the index when the document is generated. Besides which, an entry may contain a list of sub-entries (exposed by the property Children). The nesting level is unlimited, though usually up to 3 levels are used.
  - Finally, to allow linking an entry to other entries in the index, the SeeAlso property on the entry contains a list of index entries that will be listed as references for the current entry in the generated index.
- IndexEntryOccurrence: This class describes a single occurrence of an entry in the document.
  - Elements of this type are contained in the Occurrences collection of an IndexEntry.
  - One or more occurrences can be specified (as parameters to the constructor) when an instance of an index entry is created, and more occurrences can be added to the entry later.
  - The main functional property of this class is Target, of the type C1LinkTarget, which points to the location of the occurrence.

## Generating an Index In Code

Typically, the following steps would be involved in providing a simple one-level index in a document that is created in code:

1. An instance of the RenderIndex class should be created and stored in a local variable (as noted above, the index may not precede the occurrences of its entries).
2. As content (render objects) is added to the document, some program logic should identify strings that are to become entries (terms) in the index. Each such string should be tested on whether it has already been added to the Entries collection of the index object created in step 1. If this is a new entry, a new IndexEntry object should be created for it, and added to the index.
3. An entry occurrence (IndexEntryOccurrence) should be added to the existing or newly created entry, to point to the location of the occurrence in the document. Usually the location would be identified by the RenderObject that contains it and is being added to the document.
4. When all occurrences of the entries have been added to the document, the RenderIndex object created in step 1 can be added to the document's body.
5. When the document is generated, the RenderIndex object produces a hyperlinked index of the entries that have been added to it. The entries are automatically sorted, split into groups corresponding to each letter, and letter headings added.

Of course, this is only one simple possible scenario designed to demonstrate the relationship between the main objects involved in creating an index. Other possibilities include the creation of indexed terms (index entries) prior to document creation (for example, based on an external dictionary of terms), adding nested entries (sub-entries), and so on.

### Customizing the Index's Appearance

The following properties are provided to customize the way the generated index looks:

- **Styles** (see also Styles):
  - Style: specifies the style for the whole index (including headings, entries, and so on).
  - HeadingStyle: specifies the style used for letter headings (the heading is a letter preceding the group of entries starting with that letter). In the generated index, each heading (usually just the capitalized letter preceding the group of entries beginning with it) is represented by a separate render object (RenderText) to which this style is applied.
  - EntryStyles: an indexed property specifying the styles of entries at different levels. For instance, `EntryStyles[0]` (`EntryStyles(0)` in VB) specifies the style of entries at the top level, `EntryStyles[1]` (`EntryStyles(1)` in VB) specifies the style of sub-entries, and so on. (If the number of nested levels in the index is greater than the number of elements in the EntryStyles collection, the last style in the collection is used for nested styles.)

    In the generated index, each entry (the term followed by the list of pages where it appears) is represented by a separate RenderParagraph object, to which the style determined by this property indexed by the entry's nesting level is applied. For instance, this style allows you to specify the minimum number of lines of an entry text before a page break can be inserted (via MinOrphanLines).
  - EntryStyle: this is a shortcut for the first (with index 0) element of the EntryStyles collection.
  - SeeAlsoStyle: allows you to specify the style of the "see also" text used to precede cross references between entries (see SeeAlso).
  - Style: allows you to override the style for a particular entry.
  - SeeAlsoStyle: allows you to override the style of the "see also" text for a particular entry.
- **Other properties:**
  - RunIn: a Boolean property (False by default) which, if True, indicates that sub-entries should appear in line with the main heading rather than indented on separate lines.
  - EntryIndent: a Unit property specifying the indent of sub-entries relative to the main entry. The default is 0.25 inch.
  - EntryHangingIndent: a Unit property specifying the hanging indent (to the left) of the first line of an entry's text relative to the following lines (used if the list of references does not fit on a single line). The default is -0.125 inch.
  - LetterSplitBehavior: a SplitBehaviorEnum property that determines how a letter group (entries starting with the same letter) can be split vertically. The default is SplitBehaviorEnum.**SplitIfNeeded**. Note that headings (represented by their letters by default) are always printed together with their first entry.
  - Italic: similar to Bold but uses italic face instead of bold.
  - LetterFormat: a string used to format the letter headings. The default is "{0}".
  - TermDelimiter: a string used to delimit the entry term and the list of term's occurrences (page numbers). The default is a comma followed by a space.
  - RunInDelimiter: a string used to delimit the entries when a run-in (see RunIn) index is generated. The default is a semicolon.

- OccurrenceDelimiter: a string used to delimit the list of occurrences of an entry (page numbers). The default is a comma followed by a space.
- PageRangeFormat: a format string used to format page ranges of entries' occurrences. The default is "{0}-{1}".
- SeeAlsoFormat: a string used to format the "see also" references. The default is " (see {0})" (a space, followed by an opening parentheses, followed by the format item used to output the reference, followed by a closing parentheses).
- FillChar: a character used as filler when the page numbers are aligned to the right (PageNumbersAtRight is True). The default value of this property is a dot.
- PageNumbersAtRight: a Boolean property indicating whether to right-align the page numbers. The default is False.
- EntrySplitBehavior: a SplitBehaviorEnum property that determines how a single entry can split vertically. The default is SplitBehaviorEnum.**SplitIfLarge**. This property applies to entries at all levels.
- Bold: a Boolean property that allows you to highlight the page number corresponding to a certain occurrence of an entry using bold face. (For example, this can be used to highlight the location where the main definition of a term is provided,.

## Index Styles Hierarchy

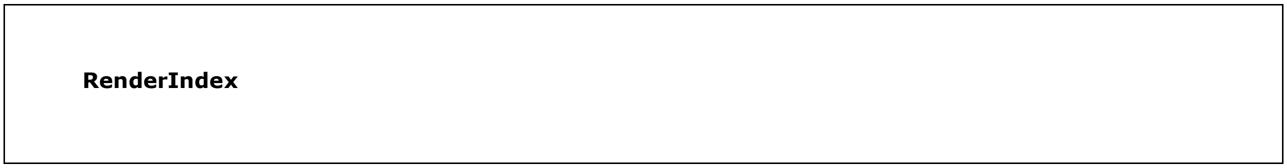
The hierarchy of index-specific styles is as follows:

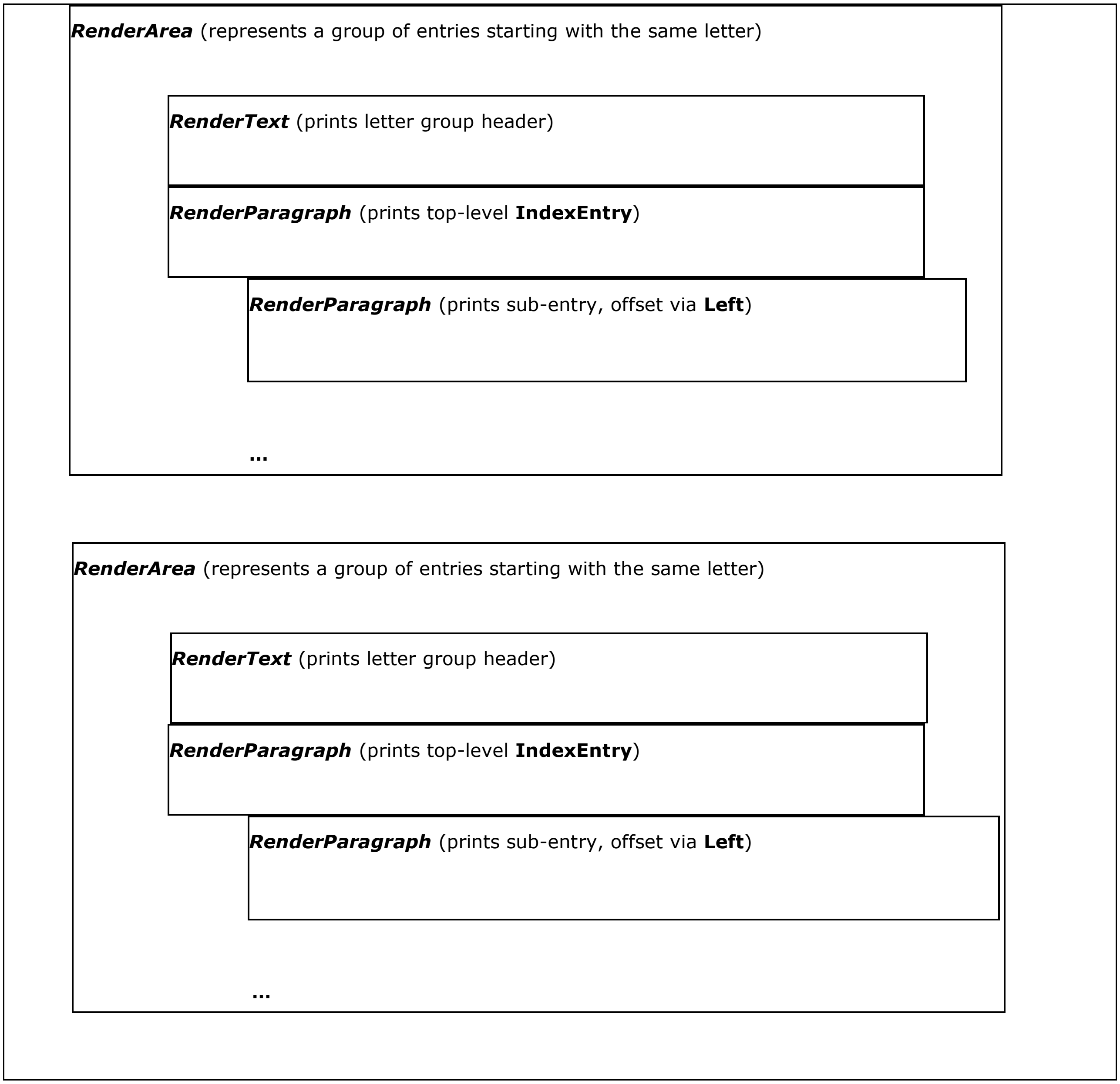
- Style, of the RenderIndex object, serves as **AmbientParent** for all other index-specific styles
    - HeadingStyle
    - EntryStyles
        - Style
    - SeeAlsoStyle
        - SeeAlsoStyle

With the exception of the Style of RenderIndex, all of the styles listed above serve as both Parent and AmbientParent for the inline styles of related objects. So for example, while setting a font on the Style of a RenderIndex will affect all elements of that index (unless overridden by a lower-level style), specifying a border on that style will draw that border around the whole index but not around individual elements in it. On the other hand, specifying a border on SeeAlsoStyle for instance will draw borders around each "see also" element in the index.

### *The Structure of the Generated Index*

The following figure shows the structure and hierarchy of the render object tree created by a RenderIndex object when the document generates (note that in the figure, only the RenderIndex object at the top level is created by user code; all other objects are created automatically):




***RenderArea*** (represents a group of entries starting with the same letter)
***RenderText*** (prints letter group header)
***RenderParagraph*** (prints top-level **IndexEntry**)
***RenderParagraph*** (prints sub-entry, offset via **Left**)
...
***RenderArea*** (represents a group of entries starting with the same letter)
***RenderText*** (prints letter group header)
***RenderParagraph*** (prints top-level **IndexEntry**)
***RenderParagraph*** (prints sub-entry, offset via **Left**)
...

## Outline

C1PrintDocument supports outlines. The document outline is a tree (specified by the Outlines property), with nodes (of the type OutlineNode) pointing to locations in the document. The outline is shown on a tab in the navigation panel of the preview, and allows navigating to locations corresponding to items by clicking on the items. Also, outlines are exported to formats supporting that notion (such as PDF).

To create an outline node, use any of the overloaded **Outline** constructors. You can specify the text of the outline, the location within the document (a render object or an anchor), and an icon to be shown in the outline tree panel in the preview. Top-level nodes should be added to the **Outlines** collection of the document. Each outline node may, in its turn, contain a collection of child nodes in the **Children** collection, and so on.

**Tip:** The outline nodes for each item that is being rendered on document can be clicked. Clicking a node will show the items attached with that node. For an example of how to add outline nodes, see Adding Outline Entries to the Outline Tab.

## Embedded Fonts

When a C1PrintDocument is saved as a C1DX or C1D file, fonts used in the document may be embedded in that file. In that case, when that document is loaded from the file on a different system, text drawn with fonts that have been embedded is guaranteed to render correctly even if the current system does not have all the original fonts installed. Font embedding may be particularly useful when rare or specialized fonts are used (such as, a font drawing barcodes). Note that when a font is embedded in a C1PrintDocument that does not mean that all glyphs from that font are embedded. Instead, just the glyphs actually used in the font are embedded.

The following C1PrintDocument properties are related to font embedding:

- EmbeddedFonts – this is the collection that contains fonts embedded in the document. Note that while it may be automatically populated, it may also be manually changed for custom control over font embedding (see below for more details).
- DocumentFonts – this collection is automatically populated, depending on the value of the C1PrintDocument.FontHandling property. It may be used to find out which fonts are used in the document.
- FontHandling – this property determines whether and how the two related collections (EmbeddedFonts and DocumentFonts) are populated.

### *Font Substitution*

When a text is rendered using a font, and a glyph appears in the text that is not present in the specified font, a substitute font may be selected to render that glyph. For instance, if the Arial font is used to render Japanese hieroglyphs, Arial Unicode MS font may be used to actually render the text. C1PrintDocument can analyze this and add the actual fonts used (rather than those specified) to the DocumentFonts and/or EmbeddedFonts collections. To do that, the FontHandling must be set to FontHandling.**BuildActualDocumentFonts** or FontHandling.**EmbedActualFonts**. The downside to those settings is that it takes time, making the document generate slower. Hence it may be recommended that those settings are used only if the document contains characters that may be missing from the fonts that are specified (for example, text in Far Eastern languages using common Latin fonts).

When font substitution is analyzed, the following predefined set of fonts is searched for the best matching font containing the missing glyphs:

- MS UI Gothic
- MS Mincho
- Arial Unicode MS
- Batang
- Gulim
- Microsoft YaHei
- Microsoft JhengHei
- MingLiU
- SimHei
- SimSun

### *Selective Font Embedding*

If your document uses a specialized font along with the commonly available fonts such as Arial, you may want to embed just that specialized font (or several fonts) without embedding all fonts used in the document. To do that, follow these steps:

1. Set FontHandling to a value other than FontHandling.**EmbedFonts** or FontHandling.**EmbedActualFonts**. This will leave the **EmbeddedFonts** collection empty when the document generates;
2. Add the specialized fonts (they must be installed on the current system) to the document's **EmbeddedFonts** collection manually in code. To create an **EmbeddedFont**, pass the.NET **Font** object corresponding to your custom font to the **EmbeddedFont**'s constructor. Add the required glyphs to the font using the **EmbeddedFont.AddGlyphs** method (several overloads are provided).

When a C1PrintDocument with the **EmbeddedFonts** collection created in this way is saved (as C1DX or C1D file), just the fonts specified in that collection are embedded in the document.

## Dictionary

If one item (for example, an image or an icon) is used in several places in a document, you can store that item once in a location available from the whole document and reference that instance, rather than inserting the same image or icon in every place it is used. For that, C1PrintDocument provides a dictionary.

To access the dictionary, use the Dictionary property. The dictionary contains items of the base type DictionaryItem, which is an abstract type. Two derived classes are provided, DictionaryImage and DictionaryIcon, to store images and icons correspondingly. Items in the dictionary are referenced by names. In order to use an item, you must assign a name to it. The name must be unique within the dictionary.

Dictionary items may be used in the following places in the document:

- As the background image of a style, using the property BackgroundImageName.
- As the image in a RenderImage object, using the property ImageName.

So, if in your document the same image is used in several places, do the following:

- Add the image to the dictionary, for instance like this:
  - Visual Basic
    ```
    Me.C1PrintDocument1.Dictionary.Add(New
    C1.C1Preview.DictionaryImage("image1", Image.FromFile("myImage.jpg")))
    ```
  - C#
    ```
    this.c1PrintDocument1.Dictionary.Add(new DictionaryImage("image1",
    Image.FromFile("myImage.jpg")));
    ```
- Use that image by setting either the BackgroundImageName on styles, or the ImageName property on render images, for example, like this:
  - Visual Basic
    ```
    Dim ri As New C1.C1Preview.RenderImage()
    ri.ImageName = "image1"
    ```
  - C#
    ```
    RenderImage ri = new RenderImage();
    ri.ImageName = "image1";
    ```

## C1Report Definitions

C1PrintDocument has the ability to import and generate C1Report definitions.

To import a C1Report into a C1PrintDocument, use the ImportC1Report method. For example, the following code can be used to import and preview the **Alphabetical List of Products** report included in the **ReportBrowser** sample shipped with **Reports for WinForms**:

- Visual Basic

```
Dim doc As C1PrintDocument = New C1PrintDocument()
doc.ImportC1Report("NWind.xml", "Alphabetical List of Products")
Dim pdlg As C1PrintPreviewDialog = New C1PrintPreviewDialog()
pdlg.Document = doc
pdlg.ShowDialog()
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.ImportC1Report("NWind.xml", "Alphabetical List of Products");
C1PrintPreviewDialog pdlg = new C1PrintPreviewDialog();
pdlg.Document = doc;
pdlg.ShowDialog();
```

After a C1Report has been imported into C1PrintDocument, the document has the following structure:

**For a report without grouping:**

- The page footer is represented by a RenderSection object and is assigned to the document's **PageLayouts.Default.PageFooter**.
- For each section of the report, a RenderSection object is created and added to the Body of the document in the following order:
    - Header (the report header, SectionTypeEnum**.Header**)
    - PageHeader (the page header, SectionTypeEnum**.PageHeader**)
    - Detail (the data section, SectionTypeEnum**.Detail**)
    - Footer (the report footer, SectionTypeEnum**.Footer**)

  A copy of the **PageHeader** is also assigned to the **PageHeader.LayoutChangeAfter.PageLayout.PageHeader**. This complex structure is needed because in **C1Report**, the first page header is printer after the report header.

**For a report with grouping:**

For each group, a RenderArea is created, and the following object tree is placed between the PageHeader and Footer (for 2 groups):

- RenderArea representing the top level group
    - RenderSection representing GroupHeader1
    - RenderArea representing the nested group
        - RenderSection representing GroupHeader2
        - Detail
        - RenderSection representing GroupFooter2
    - RenderSection representing GroupFooter1

## *C1Report Import Limitations*

Importing C1Report into C1PrintDocument has the following limitations:

- Only a report definition contained in an XML file can be imported. That is, if an application that produces a report in code via C# or VB.NET handlers attached to C1Report events, that report cannot be imported into C1PrintDocument.
- In C1Report, if there is no printer installed, and CustomWidth and CustomHeight are both set to 0, the paper size is always set to Letter (8.5in x 11in). In C1PrintDocument, the paper size is determined based on the current locale, for example, for many European countries it will be set to A4 (210cm x 297cm).
- Scripting limitations:

  C1PrintDocument:

  - The **Font** property is read-only.

  Field:

  - The Section property is read-only.
  - The Font property is read-only.
  - The LineSpacing property does not exist.
  - The Field.Subreport property is read-only.
  - If the LinkTarget property contains an expression, it will not be evaluated and will be used literally.
  - The SubreportHasData property does not exist.
  - The LinkValue property does not exist.

  Layout:

  - The ColumnLayout property is not supported, columns always go top to bottom and left to right.
  - The LabelSpacingX property does not exist.
  - The LabelSpacingY property does not exist.
  - The OverlayReplacements property does not exist.
- In **OnFormat** event handlers, properties that affect the pagination of the resulting document should not be changed. For example, ForcePageBreak cannot be used.
- The dialog box for entering the report's parameters is not shown. Instead, default values are used. If the default is not specified, it is determined by the type of the parameter, for example, 0 is used for numbers, empty string for strings, current date for dates, and so on.
- Database fields cannot be used in PageHeader and PageFooter.
- In C1Report, in multi-column reports the report header is printed across all columns; in C1PrintDocument, it will be printed only across the first column.
- Across is not supported for columns.

### *Working with Printer Drivers*

Several new members in **ComponentOne Reports for WinForms** were added to work around specific problems caused by printer drivers.

The following members were added to resolve issues with printer drivers:

| **Class** | **Member** | **Description** |
|---|---|---|
| C1PreviewPane | AdjustPrintPage event | Fired from within the **PrintPage** event handler of the C1PrintManager used to print |

| | | |
|---|---|---|
| | | the document. |
| C1PrintManager | AdjustPrintPage event | Fired from within the **PrintDocument.PrintPage** event handler of current print manager, prior to actually printing the page. |
| C1PrintOptions | DrawPrintableAreaBounds property | Gets or sets a value indicating whether a line is drawn around the printable area of the page (useful to debug printer issues). |
| | PrintableAreaBoundsPen property | Gets or sets the pen used to draw printable area bounds if DrawPrintableAreaBounds is **True**. |
| | PrintAsBitmap property | Gets or sets a value indicating whether page metafiles should be converted to bitmaps and clipped to printer's hard margins prior to printing. |

The members listed in the table above may be used to work around certain printer issues. For instance, consider the following scenario of a machine running Windows Vista 64 with an HP-CP1700 printer (using Vista's built-in printer driver). In this example, if **ClipPage** was **False** (default) and a document page was wider than the printer's hard margin, empty pages were emitted and document content was not printed. For example, the following document produced just two empty pages if printed:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.Style.Font = New Font("Arial", 32)
For i As Integer = 0 To 19
    Dim rtx As New RenderText(i.ToString())
    rtx.X = String.Format("{0}in", i)
    rtx.Y = "10cm"
    rtx.Style.FontSize = 64
    doc.Body.Children.Add(rtx)
Next
doc.Generate()
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.Style.Font = new Font("Arial", 32);
for (int i = 0; i < 20; ++i)
{
    RenderText rtx = new RenderText(i.ToString());
    rtx.X = string.Format("{0}in", i);
    rtx.Y = "10cm";
    rtx.Style.FontSize = 64;
    doc.Body.Children.Add(rtx);
}
doc.Generate();
```

This issue can be now worked around by doing two things:

1. Set PrintAsBitmap to **True** (for example, on C1PreviewPane).
2. Attach the following event handler to the AdjustPrintPage event (available also via AdjustPrintPage):

   - Visual Basic

   ```
   Dim doc As New C1PrintDocument()
   doc.Style.Font = New Font("Arial", 32)
   Private  Sub PreviewPane_AdjustPrintPage(ByVal sender As Object, ByVal
   e As AdjustPrintPageEventArgs)
   ```

```
    Dim pa As RectangleF =  e.PrintableArea
    If Not e.PrintPageEventArgs.PageSettings.Landscape Then
        pa.Width = 800 ' System set to 824
        pa.X = 25 ' System set to 13
        pa.Y = 13 ' System set to 6.666...
    Else
        pa.X = 13
        pa.Y = 0
    End If
    e.PrintableArea = pa
End Sub
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.Style.Font = new Font("Arial", 32);
void PreviewPane_AdjustPrintPage(object sender,
AdjustPrintPageEventArgs e)
{
    RectangleF pa = e.PrintableArea;
    if (!e.PrintPageEventArgs.PageSettings.Landscape)
    {
        pa.Width = 800; // System set to 824
        pa.X = 25; // System set to 13
        pa.Y = 13; // System set to 6.666...
    }
    else {
        pa.X = 13;
        pa.Y = 0;
    }
    e.PrintableArea = pa;
}
```

This code fixes the wrong hard page margins set by the printer driver, and avoids the problem described above.

## Report Definition Language (RDL) Files

Import support for RDL files is included in C1PrintDocument. Report Definition Language (RDL) import allows reading RDL report definitions into an instance of the **C1PrintDocument** component. The resulting document is a data-bound representation of the imported report. RDL support in **C1PrintDocument** is based on the Microsoft RDL Specification for SQL Server 2008.

**Note:** RDL import in C1PrintDocument (provided by **ImportRdl** and **FromRdl** methods) is now obsolete. C1RdlReport should be used instead. See Working with C1RdlReport for more information.

You can use the following code to import an RDL file:

- Visual Basic

```
Dim doc As New C1PrintDocument()
doc.ImportRdl("myReport.rdl")
doc.Generate()
```

- C#

```
C1PrintDocument doc = new C1PrintDocument();
doc.ImportRdl("myReport.rdl");
doc.Generate();
```

Note that not all RDL properties are currently supported, but support will be added in future releases. For more information, see the RDL Import Limitations topic.

### RDL Import Limitations

The current implementation of RDL import in **Reports for WinFroms** has some limitations. These limitations include:

- Gauge, Chart, and SubReport objects are not supported.
- Expressions for sub-properties of complex RDL properties (such as border width) are not supported.
- Most aggregate functions in RDL have the optional "recursive" parameter. It is not supported.

The following RDL properties are not fully supported yet:

- QueryParameter.Value: Only a literal value may be specified.
- ReportParameter: Parameters referencing data are not supported.
- Hyperlink: Cannot be specified as expression; if several actions are associated with a hyperlink, only the first action is supported.
- ReportItem.Visibility: Cannot be specified as expression.
- ReportItem.Bookmark: Cannot be specified as expression.
- Style.TextAlign.General: Left align is used.

The following RDL properties are currently not supported:

- Document.AutoRefresh
- Document.CustomProperties
- Document.Code
- Document.Width
- Document.Language
- Document.CodeModules
- Document.Classes
- Document.ConsumeContainerWhitespace
- Document.DataTransform
- Document.DataSchema
- Document.DataElementName
- Document.DataElementStyle
- ConnectionProperties.Prompt
- ConnectionProperties.DataProvider
- DataSet.CaseSensitivity
- DataSet.Collation
- DataSet.AccentSensitivity
- DataSet.KanatypeSensitivity
- DataSet.WidthSensitivity
- DataSet.InterpretSubtotalsAsDetails
- Body.Height

- ReportItem.ToolTip
- ReportItem.DocumentMapLabel
- ReportItem.CustomProperties
- ReportItem.DataElementName
- ReportItem.DataElementOutput
- TextBox.HideDuplicates
- TextBox.ToggleImage
- TextBox.UserSort
- TextBox.DataElementStyle
- TextBox.ListStyle
- TextBox.ListLevel
- TextRun.ToolTip
- TextRun.MarkupType
- Style.Format
- Style.LineHeight
- Style.Direction
- Style.Language
- Style.Calendar
- Style.NumeralVariant
- Style.TextEffect

**Note:** RDL import in C1PrintDocument (provided by **ImportRdl** and **FromRdl** methods)  is now obsolete. C1RdlReport should be used instead. See Working with C1RdlReport for more information.

# Working with C1MultiDocument

The C1MultiDocument component is designed to allow creating, persisting, and exporting large documents that cannot be handled by a single C1PrintDocument object due to memory limitations.

A C1MultiDocument object provides a Items collection that can contain one or more elements of the type C1MultiDocumentItem. Each such element represents a C1PrintDocument. Use of compression and temporary disk storage allows combining several C1PrintDocument objects into a large multi-document that would cause an out of memory condition if all pages belonged to a single C1PrintDocument. The following snippet of code illustrates how a multi-document might be created and previewed:

- Visual Basic

```
Dim mdoc As New C1MultiDocument()
mdoc.Items.Add(C1PrintDocument.FromFile("myDoc1.c1dx"))
mdoc.Items.Add(C1PrintDocument.FromFile("myDoc2.c1dx"))
mdoc.Items.Add(C1PrintDocument.FromFile("myDoc3.c1dx"))
Dim pview As New C1PrintPreviewDialog()
pview.Document = mdoc
pview.ShowDialog()
```

- C#

```
C1MultiDocument mdoc = new C1MultiDocument();
mdoc.Items.Add(C1PrintDocument.FromFile("myDoc1.c1dx"));
mdoc.Items.Add(C1PrintDocument.FromFile("myDoc2.c1dx"));
mdoc.Items.Add(C1PrintDocument.FromFile("myDoc3.c1dx"));
C1PrintPreviewDialog pview = new C1PrintPreviewDialog();
pview.Document = mdoc;
pview.ShowDialog();
```

C1MultiDocument supports links between contained documents, common TOC, common page numeration, and total page count.

Note that a C1MultiDocument does not store references to the C1PrintDocument objects added to it – rather, it serializes them (as .c1d/x) and stores the result. Thus, you can create really large multi-documents without running out of memory – provided, of course, that your code itself does not keep references to the individual C1PrintDocument objects that were added to the C1MultiDocument. So when using C1MultiDocument please make sure that you do not keep references to the individual document objects after you have added them to the multi-document.

C1MultiDocument can be persisted as "C1 Open XML Multi Document" with the default extension of .c1mdx.

C1MultiDocument can be exported to most formats using any of the Export method overloads. See Exporting a C1MultiDocument File for details.

C1MultiDocument can be printed using any of the Print and PrintDialog methods overloads. See Printing a C1MultiDocument File for details.

## C1MultiDocument Limitations

The primary purpose of the C1MultiDocument component is to handle large documents that it would otherwise be impossible to create/export/print due to memory limitations. There are no limitations on the size and number of documents. But you may have to use disk storage rather than (default) memory, see the SetStorage methods for details.

A multi-document has a limitation that may or may not be significant depending on the specific application: it does NOT allow you to access the object model of **C1PrintDocument**s contained within. In other words, while you can write the following code:

- Visual Basic

```
Dim mdoc As New C1MultiDocument()
mdoc.Items.Add(C1PrintDocument.FromFile("file1.c1dx"))
mdoc.Items.Add(C1PrintDocument.FromFile("file2.c1dx"))
mdoc.Items.Add(C1PrintDocument.FromFile("file3.c1dx"))
```

- C#

```
C1MultiDocument mdoc = new C1MultiDocument();
mdoc.Items.Add(C1PrintDocument.FromFile("file1.c1dx"));
mdoc.Items.Add(C1PrintDocument.FromFile("file2.c1dx"));
mdoc.Items.Add(C1PrintDocument.FromFile("file3.c1dx"));
```

You CANNOT then add something like the following:

- Visual Basic

```
Dim doc As C1PrintDocument = mdoc.Items(1)
```

- C#

```
C1PrintDocument doc = mdoc.Items[1];
```

If that limitation is not an issue for a particular application, then you can even use the C1MultiDocument component to "modularize" the application even if there are no memory problems.

## Creating and Previewing a C1MultiDocument File

To add an item to the C1MultiDocumentItemCollection, you can use the Add method. To load a file into the C1MultiDocument component you can use the Load method. To remove a file, you would use the Clear method. This method clears any file previously loaded into the C1MultiDocument component.

To add an item to the C1MultiDocumentItemCollection, you can use the Add method. Complete the following steps:

1. In Design View, double-click on the form to open the Code Editor.
2. Add the following code to the **Load** event:

   - Visual Basic

   ```
   Dim ppc As New C1PrintPreviewControl
   Controls.Add(ppc)
   ppc.Dock = DockStyle.Fill
   Dim pdoc As New C1PrintDocument
   Dim pdoc2 As New C1PrintDocument
   Dim mdoc As New C1MultiDocument
   pdoc.Body.Children.Add(New C1.C1Preview.RenderText("Hello!"))
   pdoc2.Body.Children.Add(New C1.C1Preview.RenderText("World!"))
   mdoc.Items.Add(pdoc)
   mdoc.Items.Add(pdoc2)
   ppc.Document = mdoc
   mdoc.Generate()
   ```

   - C#

   ```
   C1PrintPreviewControl ppc = new C1PrintPreviewControl();
   Controls.Add(ppc);
   ppc.Dock = DockStyle.Fill;
   C1PrintDocument pdoc = new C1PrintDocument();
   C1PrintDocument pdoc2 = new C1PrintDocument();
   C1MultiDocument mdoc = new C1MultiDocument();
   pdoc.Body.Children.Add(new C1.C1Preview.RenderText("Hello!"));
   pdoc2.Body.Children.Add(new C1.C1Preview.RenderText("World!"));
   mdoc.Items.Add(pdoc);
   ```

```
mdoc.Items.Add(pdoc2);
ppc.Document = mdoc;
mdoc.Generate();
```

This code loads two C1PrintDocuments into the C1MultiDocument component and displays the documents in a C1PrintPreviewControl at run time.

## Exporting a C1MultiDocument File

C1MultiDocument can be exported to most formats using any of the Export method overloads. For example, in the following example the C1MultiDocument will be exported to a PDF file. The Boolean value, **True**, indicates that a progress dialog box should be shown.

- Visual Basic
  ```
  Me.C1MultiDocument1.Export("C:\exportedfile.pdf", True)
  ```
- C#
  ```
  this.c1MultiDocument1.Export(@"C:\exportedfile.pdf", true);
  ```

If you include the above code in a button's **Click** event handler, the **C1MultiDocument**'s content will be exported to a PDF file when the button is clicked at run time.

## Printing a C1MultiDocument File

C1MultiDocument can be printed using any of the Print and PrintDialog methods overloads. For example, the following code opens a **Print** dialog box.

- Visual Basic
  ```
  Me.C1MultiDocument1.PrintDialog()
  ```
- C#
  ```
  this.c1MultiDocument1.PrintDialog();
  ```

If you include the above code in a button's **Click** event handler, the **Print** dialog box will appear when the button is clicked at run time.

## C1MultiDocument Outlines

C1MultiDocument includes outline support. A collection of outline nodes specific to the multi-document may be specified via the Outlines property. The resulting outline (such as for the preview) is built as a combination of outline nodes in that collection and outline nodes in the contained documents. This outline can be built programmatically using the MakeOutlines() method.

The multi-document's own **Outlines** collection is processed first, and nodes from that collection are included in the resulting outline. If a node is also specified as the value of the **OutlineNode** of a contained C1MultiDocumentItem (for example, the two properties reference the same object), the whole outline of the document or report represented by that item is inserted into the resulting outline. Depending on the value of the multi-document item's **NestedOutlinesMode** property, the outline of the document or report is either nested within the outline node, or replaces it. Finally, outlines of documents and reports represented by items that are not included in the multi-document's **Outlines** collection are automatically appended to the resulting outline sequentially.

Outlines support is provided by the following properties and methods:

- Outlines property
- MakeOutlines method
- Outlines property
- OutlineNode property

# Reports for WinForms Samples

Please be advised that this ComponentOne software tool is accompanied by various sample projects and/or demos which may make use of other development tools included with the ComponentOne Studios.

> **Note:** The ComponentOne Samples are also available at http://helpcentral.componentone.com/Samples.aspx.

## Reporting Samples

:

- Visual Basic Samples

| Sample | Description |
|---|---|
| Chart | Add charts to reports using **C1Report** and **C1Chart**. |
| CreateReport | Create reports dynamically using code. This sample uses the **C1Report** component. |
| CustomData | Create custom data source objects for use with C1Report. This sample uses the **C1Report** and **C1PrintPreview** component. |
| Embedded | Load report definitions into the **C1Report** component at design time. This sample uses the **C1Report** and **C1PrintPreview** component. |
| HtmlFields | Render reports to HTML preserving HTML formatting. |
| Newsletter | Create reports without data sources (unbound reports). This sample uses the **C1Report** and **C1PrintPreview** components. |
| NorthWind | View reports imported from the NorthWind database. This sample uses the **C1Report** component. |

- C# Samples

| Sample | Description |
|---|---|
| AddScriptObject | Add custom objects to C1Report's script engine. |
| AdHocSorting | Select the sorting criteria before rendering the report. |
| ADOReport | Use ADODB.Recordset objects as **C1Report** data sources. |
| Chart | Add charts to reports using **C1Report** and **C1Chart**. This sample uses the **C1Report** and **C1Chart** components. |
| CreateReport | Create reports dynamically using code. This sample uses the **C1Report** component. |
| CustomFields | Create custom Chart and Gradient fields that can be added to any report. |
| CustomHyperlinks | Perform custom actions when hyperlinks are clicked. |
| CustomPaperSize | Create reports that use custom paper sizes. This sample uses the **C1Report** and **C1PrintPreview** components. |

| | |
|---|---|
| DynamicFormat | Use script properties to format the report based on its contents. This sample uses the **C1Report** component. |
| Email | Send reports by e-mail. |
| ExportXml | Export reports to an XML format. |
| FlexReport | Use a **C1FlexGrid** control as a data source for your reports. |
| HierReport | Create reports based on hierarchical data. This sample uses the **C1Report** component. |
| HtmlFields | Render reports to HTML preserving HTML formatting. |
| Images | Load images into a report using two methods. |
| MixedOrientation | Renders two **C1Reports** (one portrait, one landscape) into a single PDF document. |
| PageCountGroup | Keep separate page counts for each group in a report. |
| ParameterizedFilter | Create reports with a parameterized filter. |
| ParseParameters | Parse a PARAMETERS statement in a RecordSource string. This sample uses the **C1Report** component. |
| ProgressIndicator | Display a progress indicator form while a report is rendered. |
| ReportBrowser | Open report definition files and list their contents. This sample uses the **C1Report** component. |
| ReportBuilder | Create report definitions automatically based on DataTables. |
| ReportDictionary | Add a custom look up dictionary object to C1Report's script engine. |
| RTFReport | Shows how to render RTF fields in a report. This sample uses the **C1Report** component. |
| SubReportDataSource | Use custom data sources with subreports. This sample uses the **C1Report** component. |
| XMLData | Use any XML document as a report data source. This sample uses the **C1Report** component. |
| ZipReport | Compress and encrypt report definition files. This sample uses the **C1Report** and **C1Zip** components. |

- XML Samples

| Sample | Description |
|---|---|
| CommonTasks | A collection of reports that show how to perform common tasks. |
| SampleReports | XML report definition files that show C1Report's features. |

### Printing and Previewing Samples

- Visual Basic and C# Samples

| Sample | Description |
|---|---|
| AutoSizeTable | The sample shows how to adjust the widths of a table's columns based on their content. The sample provides a method **AutoSizeTable** that can be used as-in in any application that needs to automatically size tables based on their content. |
| CoordinatesOfCharsInText | Shows how to use the **GetCharRect()** method (advanced).<br>The sample shows how to use the **GetCharRect()** method available on RenderText and RenderParagraph classes, which allows to find out the position and size of individual characters in the text. In the sample, a red rectangle is drawn around each character. |
| DataBinding | The sample demonstrates binding to a simple list (including binding to an empty list), binding to a MS Access database, the use of grouping, aggregate functions, and binding of table row/column groups. This sample requires a 2006 v3 version (C1Preview.2 2.0.20063.41002) or later. |
| Hyperlinks | Shows how to create various types of hyperlinks<br>The sample demonstrates how to create and set up several types of hyperlinks supported by C1PrintDocument and the preview controls: hyperlinks to anchors within the same document, hyperlinks to anchors in other C1PrintDocument objects, hyperlinks to locations within the document (render objects, pages), hyperlinks to external files/URLs. |
| ObjectCoordinates | The sample shows how to connect the coordinates in the preview pane with the objects in the C1PrintDocument being previewed. Methods are provided that find the RenderObject currently under the mouse, query the properties of the object, highlight it in the preview, and manipulate it: change the object's background color, text, or other properties. The changes are immediately reflected in the document.<br>Note that for highlighting to work this sample requires a 2006 v2 version (C1Preview.2.0.20062.40855) or later. |
| PageLayout1 | Shows how to use the PageLayouts property.<br>The sample creates a document with different page layouts for the first page, even pages and odd pages. The different layouts are specified declaratively via the PageLayouts property of C1PrintDocument, no even handling is involved. |
| PageLayout2 | Shows how to use the LayoutChangeBefore property of RenderObject.<br>The sample creates a document with an object that forces a page break, and a different page layout that is "nested" within the current layout, so that the current layout is automatically restored when the nested object is over. |
| RenderObjects | Introduces most of the RenderObject types provided by C1PrintDocument.<br>The sample creates and previews a C1PrintDocument, in which most of the RenderObject types provided by C1PrintDocument are included: RenderArea, RenderText, RenderGraphics, RenderEmpty, RenderImage, RenderRichText, RenderPolygon, RenderTable, RenderParagraph. |
| RenderTOC | Shows how to use the RenderToc object.<br>The sample demonstrates how to create the table of contents for a document using the dedicated RenderToc render object. |
| RotatedText | The sample shows how to insert rotated text into **C1PrintDocument.Text** rotated at different angles is shown. |
| Stacking | Shows how to use stacking rules for render objects' positioning<br>The sample demonstrates how to use the **RenderObject.Stacking** |

| | |
|---|---|
| | property to set stacking rules for block (top to bottom and left to right) and inline (left to right) positioning of objects. Relative positioning of objects is also demonstrated. |
| Tables1 | Shows how to create tables, set up table headers and footers.<br>The sample creates and previews a C1PrintDocument with a table. Demonstrates how to set up table headers (including running headers) and footers. Shows how to add orphan control (the minimum rows printed on the same page before the footer is specified). |
| Tables2 | The sample shows the basic features of tables in C1PrintDocument.<br>The following features of tables are demonstrated:<br>• Table borders (GridLines property, allowing to specify the 4 outer and 2 inner lines).<br>• Borders around individual cells and groups of cells.<br>• Style attributes (including borders) for groups of disconnected cells.<br>• Cells spanning rows and columns.<br>• Content alignment within the cells (spanned or otherwise).<br>• Table headers and footers.<br>• Tags (such as page number/total page count) in table footer.<br>• Style attributes: borders, font and background images. |
| Tables3 | Shows multiple inheritance of styles in C1PrintDocument tables.<br>The sample demonstrates multiple inheritance of styles in tables. A table with some test data is inserted into the document. Some style attributes are redefined for the styles of a row, a column, and a cell group. Cells at the intersections of the groups inherit styles for all, combining them. |
| TabPosition | Shows how to use the **TabPosition** property of text rendering objects.<br>The sample creates a document with a RenderParagraph object, on which the **TabPositions** property is defined, specifying the tab positions calculated on document reflow depending on the current page width. |
| VisibleRowsCols | Demonstrates the Visible property of table rows/columns.<br>The sample shows the Visible property of RenderTable's rows and columns, that allows to you hide table rows and columns without removing them from the table. This sample requires a 2006 v3 version (C1Preview.2 2.0.20063.41002) or later. |
| WideTables | Shows how to create wide tables spanning several pages<br>The sample demonstrates the feature of C1PrintDocument which allows rendering of wide objects spanning multiple pages horizontally. To enable this feature, the object's **CanSplitHorz** property should be set to **True**. The preview is also adjusted to better show wide objects (margins are hidden, the gap between pages set to zero, and the end user is prevented from showing the margins). |
| WrapperDoc | This sample provides source code for a very simple wrapper around the new C1PrintDocument implementing some of the RenderBlock/Measure methods from the "classic" (old) **C1PrintDocument**. This sample may be especially useful to facilitate conversions of applications using the classic preview to the new preview. |
| ZeroWidthRowsCols | Demonstrates the treatment of table columns with zero width.<br>The sample shows that columns with zero width/rows with zero height are not rendered at all (as if their Visible property were set to **False**). This |

| | |
|---|---|
| | sample requires a 2006 v3 version (C1Preview.2 2.0.20063.41002) or later. |

# Reports for WinForms Task-Based Help

The task-based help assumes that you are familiar with programming in .NET, have a basic knowledge of reports, and know how to use controls in general. By following the steps outlined in the help, you will be able to create projects demonstrating a variety of **Reports for WinForms** features, and get a good sense of what the **Reports for WinForms** components can do.

## Reporting Task-Based Help

The task-based help assumes that you are familiar with programming in .NET, have a basic knowledge of reports, and know how to use controls in general. By following the steps outlined in the help, you will be able to create projects demonstrating a variety of C1Report features, and get a good sense of what the C1Report component can do.

Most of this section's topics have pre-built reports that illustrate them. The pre-built reports can be found in the **CommonTasks.xml** report definition file, which is available for download from the ComponentOne HelpCentral Sample page.

Note that you should have the following namespace referenced to your project:

- C1.C1Report

### *Adding Images to the Report*

Using the **C1ReportDesigner**, you can add unbound or bound images and create watermarks.

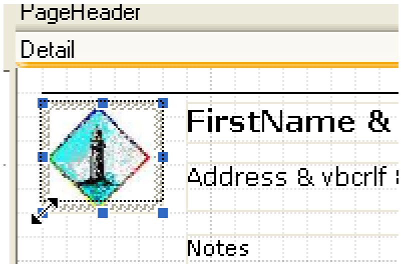
## Creating Unbound Images

Unbound images are static images such as logos and watermarks that are not stored in the database. To add unbound image fields to your report, complete the following tasks:

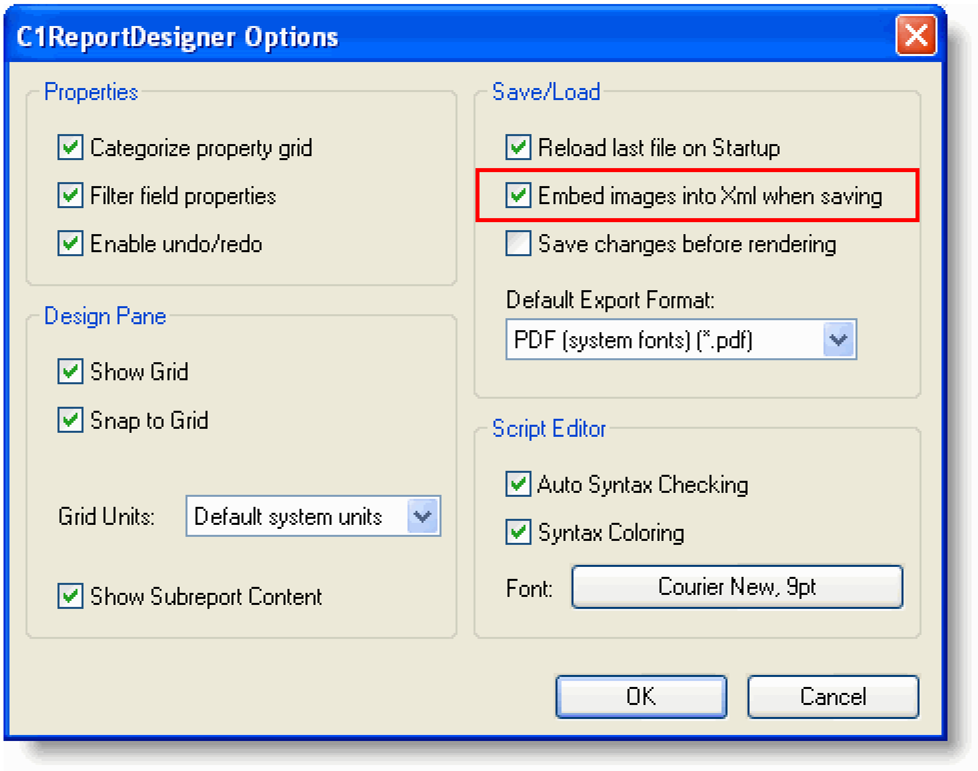
1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Design** button to begin editing the report.
4. In Design mode, click the **Add Unbound Picture** button located in the **Fields** group of the **Design** tab.

   The **Open** dialog box appears.
5. Select the image file you want to include in the report, and click **Open**.
6. Click on your report where you would like to place the image, and then resize the field to show the image.

   The following unbound image has been added to the report and is being resized:



Note that the image file can be embedded in the report definition, or it can be a reference to an external file. To choose the option you prefer, in the Designer select the Application button and in the menu that appears select **Options**. The **C1ReportDesigner Options** dialog box appears where you can make your selection:



C1ReportDesigner Options

Properties

☑ Categorize property grid

☑ Filter field properties

☑ Enable undo/redo

Design Pane

☑ Show Grid

☑ Snap to Grid

Grid Units: Default system units

☑ Show Subreport Content

Save/Load

☑ Reload last file on Startup

☑ Embed images into Xml when saving

□ Save changes before rendering

Default Export Format: PDF (system fonts) (*.pdf)

Script Editor

☑ Auto Syntax Checking

☑ Syntax Coloring

Font: Courier New, 9pt

**Sample Report Available**

For the complete report, see report "03: Unbound Images" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Creating Bound Images

Bound images are images stored in database fields. To display these images in your reports, add a field to the report and set its **Picture** property to a string containing the name of the column where the image is stored.

**To add bound image fields to your report using the C1ReportDesigner:**

1. In Design mode of the **C1ReportDesigner**, click the **Add Bound Picture** button located in the **Fields** group of the **Design** tab.

   This shows a menu with all binary fields in the current data source.

2. Select the field you want to add to the report.

**To add bound image fields to your report using code:**

If the field "Photo" in the database contains embedded OLE objects or raw image streams, and the report contains a field called "fEmployeePhoto", then the following code would display the employee photo in the field:

- Visual Basic

```
fEmployeePhoto.Picture = "Photo"
```

- C#

```
fEmployeePhoto.Picture = "Photo";
```

**Sample Report Available**

For the complete report, see report "04: Bound Images" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Creating a Watermark

Watermarks are images displayed behind the report content. The images are often washed-out to prevent them from interfering with the actual report content.

To display an image as a watermark, set the Picture property to a file that contains the image. You can also control the way the watermark is scaled and the pages on which it should appear using the PictureAlign and PictureShow properties.

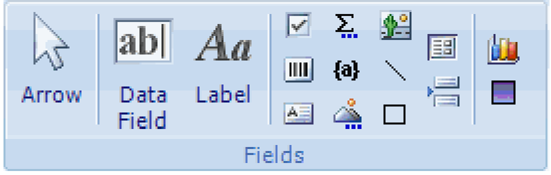
**Sample Report Available**

For the complete report, see report "05: Watermark" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

### *Creating Report Fields*

With the report definition loaded into the component and a data source defined, you can add and edit report fields.

The **Fields** group in the **C1ReportDesigner**'s **Design** tab allows you to easily add fields to your report. For button descriptions, see Enhancing the Report with Fields.



To add a field to your report, click any of these buttons and complete the following steps:

- Drag the mouse over the report and the cursor changes into a cross-hair ┼. Click and drag to define the rectangle that the new field will occupy, and then release the button to create the new field.

  If you change your mind, hit the ESC key or click the **Undo** button to cancel the operation.

  Note that C1Report only has one type of **Field** object. The buttons simply set some properties on the **Field** object to make it look and act in a certain way.

  OR

- You can also add fields by copying and pasting existing fields, or by holding down the CTRL key and dragging a field or group of fields to a new position to create a copy.

## Creating Charts

In the initial versions of C1Report, adding charts to reports required handling the StartSection event, generating the chart, and assigning the chart image to a field's Picture property. This is not hard to do, and continues to be the most flexible way to add dynamic images to reports.

However, this approach has two drawbacks:

- It requires you to write code outside the report definition, which means only your application will be capable of showing the report the way it is meant to be shown.
- It requires you to write code for generating the chart, which can be tedious.

The current C1Report supports custom report fields, including a chart field that is based on the **C1Chart** control.

To add a chart field to a Group Header section in your report, complete the following steps:

1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In the **Fields** group of the **Design** tab, click the **Add Chart Field** button .
5. Click in the Group Header section of your report and drag the field to resize the chart.
6. From the Properties window, set the chart field's **Chart.DataX** and **Chart.DataY** properties to the values you want to display on the chart. You can show several series by setting the **Chart.DataY** property to a list of fields delimited by semicolons (for example, "UnitsInStock;ReorderLevel").

The chart data is automatically scoped to the current report group. For example, when rendering the "Beverages" section, only data for that category will be charted. You can customize the chart using many other properties such as **Chart.ChartType**, **Chart.GridLines**, **Chart.Use3D**, and **Chart.Palette** properties.

For more information on creating chart fields, see the Adding Chart Fields topic.

**Sample Report Available**

For the complete report, see report "11: Charts" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Creating Custom Fields

You can create your own custom fields and add them to the Report Designer palette. To do that, you have to:

1. Create a custom field class that derives from **C1.Win.C1Report.Field**.
2. Register your custom field assembly in the Report Designer's settings file.

This is how the **Chart** and **Gradient** fields are implemented. The source code for these custom fields is available; you can use it as a starting point to create your own custom fields. The **Chart** and **Gradient** fields are registered in the **C1ReportDesigner** settings file with this entry:

```
<customfields>
  <item
value="C1.Win.C1Report.CustomFields;C1.Win.C1Report.CustomFields.Chart" />
  <item
value="C1.Win.C1Report.CustomFields;C1.Win.C1Report.CustomFields.Gradient"
/>
</customfields>
```

For example, to add a new field to the Designer palette, add your control to the `<customfields>` section in the "C1ReportDesigner.2.exe.settings" file:

```
<customfields>
<item
value="C1.Win.C1Report.CustomFields.2;C1.Win.C1Report.CustomFields.Chart"
/>
<item
value="C1.Win.C1Report.CustomFields.2;C1.Win.C1Report.CustomFields.Gradien
t" />

<!-- THIS LINE WILL ADD A NEW FIELD TO THE DESIGNER -->
<item value="MyCustomFieldAssembly;MyCustomFieldAssembly.MyField" />
</customfields>
```

This assumes that your field is called "MyField" and it can be found in the assembly called "MyCustomFieldAssembly"

**Sample Report Available**

For the complete report, see report "12: Custom Fields" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

### *Customizing the Page Headers*

This section explains how to customize the behavior of page headers.

## Adding a Continued Label to Headers on Page Breaks

Group Header sections are repeated across page breaks if their Repeat property is set to **True**. This makes the report easier to read, but it can be hard to tell if a group header on a page marks the beginning of a group or is just a continuation.

One way to address this is to add a field with a "Continued" label named, **fContinued**, for example to the group header and control its visibility with script. To do this, complete the following steps:

1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.

2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.

3. Click the **Close Print Preview** button to begin editing the report.

4. In Design mode, select **Detail** from the drop-down list above the Properties window.

5. Locate the Detail.OnPrint property and click the empty field next to it, and then click the **ellipsis** button.

6. The **VBScript Editor** appears. Enter the following VBScript expression in the code editor:
   ```
   ' VBScript: Detail.OnPrint
   fContinued.Visible = true
   ```

7. Then select **GroupFooter** from the drop-down list above the Properties window.

8. Locate the GroupFooter.OnPrint property and click the empty field next to it, and then click the **ellipsis** button.

9. The **VBScript Editor** appears. Enter the following VBScript expression in the code editor:
   ```
   ' VBScript: GroupFooter.OnPrint
   fContinued.Visible = false
   ```

If the **fContinued** field is initially invisible, then the script will show the label only on continued page headers. This script ensures that the **fContinued** field is visible within the group. Any page breaks created after the group footer and before the next Detail section will not show the label.

**Sample Report Available**

For the complete report, see report "18: Continued Headers" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Changing Page Headers Dynamically

To specify whether Page Header and Page Footer sections should appear on all pages, or be suppressed on the pages that contain the report Header and report Footer sections use C1Report's PageHeader and PageFooter properties.

Sometimes you may want to further customize this behavior. For example, you may want to render different headers on odd and even pages. This can be done with some script that shows or hides fields depending on the page being rendered. To do this, complete the following steps:

1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In Design mode, select **Detail** from the drop-down list above the Properties window.
5. Locate the OnFormat property and click the empty field next to it, and then click the **ellipsis** button.
6. The **VBScript Editor** appears. Enter the following VBScript expression in the code editor:

   ```
   odd = (page mod 2 <> 0)
   h1odd.Visible = odd
   h2odd.Visible = odd
   h1even.Visible = not odd
   h2even.Visible = not odd
   ```

   This script will show or hide fields for odd and even pages if a report header contains several fields named "h<x>odd" and "h<x>even".

   Note that to prevent the page header from showing blank spaces, all the fields should have the CanShrink property set to **True**.

**Sample Report Available**

For the complete report, see report "09: Dynamic Page Header" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

### *Customizing the Page Layout*

The following topics explain how you can customize the layout of your report.

## Controlling Page Breaks

By default, C1Report fills each page until the bottom, inserts a page break, and continues rendering in the next page. You can override this behavior using several properties:

- Group.KeepTogether: Determines whether Group Header sections are allowed to render on a page by themselves, if they must be rendered with at least one Detail section, or if the entire group should be kept together on a page.
- Section.KeepTogether: Determines whether page breaks are allowed within sections. It has lower precedence than Group.KeepTogether.
- ForcePageBreak: Allows you to specify that page breaks should be inserted before, after, or before and after the section.
- Field.KeepTogether: Determines whether page breaks are allowed within fields. This allows long **Text** fields to span multiple pages. It has lower precedence than Section.KeepTogether.
- ForcePageBreak: Allows you to specify that page breaks should be inserted before, after, or before and after the field.

Set these properties through the Properties grid of the **C1ReportDesigner**.

You can use script to change the properties while the report is being rendered. For example, to cause page breaks after each 10 Detail sections, complete the following steps:

1. Open the **C1ReportDesigner** application. For more information on how to access **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In Design mode, select **Detail** from the drop-down list above the Properties window.
5. Locate the OnPrint property and click the empty field next to it, and then click the **ellipsis** button.
6. The **VBScript Editor** appears. Enter the following VBScript expression in the code editor:

```
cnt = cnt + 1
detail.forcepagebreak = "none"
if cnt >= 10 then
  cnt = 0
  detail.forcepagebreak = "after"
endif
```

**Sample Report Available**

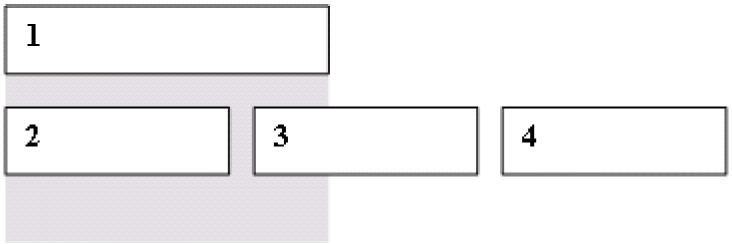
For the complete report, see report "07: Force Page Breaks" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Creating CanGrow/CanShrink Fields

It is common for report fields to have content that may span multiple lines or collapse to no lines at all. In some cases, you may want to allow these fields to grow or shrink to fit their content rather than clip the excess or leave white spaces in the report.

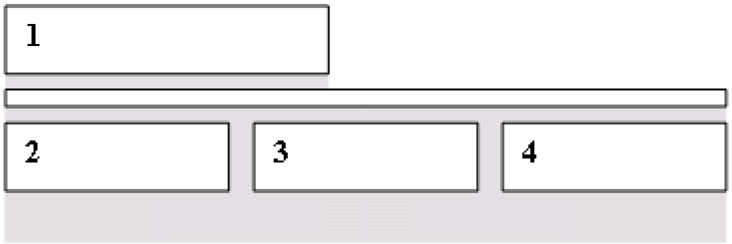
To do this, in Design mode of the **C1ReportDesigner** set the **Field** object's CanGrow and CanShrink properties to **True**.

Fields that grow push down the fields below them. Likewise, fields that can shrink push up the fields below them. Below in this case means "strictly" below, as shown in the following diagram:



Field 1 will push or pull fields 2 and 3 when it grows or shrinks. Field 4 will not be affected because it is not directly below field 1. The shaded area in the diagram shows the region affected by field 1.

If you want field 4 to remain aligned with fields 2 and 3, add an extra field spanning the whole area above fields 2 and 3. The new field will be pushed down by field 1 and will in turn push fields 2, 3, and 4. The following diagram shows this new layout:



**Sample Report Available**

For the complete report, see report "06: CanGrow CanShrink" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Creating a Gutter Margin

Gutter margins are extra space added to the margins next to the binding. They make it easier to bind the pages into folders, brochures, and so on.

To add a gutter margin to a report, you should increase the MarginLeft property on odd pages and use the default value on even pages. This can be done with script. To add script that changes the margins based on the page being rendered, complete the following steps:

1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In Design mode, select **Detail** from the drop-down list above the Properties window.
5. Locate the OnPrint property and click the empty field next to it, and then click the **ellipsis** button.
6. The **VBScript Editor** appears. Enter the following VBScript expression in the code editor:

```
' VBScript: Report.OnOpen
gutter = report.layout.marginleft ' initialize variable
```

```
' VBScript: Report.OnPage
report.layout.marginleft = _
  Iif(page mod 2 = 1, gutter, gutter - 1440)
```

**Sample Report Available**

For the complete report, see report "10: Gutter" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Defining and Using Global Constants

There is no special mechanism for defining and using global constants in a report, but you can add hidden fields to the report and use their values as global parameters. To do this, complete the following steps:

1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In the **Fields** group of the **Design** tab, click the **Add Label** button to add a field to your report.
5. Click on your report where you want the field placed and drag to resize the field.
6. Set the following properties for the field:
   - Field.Name = **linesPerPage**
   - Field.Text = **14**
   - Field.Visible = **False**
7. To control the number of detail lines per page, use script. Select **Detail** from the drop-down list above the Properties window.
8. Locate the OnPrint property and click the empty field next to it, and then click the **ellipsis** button.

   The **VBScript Editor** appears.
9. Enter the following VBScript expression in the code editor:

   ```
   cnt = cnt + 1
   detail.forcepagebreak = "none"
   if cnt >= linesPerPage then
     cnt = 0
     detail.forcepagebreak = "after"
   endif
   ```

   Note that the value in the **linesPerPage** field can be set prior to rendering the report, by changing the field's **Text** property.

**Sample Report Available**

For the complete report, see report "08: Global Constant" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Specifying Custom Paper Size

By default, C1Report creates reports using the default paper size on the default printer.

You can specify the paper size and orientation using the PaperSize and Orientation properties. However, C1Report checks that the selected paper size is available on the current printer before rendering, and changes to the default paper size if the selected setting is not available.

If you want to specify a certain paper size and use it regardless of the printers available, set the PaperSize property to **Custom**, and set the CustomWidth and CustomHeight properties to the page dimensions (in *twips*).

**To specify a custom paper size of 25" x 11" for your report using the C1ReportDesigner:**

1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In Design mode, select your report from the drop-down list above the Properties window.
5. Locate **Layout** and expand the property node to view all available properties.
6. Set the **Custom Height** property to **25"** or **25in**.

   Notice that the measurement is converted into *twips* automatically. The Property window display the measurement in *twips* (36000).
7. Set the **Custom Width** property to **11"** or **11in**.

   The Property window displays the measurement in *twips* (15840).
8. Set the **PaperSize** property to **Custom**.

**To specify a custom paper size of 25" x 11" for your report using code:**

Regardless of what is available on the printer, the following code sets the report paper to 25" x 11":

- Visual Basic

```
c1r.Layout.PaperSize = PaperKind.Custom
c1r.Layout.CustomHeight = 25 * 1440 ' in twips
c1r.Layout.CustomWidth = 11 * 1440
```

- C#

```
c1r.Layout.PaperSize = PaperKind.Custom;
c1r.Layout.CustomHeight = 25 * 1440; // in twips
c1r.Layout.CustomWidth = 11 * 1440;
```

**Sample Report Available**

For the complete report, see report "02: Custom Paper Size" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## *Formatting Reports*

The following topics show how to apply formatting to your report. By simply modifying properties in the Properties window or adding a few lines of script to your VBScript expression, you can visually enhance your report.

## Adding Alternating Background Color

To create a report with alternating background color, use the OnPrint property of the Detail section to change the BackColor property of the section. To do this, complete the following steps:

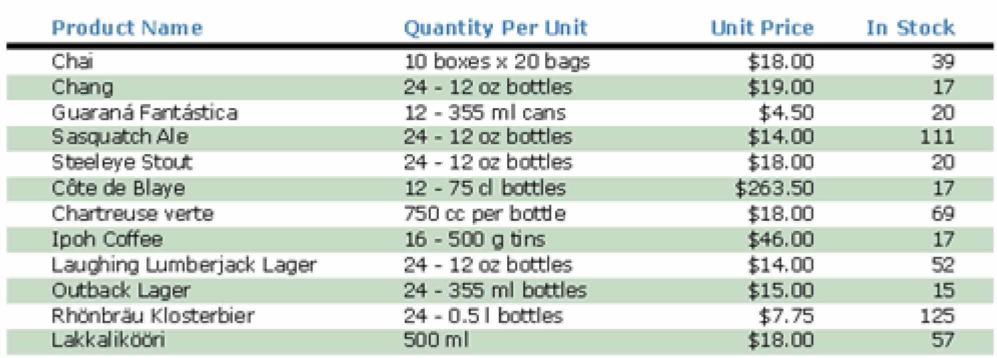
1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In Design mode, select the report from the drop-down list above the Properties window.
5. Locate the OnOpen property and enter **cnt = 0**. This initializes the cnt variable.
6. Next, select **Detail** from the drop-down list above the Properties window.
7. Locate the OnPrint property and click the empty field next to it, and then click the **ellipsis** button.
8. The **VBScrpit Editor** appears. Enter the following VBScript expression in the code editor:

```
cnt = cnt + 1
if cnt mod 2 = 0 then
  detail.backcolor = rgb(200,220,200)
else
  detail.backcolor = rgb(255,255,255)
endif
```

9. Click the **Preview** button to preview the report with alternating background.

**This topic illustrates the following:**

This report illustrates the alternating background color.



| Product Name | Quantity Per Unit | Unit Price | In Stock |
|---|---|---|---|
| Chai | 10 boxes x 20 bags | $18.00 | 39 |
| Chang | 24 - 12 oz bottles | $19.00 | 17 |
| Guaraná Fantástica | 12 - 355 ml cans | $4.50 | 20 |
| Sasquatch Ale | 24 - 12 oz bottles | $14.00 | 111 |
| Steeleye Stout | 24 - 12 oz bottles | $18.00 | 20 |
| Côte de Blaye | 12 - 75 cl bottles | $263.50 | 17 |
| Chartreuse verte | 750 cc per bottle | $18.00 | 69 |
| Ipoh Coffee | 16 - 500 g tins | $46.00 | 17 |
| Laughing Lumberjack Lager | 24 - 12 oz bottles | $14.00 | 52 |
| Outback Lager | 24 - 355 ml bottles | $15.00 | 15 |
| Rhönbräu Klosterbier | 24 - 0.5 l bottles | $7.75 | 125 |
| Lakkalikööri | 500 ml | $18.00 | 57 |

Whenever a Detail section is rendered, the counter is incremented and the **BackColor** property of the Detail section is toggled.

**Sample Report Available**

For the complete report, see report "01: Alternating Background (Greenbar report)" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Adding Conditional Formatting

In some cases you may want to change a field's appearance depending on the data it represents. For example, you may want to highlight items that are expensive, or low in stock. This can be done with script.

To do this, complete the following steps:

1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In Design mode, select **Detail** from the drop-down list above the Properties window (since this section contains the fields to add conditional formatting to).
5. Locate the OnFormat property and click the empty field next to it, and then click the **ellipsis** button.
6. The **VBScript Editor** appears. Enter the following VBScript expression in the code editor:

```
' VBScript: Detail.OnFormat
If UnitsInStock + UnitsOnOrder < ReorderLevel And _
   Discontinued = False Then
  Detail.BackColor = rgb(255,190,190)
Else
  Detail.BackColor = vbWhite
Endif
```

   The script changes the Detail section's BackColor property depending on the value of the fields **UnitsInStock**, **UnitsOnOrder**, **ReorderLevel**, and **Discontinued**.

**Sample Report Available**

For the complete report, see report "16: Conditional Formatting" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Editing the Field's Format Based on Value

You can change a report field's format based on its value by specifying an expression for the Detail section's OnFormat property.

To specify an expression for the OnFormat property, complete the following steps:

1. Open the **C1ReportDesigner**. For more information on how to access the **C1ReportDesigner**, see Accessing C1ReportDesigner from Visual Studio.
2. Create a new report or open an existing report. Once you have the report in the **C1ReportDesigner**, you can modify the report properties.
3. Click the **Close Print Preview** button to begin editing the report.
4. In Design mode, select **Detail** from the Property window's drop-down list to view the available properties for the Detail section.
5. Locate the OnFormat property and click the **ellipsis** button next to the property.
6. The **VBScript Editor** appears where you can specify an expression.

   The following expression changes the UnitsInStock field's ForeColor to red if the sum of the UnitsInStock value and the UnitsOnOrder value is less than the ReorderLevel value. There are several ways to write this expression.

   **Option 1:**

```
UnitsInStockCtl.Forecolor = Iif(UnitsInStock + UnitsOnOrder <
ReorderLevel, vbRed, vbBlack)
```

**Option 2:**

```
lowStock = UnitsInStock + UnitsOnOrder < ReorderLevel
UnitsInStockCtl.Forecolor = Iif(lowStock, vbRed, vbBlack)
```

**Option 3:**

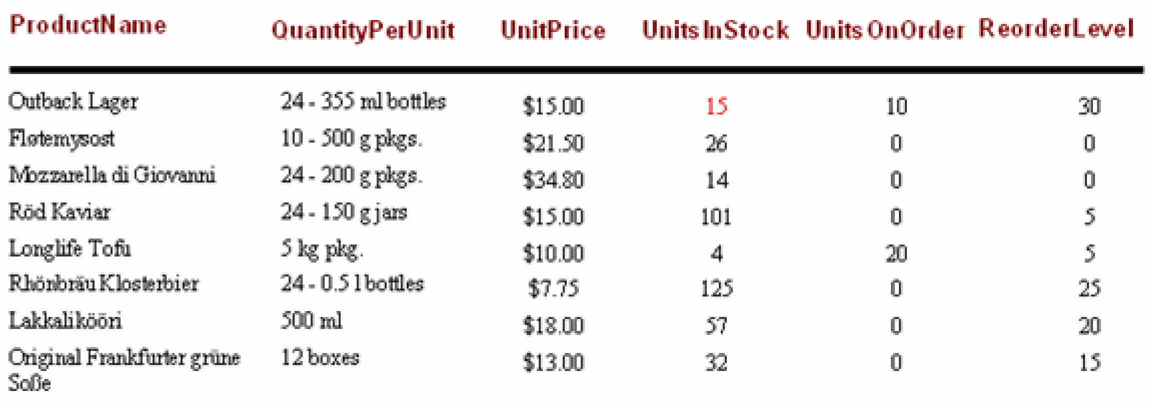
```
If UnitsInStock + UnitsOnOrder < ReorderLevel Then
    UnitsInStockCtl.Forecolor = vbRed
Else
    UnitsInStockCtl.Forecolor = vbBlack
End If
```

**Option 4:**

```
color = Iif(UnitsInStock + UnitsOnOrder < ReorderLevel, vbred, vbblack)
UnitsInStockCtl.Forecolor = color
```

**This topic illustrates the following:**

Notice that the Outback Lager's UnitsInStock value is formatted in red since the sum of the UnitsInStock and UnitsOnOrder is less than the ReorderLevel.



| ProductName | QuantityPerUnit | UnitPrice | UnitsInStock | UnitsOnOrder | ReorderLevel |
|---|---|---|---|---|---|
| Outback Lager | 24 - 355 ml bottles | $15.00 | 15 | 10 | 30 |
| Fløtemysost | 10 - 500 g pkgs. | $21.50 | 26 | 0 | 0 |
| Mozzarella di Giovanni | 24 - 200 g pkgs. | $34.80 | 14 | 0 | 0 |
| Röd Kaviar | 24 - 150 g jars | $15.00 | 101 | 0 | 5 |
| Longlife Tofu | 5 kg pkg. | $10.00 | 4 | 20 | 5 |
| Rhönbräu Klosterbier | 24 - 0.5 l bottles | $7.75 | 125 | 0 | 25 |
| Lakkalikööri | 500 ml | $18.00 | 57 | 0 | 20 |
| Original Frankfurter grüne Soße | 12 boxes | $13.00 | 32 | 0 | 15 |

## Suppressing or Forcing the Display of Zeros

To suppress the display of fields with value zero, set their Format property to #. The pound sign is a formatting symbol that displays only significant digits (no leading or trailing zeros).

To force the display of a certain number of digits, use a format like "0000". The zero forces the display of a digit, including leading and trailing zeros.

Each format string can have up to three sections separated by semi-colons. If two sections are provided, the first is used for positive values and zero, the second for negative values. If three sections are provided, the first is used for positive values, the second for negative values, and the third for zero. For example: "#;(#);ZERO".

**Sample Report Available**

For the complete report, see report "21: Suppress or Force Zeros" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

### *Loading Report Definitions*

C1Report works by combining a report definition with raw data to create reports. In order to create reports, you need to load the report definition into C1Report. The following topics explain several ways to load report definitions.

## Loading a Report Definition from a File

You can use the **C1ReportDesigner** to create report definition files (XML files that may contain one or more report definitions). For details on how to use the **C1ReportDesigner**, see the Working with C1ReportDesigner section of the documentation.

**To load a report definition from a file at design time:**

To load a report definition from a file at design time, complete one of the following tasks:

- Right-click the **C1Report** component and select the **Load Report** menu option.

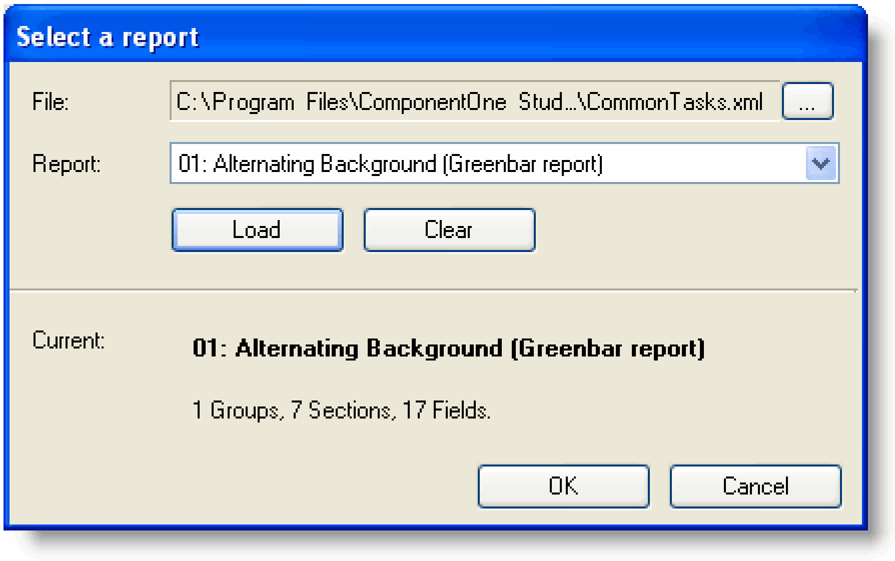
  OR

- Click the smart tag (▸) above the **C1Report** component and select **Load Report** from the **C1Report Tasks** menu.

Using the **Select a report** dialog box to select the report you want, complete the following tasks:

1. Click the **ellipsis** button. The **Open** dialog box appears and you can select the XML file.
2. The available report definitions are listed in the **Report** drop-down box. Select the report definition to load.
3. Click **Load** and **OK** to close the dialog box.

This is what the report selector dialog box looks like:



Select a report

File: C:\Program Files\ComponentOne Stud...\CommonTasks.xml

Report: 01: Alternating Background (Greenbar report)

Current: **01: Alternating Background (Greenbar report)**

1 Groups, 7 Sections, 17 Fields.

**To load a report definition from a file using code:**

To load a report definition from a file, use the Load method. It takes as parameters the name of the report definition file and the name of the report you want to load. If you want to list the reports contained in a report definition file, use the GetReportInfo method. It returns a list of the reports in the file.

For example:

- Visual Basic
```
' Get list of reports in a report definition file
Dim reports As String() = c1r.GetReportInfo(reportFile)

' Load first report into C1Report component
c1r.Load(reportFile, reports(0))
```
- C#
```
// Get list of reports in a report definition file
string[] reports = c1r.GetReportInfo(reportFile);

// Load first report into C1Report component
c1r.Load(reportFile, reports[0]);
```

## Loading a Report Definition from a String

C1Report has a ReportDefinition property that allows you to get or set the entire report definition as a string. This is a convenient way to store and retrieve report definitions in databases or in data structures within your application.

The **ReportDefinition** string contains the exact same XML that would be stored in the report definition file, for example:

- Visual Basic
```
' Load report definition into C1Report component
c1r.Load(reportFile, reportName)

' Copy report definition to the clipboard
Dim repDef As String = c1r.ReportDefinition
Clipboard.SetDataObject(repDef)

' Copy report definition to c1r2 component
c1r2.ReportDefinition = repDef
```
- C#
```
// Load report definition into C1Report component
c1r.Load(reportFile, reportName);

// Copy report definition to the clipboard
string repDef = c1r.ReportDefinition;
Clipboard.SetDataObject(repDef);

// Copy report definition to c1r2 component
c1r2.ReportDefinition = repDef;
```

### *Modifying Subreports*

This section shows how to modify *subreports*. Subreports are regular reports contained in a field in another report - the main report - that are usually designed to display detail information based on a current value in the main report, in a master-detail scenario. For more information on subreports, see the Subreport property in the reference section.

## Adding Page Headers to Subreports

C1Report ignores Page Header and Page Footer sections in subreports. Instead, it uses the Page Header and Page Footer sections defined in the main report. This is the same behavior as in Microsoft Access.

In many cases, however, you would like your subreports to include header information across page breaks. To do this, place the headers in a Group Header section and set the section's Repeat property to **True**. If your subreport doesn't have any groups, add an empty one.

**Sample Report Available**

For the complete report, see report "14: Page Headers in Subreports" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

## Retrieving Values from Subreports

In some cases you may want to pass data from the subreport back to the main report. Script variables can't be used for this because each report has its own script scope (this avoids the possibility of conflicting variable names).

To pass data from a subreport back to the main report, you have to store values in subreport fields or in the subreport's Tag property, then have the main report read those values.

In this sample, the subreport calculates the average unit price per product category and stores that value in its **Tag** property. The main report retrieves and displays that value.

**Sample Report Available**

For the complete report, see report "15: Retrieve Values from Subreports" in the **CommonTasks.xml** report definition file, which is available for download from the **CommonTasks** sample on the ComponentOne HelpCentral Sample page.

### *Rendering Reports (Previewing, Printing, and Exporting)*

Once the report definition has been loaded into the component and a data source has been defined, you can render the report to the printer, into preview controls, or to report files.

## Displaying a Progress Indicator While the Report Renders

Most preview applications have progress indicators that show which page is being rendered and have a button that allows the user to cancel the report while it is being generated. The .NET print preview controls provide this automatically for you. If you are printing the report directly to the printer or exporting it to a file, however, there is no built-in progress report UI.

Use C1Report events to create a progress report dialog, or to update a status bar while the report is being rendered. The StartPage and EndReport events are sufficient to provide feedback on which page is being printed and when the report is finished. For example, this code uses the StartPage event to provide feedback through a status bar (StatusStrip1):

- Visual Basic

```
Private Sub c1r_StartPage(ByVal sender As System.Object, ByVal e As
C1.Win.C1Report.ReportEventArgs) Handles c1r.StartPage
      StatusStrip1.Text = String.Format("Rendering page {0} of '{1}'...",
c1r.Page, c1r.ReportName)
End Sub
```

- C#

```
private void c1r_StartPage(object sender, ReportEventArgs e)
{
      statusStrip1.Text = string.Format("Rendering page {0} of '{1}'...",
      c1r.Page, c1r.ReportName);
}
```

To cancel the report before it is finished, add a **Cancel** button to your application and use it to set the **C1Report**.Cancel property to **True**. For example:

- Visual Basic

```
Private Sub _btnCancel_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCancel.Click
   c1r.Cancel = True
   Close()
End Sub
```

- C#

```
private void _btnCancel_Click(object sender, System.EventArgs e)
{
      c1r.Cancel = true;
      Close();
}
```

Note that you may also want to provide progress bars and "page n of m" indicators, but that is generally difficult to do because the page count is not known until the report has been rendered.

**Sample Project Available**

For a complete sample using a progress indicator, see the **ProgressIndicator** sample located on the ComponentOne HelpCentral Sample page.

## Previewing Reports

To preview the report, use the use the **C1Report**.Document property. Assign it to the **Document** property in the **Reports for WinForms** preview control or to the .NET PrintPreview or PrintPreviewDialog controls and the preview controls will display the report and allow the user to browse, zoom, or print it. For example:

- Visual Basic

```
' Load report definition
c1r.Load(reportFile, reportName)

' Preview the document
c1preview1.Document = c1r.Document
```

- C#

```
// Load report definition
c1r.Load(reportFile, reportName);

// Preview the document
c1preview1.Document = c1r.Document;
```

**Note:** C1Report works with the .NET preview components, but it is optimized to work with the included **Reports for WinForms** preview controls. When used with the included controls, you can see each report page as it is generated. With the standard controls, you have to wait until the entire report is ready before the first page is displayed.

## Printing Reports

To print a report directly to the printer, use the **C1Report**.Document property. This property returns a standard **PrintDocument** object that has a **Print** method and exposes printer and page settings.

For example, the following code shows a print dialog and prints the report:

- Visual Basic

```
' Load report definition
c1r.Load(reportFile, reportName)

' Get PrintDocument object
PrintDocument doc = c1r.Document

' Show a PrintDialog so user can customize the printing
Dim pd As PrintDialog = New PrintDialog()

' Use PrinterSettings in report document
pd.PrinterSettings = doc.PrinterSettings

' Show the dialog and print the report
If pd.ShowDialog() = DialogResult.OK Then
      doc.Print()
End If

' Cleanup and release PrintDialog resources
pd.Dispose()
```

- C#

```
// Load report definition
c1r.Load(reportFile, reportName);

// Get PrintDocument object
PrintDocument doc = c1r.Document;

// Show a PrintDialog so user can customize the printing
PrintDialog pd = new PrintDialog();

// Use PrinterSettings in report document
pd.PrinterSettings = doc.PrinterSettings;

// Show the dialog and print the report
if (pd.ShowDialog() == DialogResult.OK)
doc.Print();

// Cleanup and release PrintDialog resources
pd.Dispose();
```

## Exporting the Report

### Exporting the report to common file formats

C1Report has a RenderToFile method that allows you to export your report to several file formats, including HTML, RTF, PDF, TIFF, Text, and XLS. For example, the following code creates PDF and XLS versions of a report:

- Visual Basic

```
' Load report definition
c1r.Load(reportFile, reportName)

' Export to PDF
c1r.RenderToFile(outFile + ".pdf", FileFormatEnum.PDF)
c1r.RenderToFile(outFile + ".xls", FileFormatEnum.Excel)
```

- C#

```
// Load report definition
```

```
c1r.Load(reportFile, reportName);

// Export to PDF
c1r.RenderToFile(outFile + ".pdf", FileFormatEnum.PDF);
c1r.RenderToFile(outFile + ".xls", FileFormatEnum.Excel);
```

**Note:** When a document is exported to the RTF or the DOCX formats with the "preserve pagination" option selected, text is placed in text boxes and the ability to reflow text in the resulting document may be limited.

### Exporting the report to custom formats

If you want to export the report to a format that is not supported by C1Report, you can write your own export filter class and use the **C1Report**.RenderToFilter method to render the report into your custom filter.

Custom filters are classes that derive from the **C1.Win.C1Report.**ExportFilter class and override a few simple methods like StartReport, StartSection, RenderField, EndSection, and EndReport.

Writing a custom export filter is not difficult. It can be used, for example, to create custom XML representations of your reports for later processing by other applications.

## *Saving a Report Definition*

When you are done creating and viewing your report in the **C1ReportDesigner** application, select the **Application** button and select **Save** to save the report definition file. The Designer saves the report definition in XML format, which can be read back into the Designer or directly into a C1Report component.

# Printing and Previewing Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET. By following the steps outlined in the help, you will be able to create projects demonstrating a variety of **Reports for WinForms** previewing features and get a good sense of what **ComponentOne Reports for WinForms** can do. If you've never used **Reports for WinForms** before, you may want to work through the C1PrintDocument Quick Start first to familiarize yourself with the product.

For task-based help topic create a new .NET project and place a C1PrintDocument and a **C1PrintPreview** component on the form and set C1PrintPreviewControl1's **Document** property to **C1PrintDocument1**. Also, you should have the following namespaces referenced to your project:

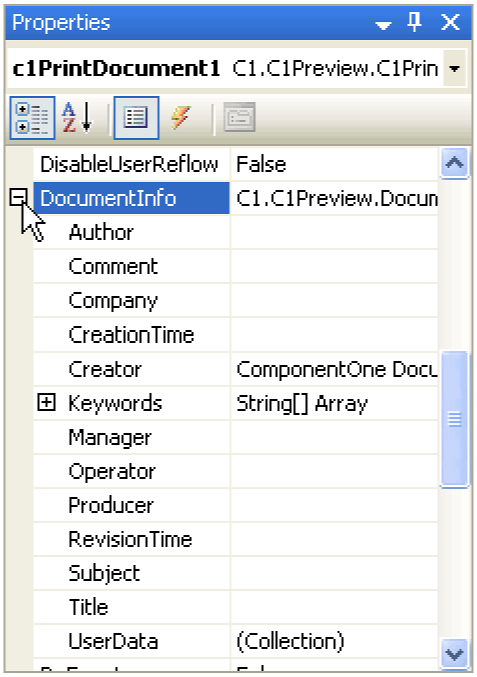
- C1.Win.C1Preview
- C1.C1Preview

## Setting Document Information

To set the document information, enter values in the Author, Comment, Company, CreationTime, Creator, Keywords, Manager, Operator, Producer, RevisionTime, Subject, and Title properties either in the designer or in code.

### In the Designer

1. In the C1PrintDocument Properties window, locate the DocumentInfo property and expand the property node.



c1PrintDocument1 C1.C1Preview.C1Prin

| | |
|---|---|
| DisableUserReflow | False |
| DocumentInfo | C1.C1Preview.Docun |
| Author | |
| Comment | |
| Company | |
| CreationTime | |
| Creator | ComponentOne Docu |
| Keywords | String[] Array |
| Manager | |
| Operator | |
| Producer | |
| RevisionTime | |
| Subject | |
| Title | |
| UserData | (Collection) |

2. Enter values and press ENTER when finished to set the property.

> **Note:** Some properties have built-in editors to help you set the property, such as:

Clicking the **ellipsis** ... button next to the **Keywords** property opens the **String Collection Editor**, where you can enter keywords for the document.

Clicking the drop-down arrow next to the **CreationTime** or **RevisionTime** properties allows you to choose a date.

The default value for the Creator property is **ComponentOne Document Engine**.

### In Code

The following code should be added to **the Form_Load** event.

To set the Author property, add the following code:

- Visual Basic

```
Me.C1PrintDocument1.DocumentInfo.Author = "Jane Doe"
```

- C#

```
this.c1PrintDocument1.DocumentInfo.Author = "Jane Doe";
```

To set the Comment property, add the following code:

- Visual Basic

```
Me.C1PrintDocument1.DocumentInfo.Comment = "This is a C1PrintDocument
file."
```

- C#

```
this.c1PrintDocument1.DocumentInfo.Comment = "This is a C1PrintDocument
file.";
```

To set the Company property, add the following code:

- Visual Basic

```
Me.C1PrintDocument1.DocumentInfo.Company = "ComponentOne"
```

- C#

```
this.c1PrintDocument1.DocumentInfo.Company = "ComponentOne";
```

To set the CreationTime property, add the following code:

- Visual Basic

```
Me.C1PrintDocument1.DocumentInfo.CreationTime = "2/29/08"
```

- C#

```
this.c1PrintDocument1.DocumentInfo.CreationTime = "2/29/08";
```

To set the Creator property, add the following code. The default value is **ComponentOne Document Engine**.

- Visual Basic

```
Me.C1PrintDocument1.DocumentInfo.Creator = "C1PrintPreview"
```

- C#

```
this.c1PrintDocument1.DocumentInfo.Creator = "C1PrintPreview";
```

To set the Keywords property, add the following code. Keywords should be separated with a space.

- Visual Basic

```
Me.C1PrintDocument1.DocumentInfo.Keywords = New String() {"C1PrintPreview
ComponentOne C1PrintDocument"}
```

- C#

```
this.c1PrintDocument1.DocumentInfo.Keywords = new string()
{"C1PrintPreview ComponentOne C1PrintDocument"}
```

To set the Manager property, add the following code:

- Visual Basic
```
Me.C1PrintDocument1.DocumentInfo.Manager = "John Smith"
```
- C#
```
this.c1PrintDocument1.DocumentInfo.Manager = "John Smith";
```

To set the Operator property, add the following code:

- Visual Basic
```
Me.C1PrintDocument1.DocumentInfo.Operator = "Joe Brown"
```
- C#
```
this.c1PrintDocument1.DocumentInfo.Operator = "Joe Brown";
```

To set the Producer property, add the following code:

- Visual Basic
```
Me.C1PrintDocument1.DocumentInfo.Producer = "ComponentOne Preview for
.NET"
```
- C#
```
this.c1PrintDocument1.DocumentInfo.Producer = "ComponentOne Preview for
.NET";
```

To set the RevisionTime property, add the following code:

- Visual Basic
```
Me.C1PrintDocument1.DocumentInfo.RevisionTime = "2/29/08"
```
- C#
```
this.c1PrintDocument1.DocumentInfo.RevisionTime = "2/29/08";
```

To set the Subject property, add the following code:

- Visual Basic
```
Me.C1PrintDocument1.DocumentInfo.Subject = "Document Creation"
```
- C#
```
this.c1PrintDocument1.DocumentInfo.Subject = "Document Creation";
```

To set the Title property, add the following code:

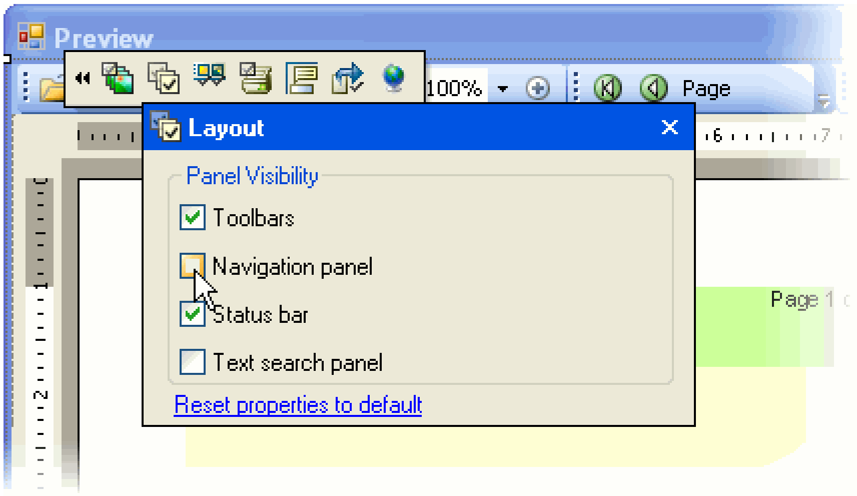
- Visual Basic
```
Me.C1PrintDocument1.DocumentInfo.Title = "Creating Documents with
C1PrintPreview"
```
- C#
```
this.c1PrintDocument1.DocumentInfo.Title = "Creating Documents with
C1PrintPreview";
```

## Hiding the Navigation Panel

To hide the navigational panel, set the NavigationPanelVisible property to **False**. This can be set at either in the designer or in code.

### In the Smart Designer

1. Open the **Main Menu** floating toolbar.
2. Select the **Layout** button to open the **Layout** dialog box.
3. Uncheck **Navigation panel** to set the NavigationPanelVisible property to **False**.



Layout

Panel Visibility

☑ Toolbars

□ Navigation panel

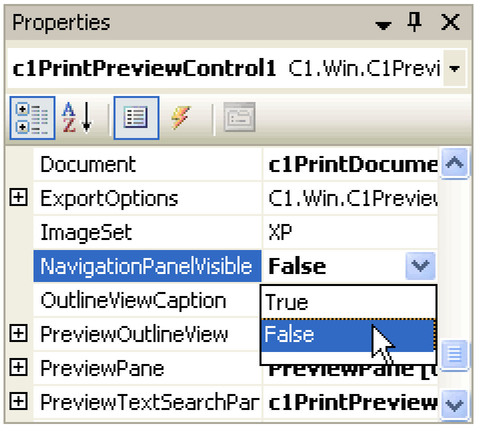
☑ Status bar

□ Text search panel

Reset properties to default

### In the Properties Window

1. Select the C1PrintPreviewControl and navigate to the Properties window.
2. Set the NavigationPanelVisible property to **False**.



| | |
|---|---|
| Document | c1PrintDocume |
| ⊞ ExportOptions | C1.Win.C1Previe |
| ImageSet | XP |
| NavigationPanelVisible | False |
| OutlineViewCaption | |
| ⊞ PreviewOutlineView | |
| ⊞ PreviewPane | PreviewPane [ |
| ⊞ PreviewTextSearchPar | c1PrintPreview |

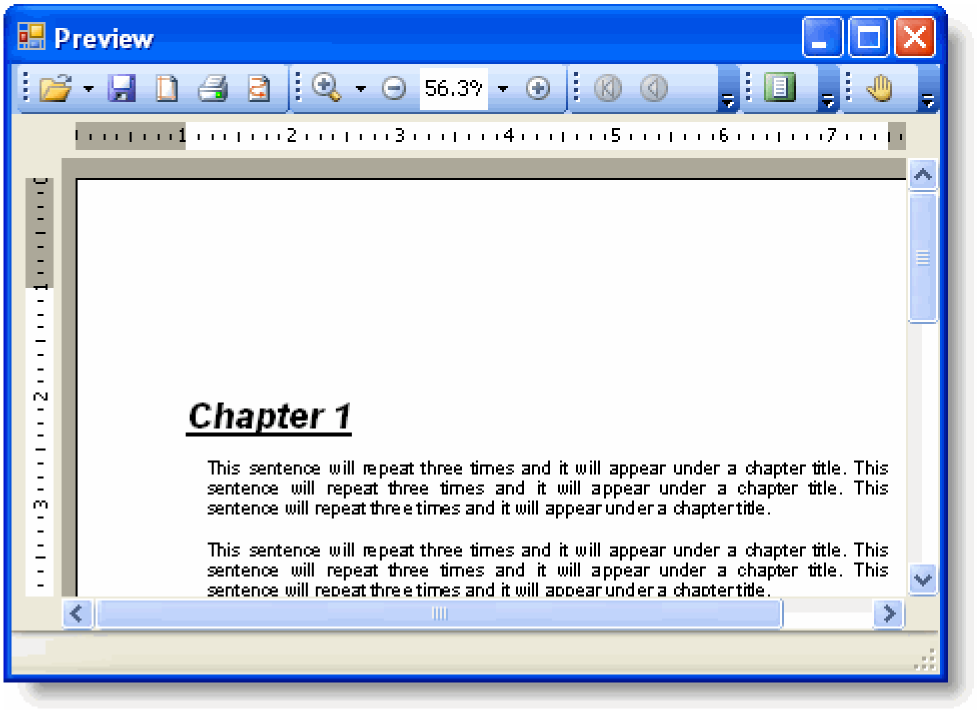
### In Code

Add the following code to the **Form_Load** event:

- Visual Basic
```
Me.C1PrintPreviewControl1.NavigationPanelVisible = False
```
- C#
```
this.c1PrintPreviewControl1.NavigationPanelVisible = false;
```

### ✓ What You've Accomplished

The navigation panel will not be visible:



Chapter 1

This sentence will repeat three times and it will appear under a chapter title. This sentence will repeat three times and it will appear under a chapter title. This sentence will repeat three times and it will appear under a chapter title.

This sentence will repeat three times and it will appear under a chapter title. This sentence will repeat three times and it will appear under a chapter title. This sentence will repeat three times and it will appear under a chapter title.

## Adding Outline Entries to the Outline Tab

To add outline entries to the **Outline** tab, use the **OutlineNodeCollection.Add** method.

1. From the Toolbox, add the **C1PrintPreviewControl** and **C1PrintDocument** controls to your project.
2. Click **C1PrintPreviewControl1** to select it and in the Properties window set its **Document** property to **C1PrintDocument1**.
3. Add the following code to the **Form_Load** event:
   - Visual Basic

```
' Make the document.
MakeDoc()

' Generate the document.
Me.C1PrintDocument1.Generate()
```

   - C#

```
// Make the document.
MakeDoc();

// Generate the document.
this.c1PrintDocument1.Generate();
```

4. Add the **MakeDoc** subroutine, which uses the **OutlineNodeCollection.Add** method to add outline entries to the **Outline** tab:
   - Visual Basic

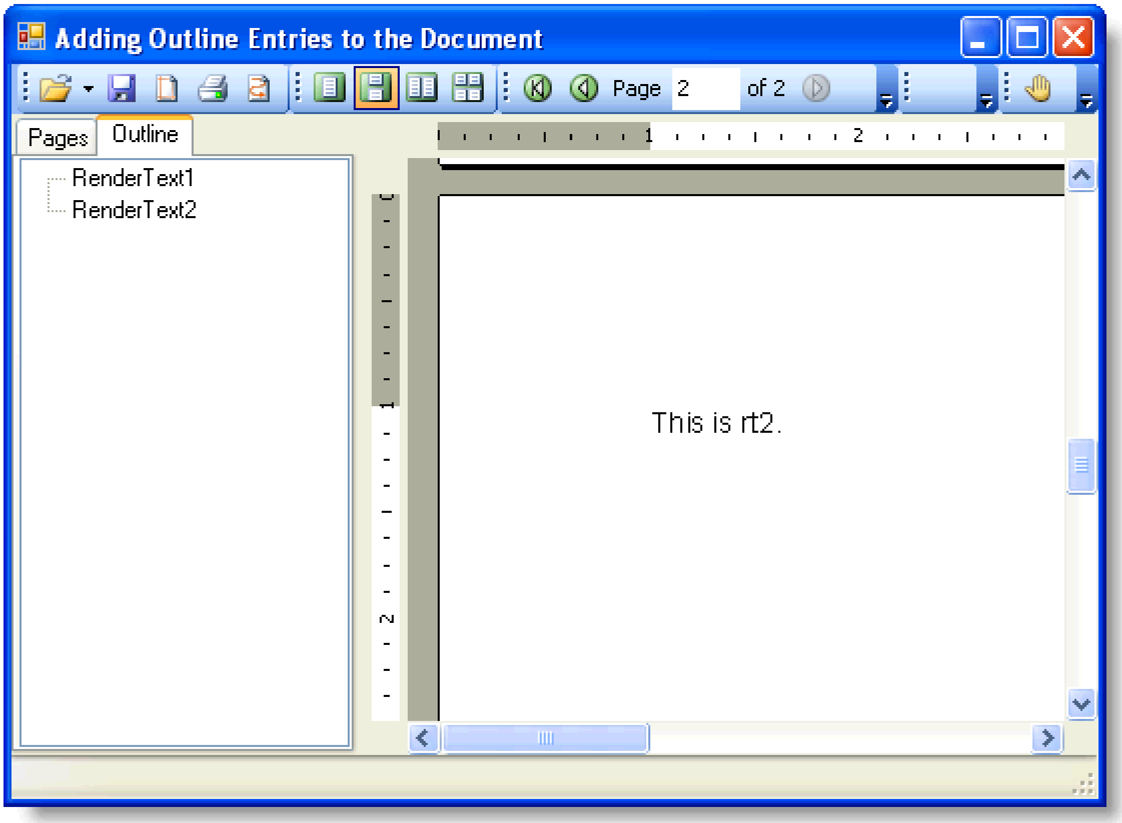
```
Private Sub MakeDoc()

    ' Create RenderText1.
    Dim rt1 As New C1.C1Preview.RenderText
    rt1.Text = "This is RenderText1."
    ' Add an outline entry point for RenderText1.
    Me.C1PrintDocument1.Outlines.Add("RenderText1", rt1)

    ' Insert a page break.
    rt1.BreakAfter = C1.C1Preview.BreakEnum.Page

    ' Create RenderText2.
    Dim rt2 As New C1.C1Preview.RenderText
    rt2.Text = "This is RenderText2."

    ' Add an outline entry point for RenderText2.
    Me.C1PrintDocument1.Outlines.Add("RenderText2", rt2)

    ' Add the RenderText to the document.
    Me.C1PrintDocument1.Body.Children.Add(rt1)
    Me.C1PrintDocument1.Body.Children.Add(rt2)
End Sub
```

- C#

```
private void MakeDoc()
{

    // Create RenderText1.
    C1.C1Preview.RenderText rt1 = new C1.C1Preview.RenderText();
    rt1.Text = "This is RenderText1.";

    // Add an outline entry point for RenderText1.
    this.c1PrintDocument1.Outlines.Add("RenderText1", rt1);

    // Add a page break.
    rt1.BreakAfter = C1.C1Preview.BreakEnum.Page;

    // Create RenderText2.
    C1.C1Preview.RenderText rt2 = new C1.C1Preview.RenderText();
    rt2.Text = "This is RenderText2.";

    // Add an outline entry point for RenderText2.
    this.c1PrintDocument1.Outlines.Add("RenderText2", rt2);

    // Add the RenderText to the document.
    this.c1PrintDocument1.Body.Children.Add(rt1);
    this.c1PrintDocument1.Body.Children.Add(rt2);
}
```

### What You've Accomplished

The outline entries "RenderText1" and "RenderText2" are added to the **Outline** tab:



## Adding Columns to a Page

To add columns to a page, use the Add method.

1. Navigate to the Toolbox, and add the **C1PrintPreviewControl** and **C1PrintDocument** controls to your project.

2. Click **C1PrintPreviewControl1** to select it and in the Properties window set its **Document** property to **C1PrintDocument1**.

3. Add the following code to the **Form_Load** event:

   - Visual Basic

```
' Make the document.
MakeDoc()

' Generate the document.
Me.C1PrintDocument1.Generate()
```

   - C#

```
// Make the document.
MakeDoc();

// Generate the document.
this.c1PrintDocument1.Generate();
```

4. Add the **MakeDoc** subroutine, which uses the Add method to add columns to all of the pages in the document:

   - Visual Basic

```
Private Sub MakeDoc()
```

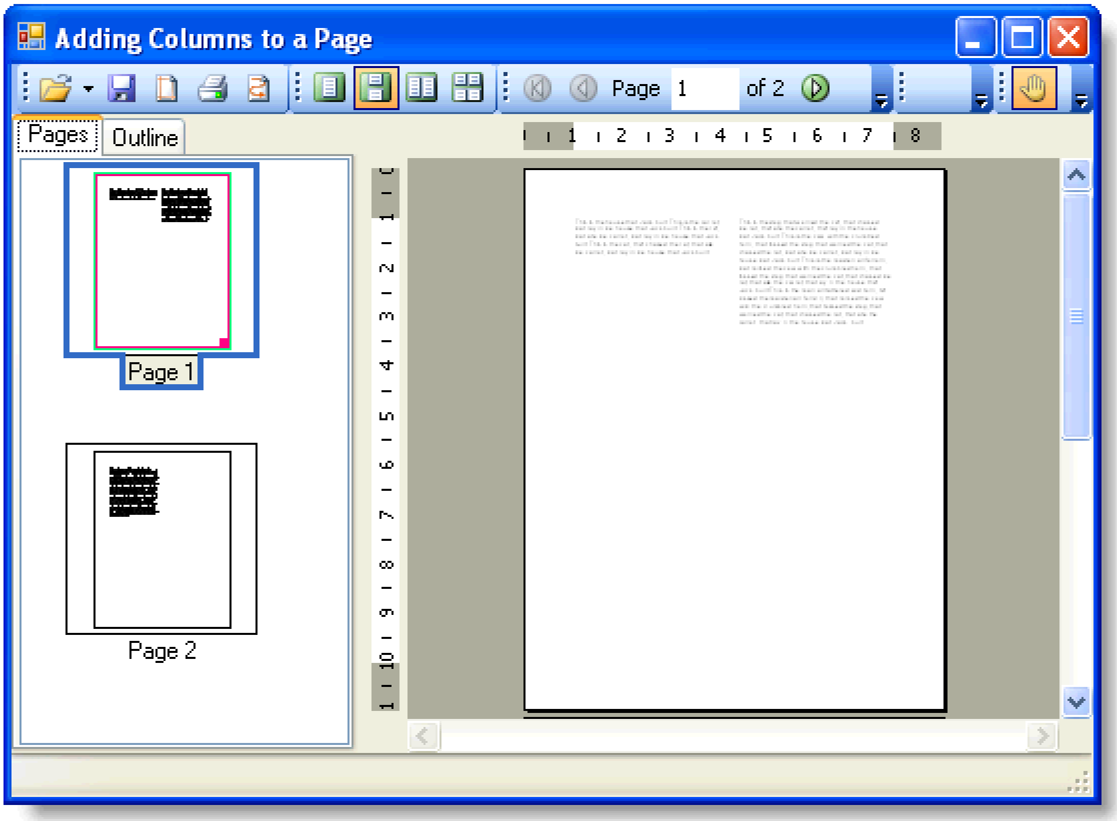
```
    ' Create page layout.
    Dim pl As New C1.C1Preview.PageLayout

    ' Add columns.
    pl.Columns.Add()
    pl.Columns.Add()
    pl.PageSettings = New C1.C1Preview.C1PageSettings()
    Me.C1PrintDocument1.PageLayouts.Default = pl

    ' Create RenderText1.
    Dim rt1 As New C1.C1Preview.RenderText
    rt1.Text = "This is the house that Jack built. This is the carrot,
that lay in the house that Jack built. This is the rat, that ate the
carrot, that lay in the house that Jack built. This is the cat, that
chased the rat, that ate the carrot, that lay in the house that Jack
built."

    ' Insert column break.
    rt1.BreakAfter = C1.C1Preview.BreakEnum.Column

    ' Create RenderText2.
    Dim rt2 As New C1.C1Preview.RenderText
    rt2.Text = "This is the dog that worried the cat, that chased the
rat, that ate the carrot, that lay in the house that Jack built. This
is the cow with the crumbled horn, that tossed the dog, that worried
the cat, that chased the rat, that ate the carrot, that lay in the
house that Jack built. This is the maiden all forlorn, that milked the
cow with the crumbled horn, that tossed the dog, that worried the cat,
that chased the rat, that ate the carrot, that lay in the house that
Jack  built. This is the man all tattered and torn, that kissed the
maiden all forlorn, that milked the cow with the crumbled horn, that
tossed the dog, that worried the cat, that chased the rat, that ate the
carrot, that lay in the house that Jack  built."

    ' Insert column break.
    rt2.BreakAfter = C1.C1Preview.BreakEnum.Column

    ' Create RenderText3.
    Dim rt3 As New C1.C1Preview.RenderText
    rt3.Text = "This is the priest all shaven and shorn, that married
the man all tattered and torn, that kissed the maiden all forlorn, that
milked the cow with the crumbled horn, that tossed the dog, that
worried the cat, that chased the rat, that ate the carrot, that lay in
the house that Jack built. This is the cock that crowed in the morn,
that waked the priest all shaven and shorn, that married the man all
tattered and torn, that kissed the maiden all forlorn, that milked the
cow with the crumbled horn, that tossed the dog, that worried the cat,
that chased the rat, that ate the carrot, that lay in the house that
Jack built. This is the farmer sowing the corn, that kept the cock that
crowed in the morn, that waked the priest all shaven and shorn, that
married the man all tattered and torn, that kissed the maiden all
forlorn, that milked the cow with the crumbled horn, that tossed the
dog, that worried the cat, that chased the rat, that ate the carrot,
that lay in the house that Jack built."

    ' Add the RenderText to the document.
    Me.C1PrintDocument1.Body.Children.Add(rt1)
    Me.C1PrintDocument1.Body.Children.Add(rt2)
    Me.C1PrintDocument1.Body.Children.Add(rt3)
End Sub
```

- C#

```
public void MakeDoc()
{

    // Create page layout.
    C1.C1Preview.PageLayout pl = new C1.C1Preview.PageLayout();

    // Add columns.
    pl.Columns.Add();
    pl.Columns.Add();
    pl.PageSettings = new C1.C1Preview.C1PageSettings();
    this.c1PrintDocument1.PageLayouts.Default = pl;

    // Create RenderText1.
    C1.C1Preview.RenderText rt1 = new C1.C1Preview.RenderText();
    rt1.Text = "This is the house that Jack built. This is the carrot,
that lay in the house that Jack built. This is the rat, that ate the
carrot, that lay in the house that Jack built. This is the cat, that
chased the rat, that ate the carrot, that lay in the house that Jack
built.";

    // Insert column break.
    rt1.BreakAfter = C1.C1Preview.BreakEnum.Column;

    // Create RenderText2.
    C1.C1Preview.RenderText rt2 = new C1.C1Preview.RenderText();
    rt2.Text = "This is the dog that worried the cat, that chased the
rat, that ate the carrot, that lay in the house that Jack built. This
is the cow with the crumbled horn, that tossed the dog, that worried
the cat, that chased the rat, that ate the carrot, that lay in the
house that Jack built. This is the maiden all forlorn, that milked the
cow with the crumbled horn, that tossed the dog, that worried the cat,
that chased the rat, that ate the carrot, that lay in the house that
Jack built. This is the man all tattered and torn, that kissed the
maiden all forlorn, that milked the cow with the crumbled horn, that
tossed the dog, that worried the cat, that chased the rat, that ate the
carrot, that lay in the house that Jack built.";

    // Insert column break.
    rt2.BreakAfter = C1.C1Preview.BreakEnum.Column;

    // Create RenderText3.
    C1.C1Preview.RenderText rt3 = new C1.C1Preview.RenderText();
    rt3.Text = "This is the priest all shaven and shorn, that married
the man all tattered and torn, that kissed the maiden all forlorn, that
milked the cow with the crumbled horn, that tossed the dog, that
worried the cat, that chased the rat, that ate the carrot, that lay in
the house that Jack built. This is the cock that crowed in the morn,
that waked the priest all shaven and shorn, that married the man all
tattered and torn, that kissed the maiden all forlorn, that milked the
cow with the crumbled horn, that tossed the dog, that worried the cat,
that chased the rat, that ate the carrot, that lay in the house that
Jack built. This is the farmer sowing the corn, that kept the cock that
crowed in the morn, that waked the priest all shaven and shorn, that
married the man all tattered and torn, that kissed the maiden all
forlorn, that milked the cow with the crumbled horn, that tossed the
dog, that worried the cat, that chased the rat, that ate the carrot,
that lay in the house that Jack built.";

    // Add the RenderText to the document.
    this.c1PrintDocument1.Body.Children.Add(rt1);
    this.c1PrintDocument1.Body.Children.Add(rt2);
    this.c1PrintDocument1.Body.Children.Add(rt3);
}
```

### What You've Accomplished

The text appears in two columns on each page of the document:



## Customizing the File Formats in the Save As Dialog Box

To customize the file formats in the **Save As** dialog box to only save files in a particular file format (for example, Adobe PDF), all of the other file formats except for PDF must be disabled using the ExportOptions property.

To save to a file format other than Adobe PDF (.pdf), replace the "PdfExportProvider" text in the following code to one of the following options:

| File Format | Export Provider |
|---|---|
| BMP Image (.bmp) | BmpExportProvider |

| | |
|---|---|
| C1 Document (.c1d) | C1dExportProvider |
| Enhanced Metafile (.emf) | EmfExportProvider |
| GIF Image (.gif) | GifExportProvider |
| HTML (.htm) | HtmlExportProvider |
| JPEG Image (.jpg) | JpegExportProvider |
| Microsoft Excel (.xls) | XlsExportProvider |
| Open XML MS Excel File (.xlsx) | XslsExportProvider |
| PNG Image (.png) | PngExportProvider |
| Rich Text Format (.rtf) | RtfExportProvider |
| TIFF Image (.tiff) | TiffExportProvider |

Add the following code to the **Form_Load** event:

- Visual Basic

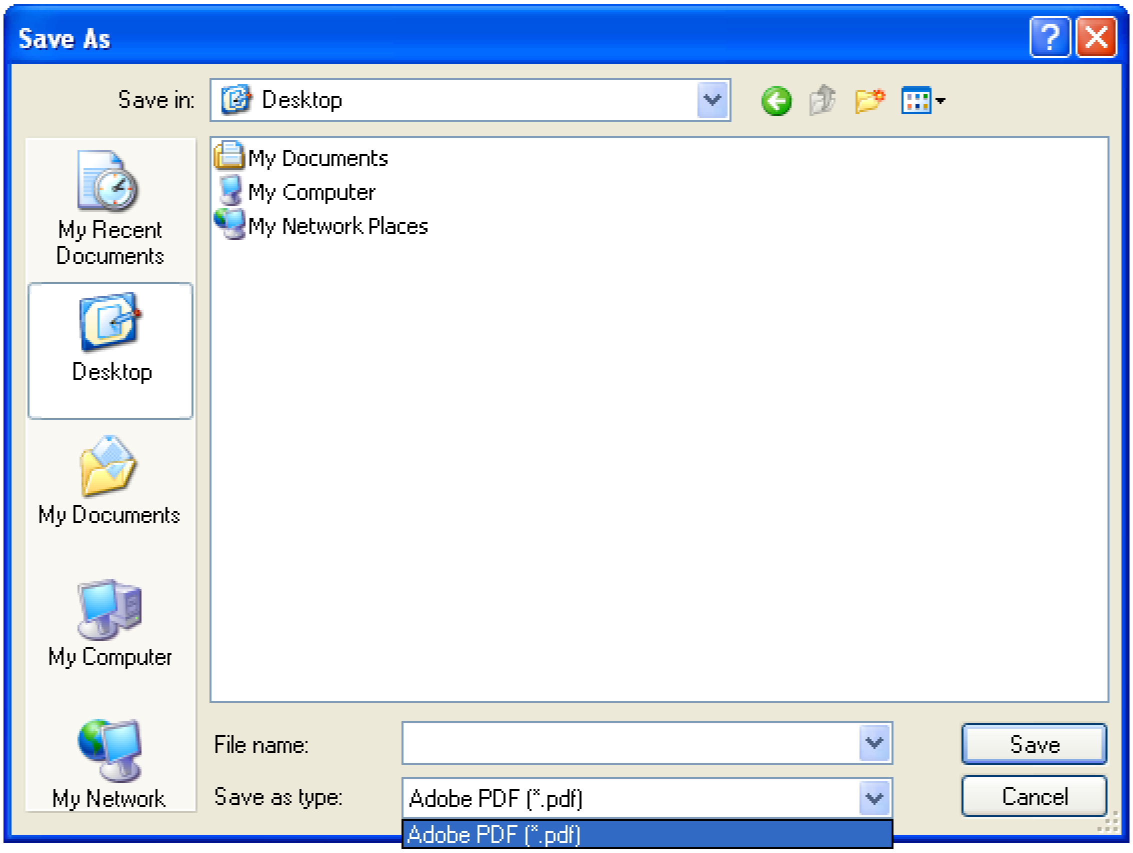
```
Dim lp As Integer = 0
While lp < Me.C1PrintPreviewControl1.ExportOptions.Count
    If Not TypeOf
(C1PrintPreviewControl1.ExportOptions(lp).ExportProvider) Is
C1.C1Preview.Export.PdfExportProvider Then
        C1PrintPreviewControl1.ExportOptions(lp).Enabled = False
    End If
    lp = lp + 1
End While
```

- C#

```
for (int lp = 0; lp < c1PrintPreviewControl1.ExportOptions.Count; lp++)
{
    if (!(c1PrintPreviewControl1.ExportOptions[lp].ExportProvider is
C1.C1Preview.Export.PdfExportProvider))
    {
        c1PrintPreviewControl1.ExportOptions[lp].Enabled = false;
    }
}
```

### ✓ What You've Accomplished

The only available file format in the **Save As** dialog box is Adobe PDF:



Save as type: Adobe PDF (*.pdf)

## Drawing Text at an Angle

To draw text at an angle, use the Graphics object and create a subroutine to rotate text.

1. Navigate to the Toolbox and add the **C1PrintPreviewControl** and **C1PrintDocument** controls to your project.
2. Click **C1PrintPreviewControl1** to select it and in the Properties window set its **Document** property to **C1PrintDocument1**.
3. Add the following code to the **Form_Load** event:
   - Visual Basic

```
Me.C1PrintDocument1.StartDoc()
Me.C1PrintDocument1.RenderBlockGraphicsBegin()

' Declare the graphics object.
Dim g As System.Drawing.Graphics
g = Me.C1PrintDocument1.CurrentBlockGraphics

Dim fontb = New Font("Arial", 12, FontStyle.Bold)

' Subroutine to alter text angle.
RotateText(g, fontb, "Hello World", -45, Brushes.CadetBlue, 10, 100)

Me.C1PrintDocument1.RenderBlockGraphicsEnd()
Me.C1PrintDocument1.EndDoc()
```

   - C#

```
this.c1PrintDocument1.StartDoc();
this.c1PrintDocument1.RenderBlockGraphicsBegin();

// Declare the graphics object.
System.Drawing.Graphics g;
g = this.c1PrintDocument1.CurrentBlockGraphics;

Font fontb = new Font("Arial", 12, FontStyle.Bold);

// Subroutine to alter text angle.
RotateText(g, fontb, "Hello World", -45, Brushes.CadetBlue, 10, 100);

this.c1PrintDocument1.RenderBlockGraphicsEnd();
this.c1PrintDocument1.EndDoc();
```

4. Add the following **RotateText** subroutine, which draws the text at an angle:

- Visual Basic

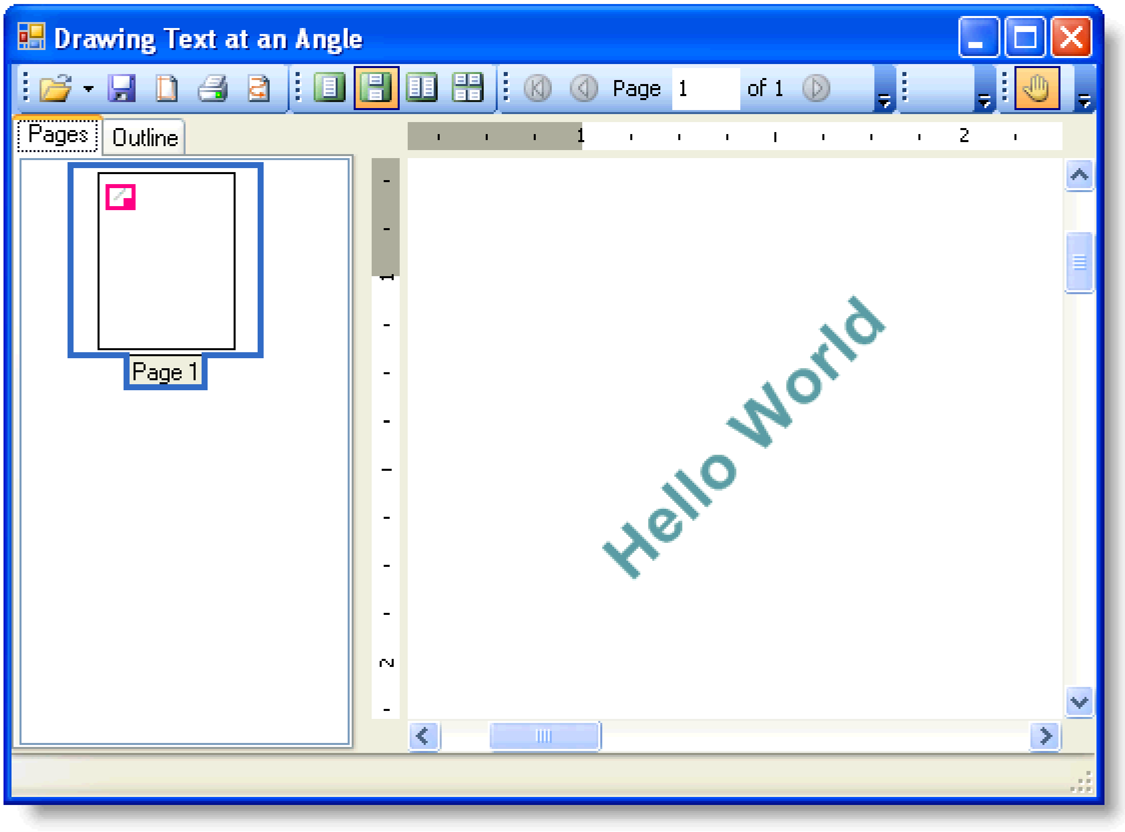
```
Public Sub RotateText(ByVal g As Graphics, ByVal f As Font, ByVal s As
String, ByVal angle As Single, ByVal b As Brush, ByVal x As Single,
ByVal y As Single)
    If angle > 360 Then
        While angle > 360
            angle = angle - 360
        End While
    ElseIf angle < 0 Then
        While angle < 0
            angle = angle + 360
        End While
    End If

    ' Create a matrix and rotate it n degrees.
    Dim myMatrix As New System.Drawing.Drawing2D.Matrix
    myMatrix.Rotate(angle, Drawing2D.MatrixOrder.Append)

    ' Draw the text to the screen after applying the transform.
    g.Transform = myMatrix
    g.DrawString(s, f, b, x, y)
End Sub
```

- C#

```
public void RotateText(Graphics g, Font f, string s, Single angle,
Brush b, Single x, Single y)
{
    if (angle > 360)
    {
        while (angle > 360)
        {
            angle = angle - 360;
        }
    }
    else if (angle < 0)
    {
        while (angle < 0)
        {
            angle = angle + 360;
        }
    }
    // Create a matrix and rotate it n degrees.
    System.Drawing.Drawing2D.Matrix myMatrix = new
System.Drawing.Drawing2D.Matrix();
    myMatrix.Rotate(angle,
System.Drawing.Drawing2D.MatrixOrder.Append);

    // Draw the text to the screen after applying the transform.
    g.Transform = myMatrix;
    g.DrawString(s, f, b, x, y);
}
```

✓ **What You've Accomplished**

The text you added appears at a 45 degree angle:



## Formatting Tables

The following task-based help topics customize features of a table. Each topic assumes that you have already created a table. For details on how to create a table, see Creating a Table with Three Columns and Rows.

### *Changing the Background Color of Rows and Columns*

To change the background color of rows and columns in a table, set the BackColor property of the row and column.

**Note:** The column BackColor property overrides the row BackColor property.

Add the following code to the **Form_Load** event before the **Add** and Generate methods:

- Visual Basic

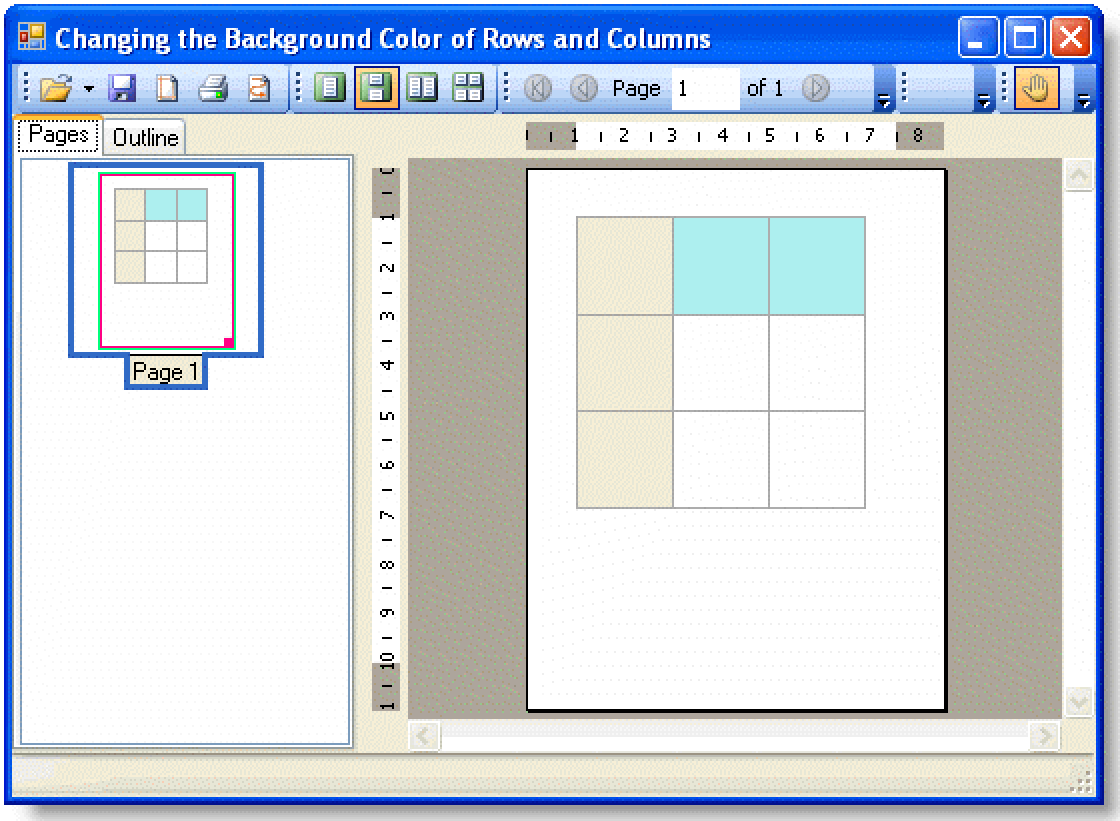
```
' Set the Column color.
table.Cols(0).Style.BackColor = Color.PapayaWhip

' Set the Row color.
table.Rows(0).Style.BackColor = Color.PaleTurquoise
```

- C#

```
// Set the Column color.
table.Cols[0].Style.BackColor = Color.PapayaWhip;

// Set the Row color.
table.Rows[0].Style.BackColor = Color.PaleTurquoise;
```

### ✓ What You've Accomplished

The first column in the table is PapayaWhip and the first row in the table is PaleTurquoise:



## *Changing the Font in a Single Table Cell*

To change the font in a single table cell, set the **Font** property for the cell. Add the following code to the **Form_Load** event before the **Add** and Generate methods:

- Visual Basic

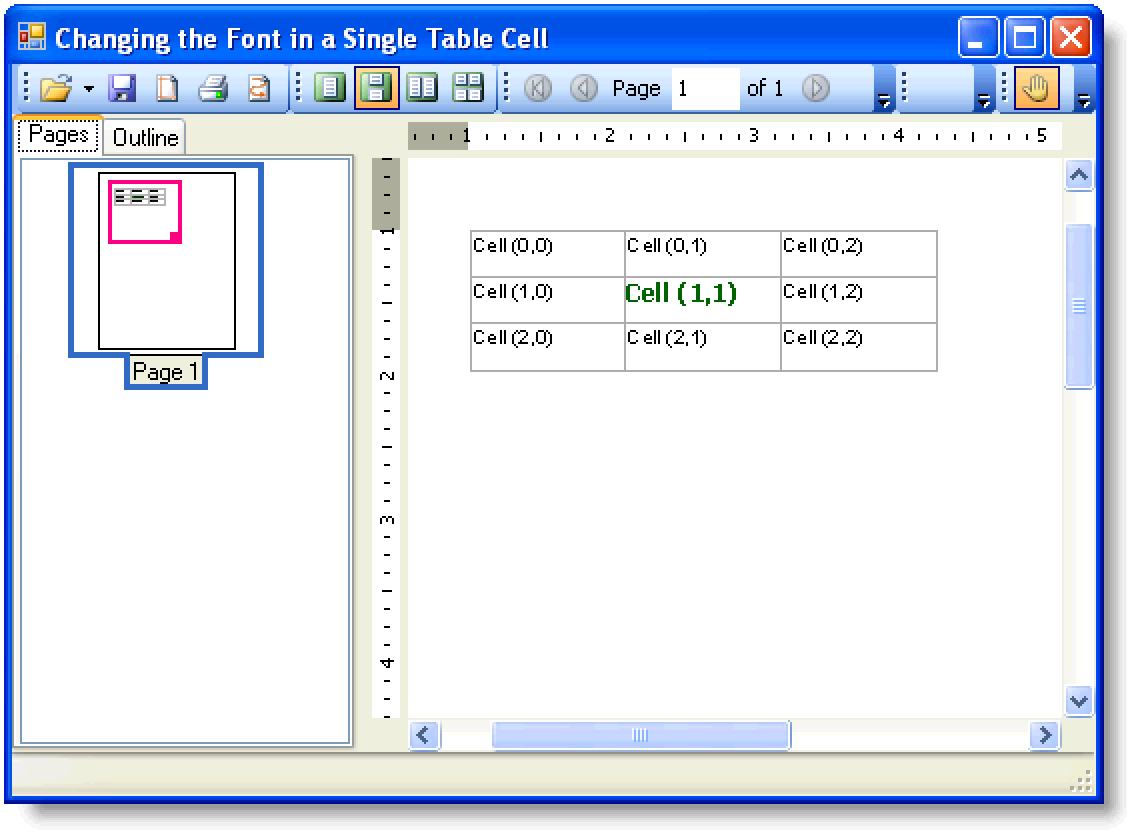
```
table.Cells(1, 1).Style.Font = New Font("Tahoma", 12, FontStyle.Bold)
table.Cells(1, 1).Style.TextColor = Color.DarkGreen
```

- C#

```
table.Cells[1, 1].Style.Font = new Font("Tahoma", 12, FontStyle.Bold);
table.Cells[1, 1].Style.TextColor = Color.DarkGreen;
```

✓ **What You've Accomplished**

The text in cell (1,1) appears in 12 point, Dark Green, Bold Tahoma font:



| | | |
|---|---|---|
| Cell (0,0) | Cell (0,1) | Cell (0,2) |
| Cell (1,0) | Cell (1,1) | Cell (1,2) |
| Cell (2,0) | Cell (2,1) | Cell (2,2) |

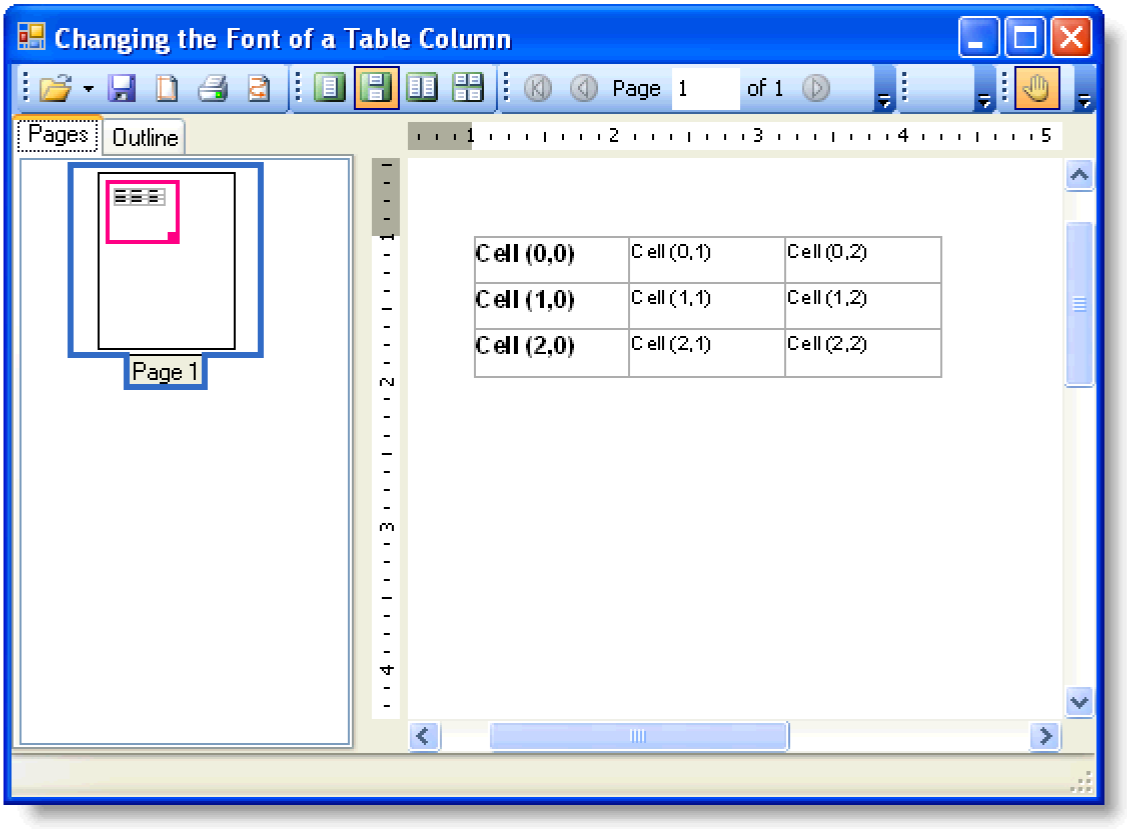
### *Changing the Font of a Table Column*

To change the font of a table column, set the Font property for the column. Add the following code to the **Form_Load** event before the **Add** and Generate methods:

- Visual Basic
```
table.Cols(0).Style.Font = New Font("Arial", 12, FontStyle.Bold)
```
- C#
```
table.Cols[0].Style.Font = new Font("Arial", 12, FontStyle.Bold);
```

✓ **What You've Accomplished**

The text in the first column appears in 12 point, Bold Arial font:



|  |  |  |
|---|---|---|
| **Cell (0,0)** | Cell (0,1) | Cell (0,2) |
| **Cell (1,0)** | Cell (1,1) | Cell (1,2) |
| **Cell (2,0)** | Cell (2,1) | Cell (2,2) |

### *Changing the Font of a Table Row*

To change the font of a table row, set the Font property for the row. Add the following code to the **Form_Load** event before the **Add** and Generate methods:

- Visual Basic

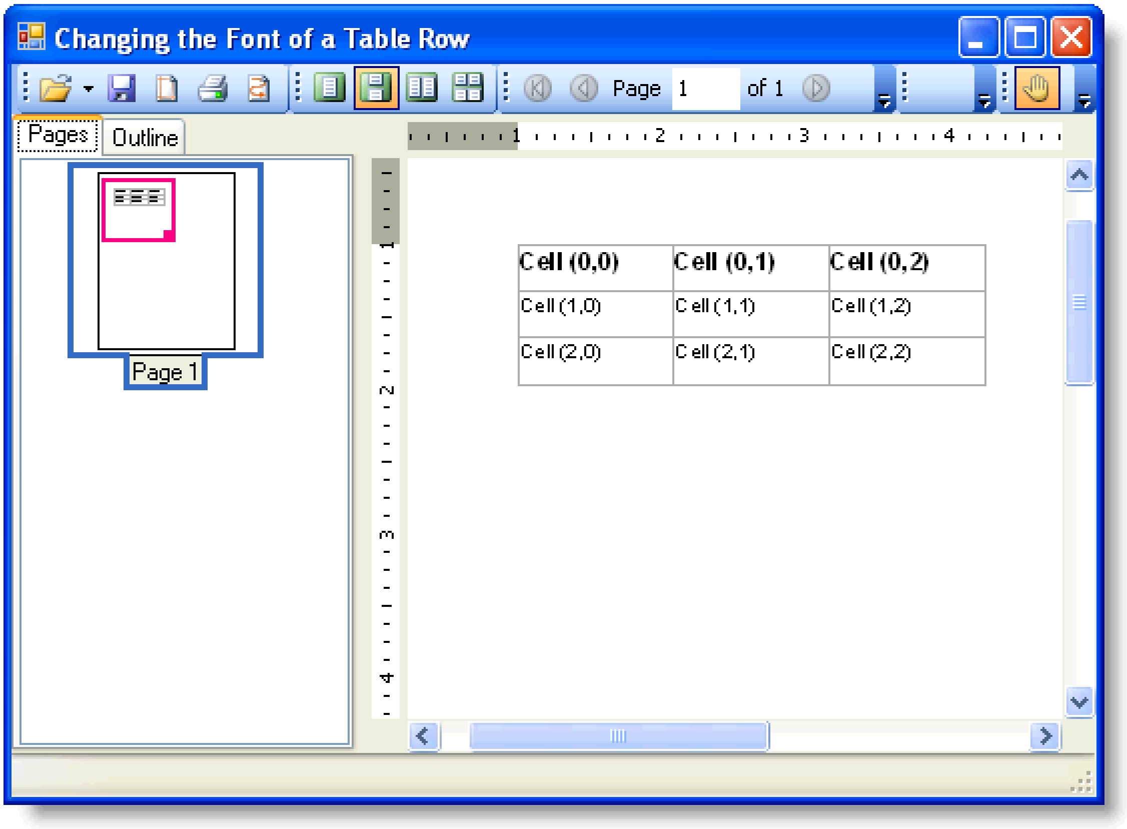
```
table.Rows(0).Style.Font = New Font("Arial", 12, FontStyle.Bold)
```

- C#

```
table.Rows[0].Style.Font = new Font("Arial", 12, FontStyle.Bold);
```

**What You've Accomplished**

The text in the first row appears in 12 point, Bold Arial font:



| Cell (0,0) | Cell (0,1) | Cell (0,2) |
|---|---|---|
| Cell (1,0) | C ell (1,1) | Cell (1,2) |
| Cell (2,0) | C ell (2,1) | Cell (2,2) |

### *Changing the Alignment of a Table*

To change the alignment of a table, set the FlowAlign property for the table. For example, add the following code to the **Form_Load** event before the Generate method to center the table on the page:

- Visual Basic

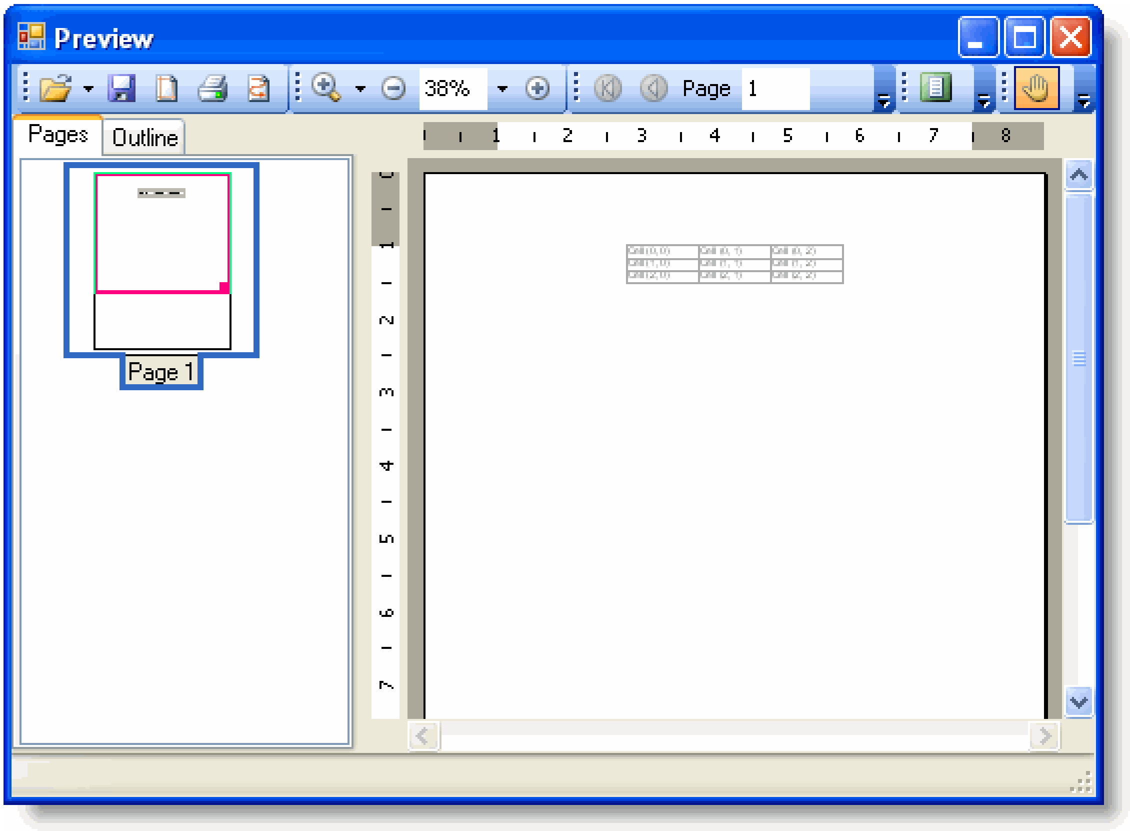
```
table.Style.FlowAlign = FlowAlignEnum.Center
```

- C#

```
table.Style.FlowAlign = FlowAlignEnum.Center;
```

✓ **What You've Accomplished**

The table appears centered on the page:



|  |  |  |
|---|---|---|
| Cell (0, 0) | Cell (0, 1) | Cell (0, 2) |
| Cell (1, 0) | Cell (1, 1) | Cell (1, 2) |
| Cell (2, 0) | Cell (2, 1) | Cell (2, 2) |

### Rendering Overlapping Objects in a Table

You can easily render overlapping objects in a table cell. For example, add the following code to the **Form_Load** event before the Generate method to add a square overlapped by two crossing lines in the table:

- Visual Basic

```
' Create a rectangle, and two lines to form an 'X'.
Dim rect As New RenderRectangle(New Unit(3, UnitTypeEnum.Cm), New Unit(3,
UnitTypeEnum.Cm))
Dim rl1 As New RenderLine(New Unit(0, UnitTypeEnum.Cm), New Unit(0,
UnitTypeEnum.Cm), New Unit(3, UnitTypeEnum.Cm), New Unit(3,
UnitTypeEnum.Cm), LineDef.[Default])
Dim rl2 As New RenderLine(New Unit(3, UnitTypeEnum.Cm), New Unit(0,
UnitTypeEnum.Cm), New Unit(0, UnitTypeEnum.Cm), New Unit(3,
UnitTypeEnum.Cm), LineDef.[Default])
rect.Style.BackColor = Color.PeachPuff

' Add the objects to the table.
table.Cells(1, 1).Area.Children.Add(rect)
table.Cells(1, 1).Area.Children.Add(rl1)
table.Cells(1, 1).Area.Children.Add(rl2)
```

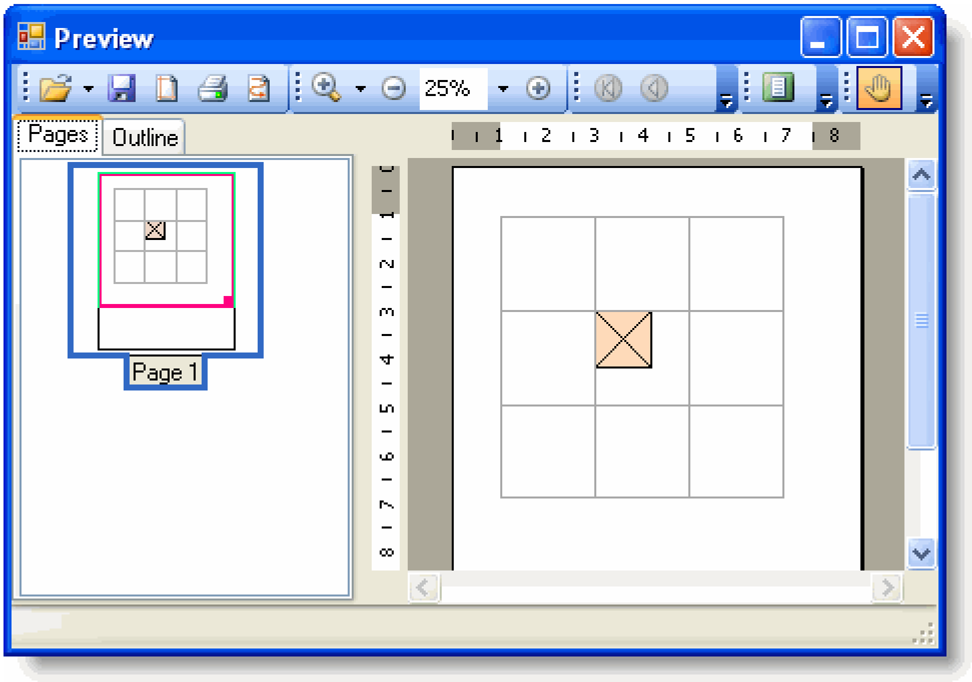
- C#

```
// Create a rectangle, and two lines to form an 'X'.
RenderRectangle rect = new RenderRectangle(new Unit(3, UnitTypeEnum.Cm),
new Unit(3, UnitTypeEnum.Cm));
RenderLine rl1 = new RenderLine(new Unit(0, UnitTypeEnum.Cm), new Unit(0,
UnitTypeEnum.Cm), new Unit(3, UnitTypeEnum.Cm), new Unit(3,
UnitTypeEnum.Cm), LineDef.Default);
```

```
RenderLine rl2 = new RenderLine(new Unit(3, UnitTypeEnum.Cm), new Unit(0,
UnitTypeEnum.Cm), new Unit(0, UnitTypeEnum.Cm), new Unit(3,
UnitTypeEnum.Cm), LineDef.Default);
rect.Style.BackColor = Color.PeachPuff;

// Add the objects to the table.
table.Cells[1, 1].Area.Children.Add(rect);
table.Cells[1, 1].Area.Children.Add(rl1);
table.Cells[1, 1].Area.Children.Add(rl2);
```

**What You've Accomplished**

The table appears with a square with two overlapping lines:



### *Inserting a Page Break*

To insert a page break, use the BreakAfter property for the RenderObject.

1. From the Toolbox, add the **C1PrintPreviewControl** and **C1PrintDocument** controls to your project.
2. Click **C1PrintPreviewControl1** to select it and in the Properties window set its **Document** property to **C1PrintDocument1**.
3. Add the following code to the **Form_Load** event:
   - Visual Basic

```
' Make the document.
MakeDoc()

' Generate the document.
Me.C1PrintDocument1.Generate()
```

   - C#

```
// Make the document.
MakeDoc();
```

```
// Generate the document.
this.c1PrintDocument1.Generate();
```

4. Add the MakeDoc subroutine, which uses the BreakAfter property to insert a page break after each RenderObject:

   - Visual Basic

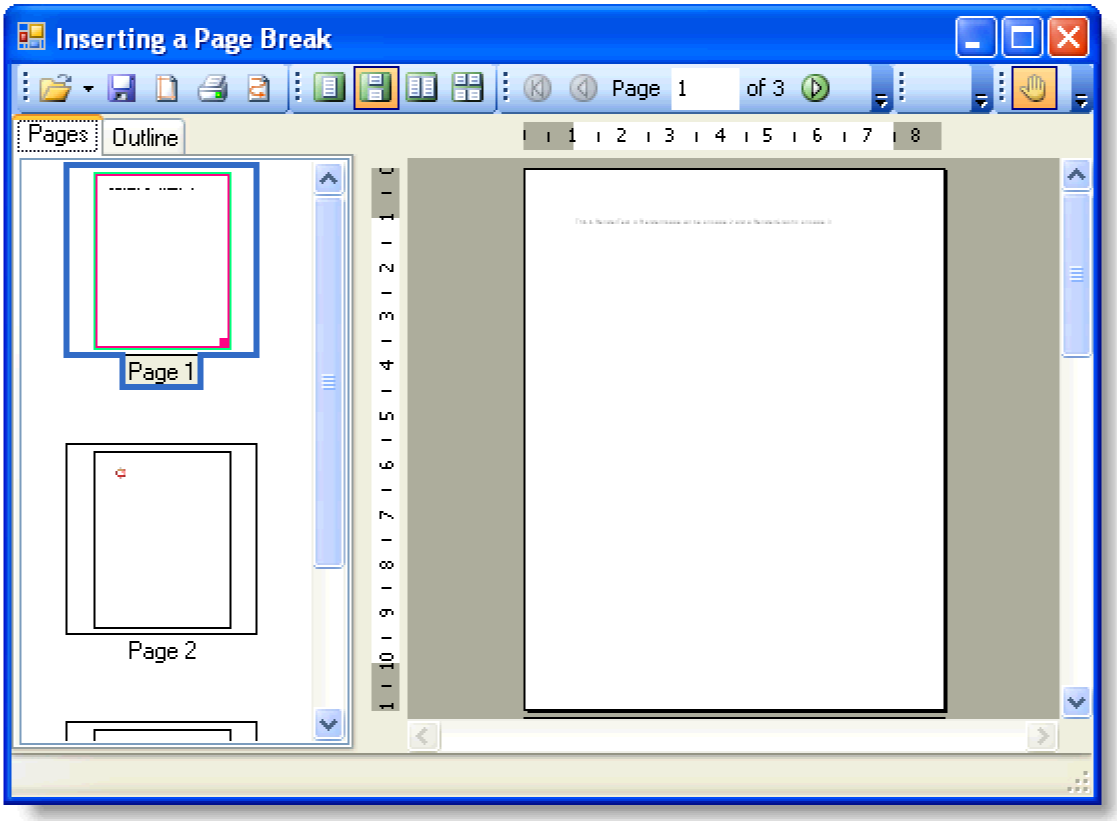
```
Private Sub MakeDoc()

    ' Create RenderText.
    Dim rt1 As New C1.C1Preview.RenderText
    rt1.Text = "This is RenderText. A RenderImage will be on page 2 and
a RenderGraphic on page 3."

    ' Add a page break.
    rt1.BreakAfter = C1.C1Preview.BreakEnum.Page

    ' Create RenderImage.
    Dim ri1 As New C1.C1Preview.RenderImage
    ri1.Image = System.Drawing.Image.FromFile("c:\c1logo.bmp")

    ' Add a page break.
    ri1.BreakAfter = C1.C1Preview.BreakEnum.Page

    ' Create a RenderGraphic.
    Dim rg1 As New C1.C1Preview.RenderGraphics()
    rg1.Graphics.FillEllipse(Brushes.DarkBlue, 200, 200, 150, 150)
    rg1.Graphics.FillPie(Brushes.DarkRed, 200, 200, 150, 150, -45, 75)

    ' Add the RenderObjects to the document.
    Me.C1PrintDocument1.Body.Children.Add(rt1)
    Me.C1PrintDocument1.Body.Children.Add(ri1)
    Me.C1PrintDocument1.Body.Children.Add(rg1)
End Sub
```

   - C#

```
private void MakeDoc()
{
    // Create RenderText.
    C1.C1Preview.RenderText rt1 = new C1.C1Preview.RenderText();
    rt1.Text = "This is RenderText. A RenderImage will be on page 2 and
a RenderGraphic on page 3.";

    // Add a page break.
    rt1.BreakAfter = C1.C1Preview.BreakEnum.Page;

    // Create RenderImage.
    C1.C1Preview.RenderImage ri1 = new C1.C1Preview.RenderImage();
    ri1.Image = System.Drawing.Image.FromFile("c:\\c1logo.bmp");

    // Add a page break.
    ri1.BreakAfter = C1.C1Preview.BreakEnum.Page;

    // Create a RenderGraphic.
    C1.C1Preview.RenderGraphics rg1 = new
C1.C1Preview.RenderGraphics();
    rg1.Graphics.FillEllipse(Brushes.DarkBlue, 200, 200, 150, 150);
    rg1.Graphics.FillPie(Brushes.DarkRed, 200, 200, 150, 150, -45, 75);

    // Add the RenderObjects to the document.
    this.c1PrintDocument1.Body.Children.Add(rt1);
    this.c1PrintDocument1.Body.Children.Add(ri1);
    this.c1PrintDocument1.Body.Children.Add(rg1);
}
```

✓ **What You've Accomplished**

A page break is inserted between the different RenderObjects:



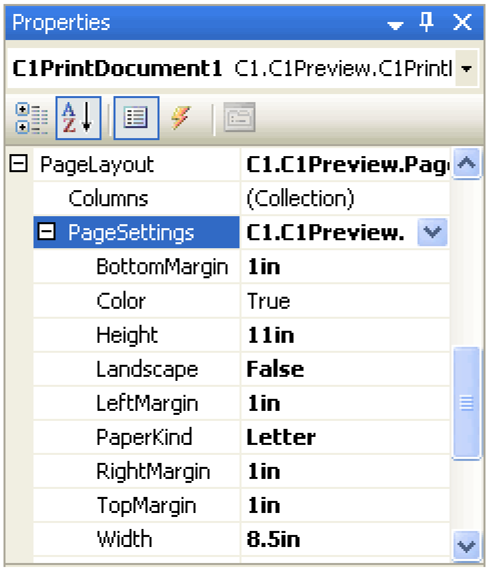
## *Printing in Landscape*

C1PrintDocument allows to you to specify different printing options and page settings for your document. The page settings can be specified in the designer or in code.

**In the Designer**

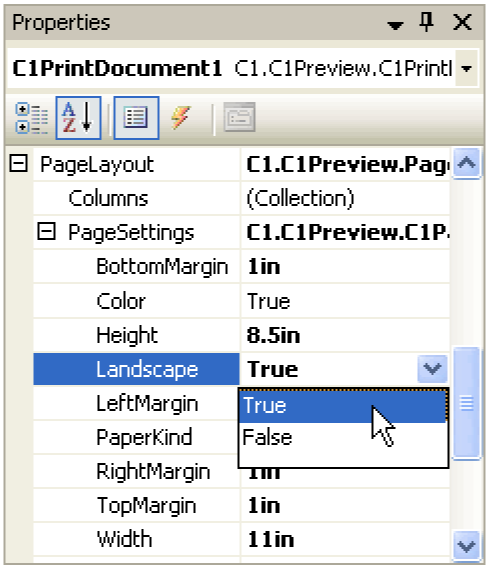
1. Select the **C1PrintDocument** control on your form. Its properties will appear in the right-pane of the Visual Studio interface.

2. Locate the **PageSettings** property for the C1PrintDocument.



| | |
|---|---|
| ⊟ PageLayout | C1.C1Preview.Pag |
| Columns | (Collection) |
| ⊟ PageSettings | C1.C1Preview. |
| BottomMargin | 1in |
| Color | True |
| Height | 11in |
| Landscape | False |
| LeftMargin | 1in |
| PaperKind | Letter |
| RightMargin | 1in |
| TopMargin | 1in |
| Width | 8.5in |

3. Locate the **Landscape** property and set it to **True**. Your C1PrintDocument will appear and print in the Landscape orientation.



| | |
|---|---|
| ⊟ PageLayout | C1.C1Preview.Pag |
| Columns | (Collection) |
| ⊟ PageSettings | C1.C1Preview.C1P. |
| BottomMargin | 1in |
| Color | True |
| Height | 8.5in |
| Landscape | True |
| LeftMargin | |
| PaperKind | |
| RightMargin | 1in |
| TopMargin | 1in |
| Width | 11in |

### In Code

You can also change your page settings for your C1PrintDocument at run time, by adding the following code to the **Form_Load** event:

- Visual Basic

```
Me.C1PrintDocument1.PageLayout.PageSettings.Landscape = True
```

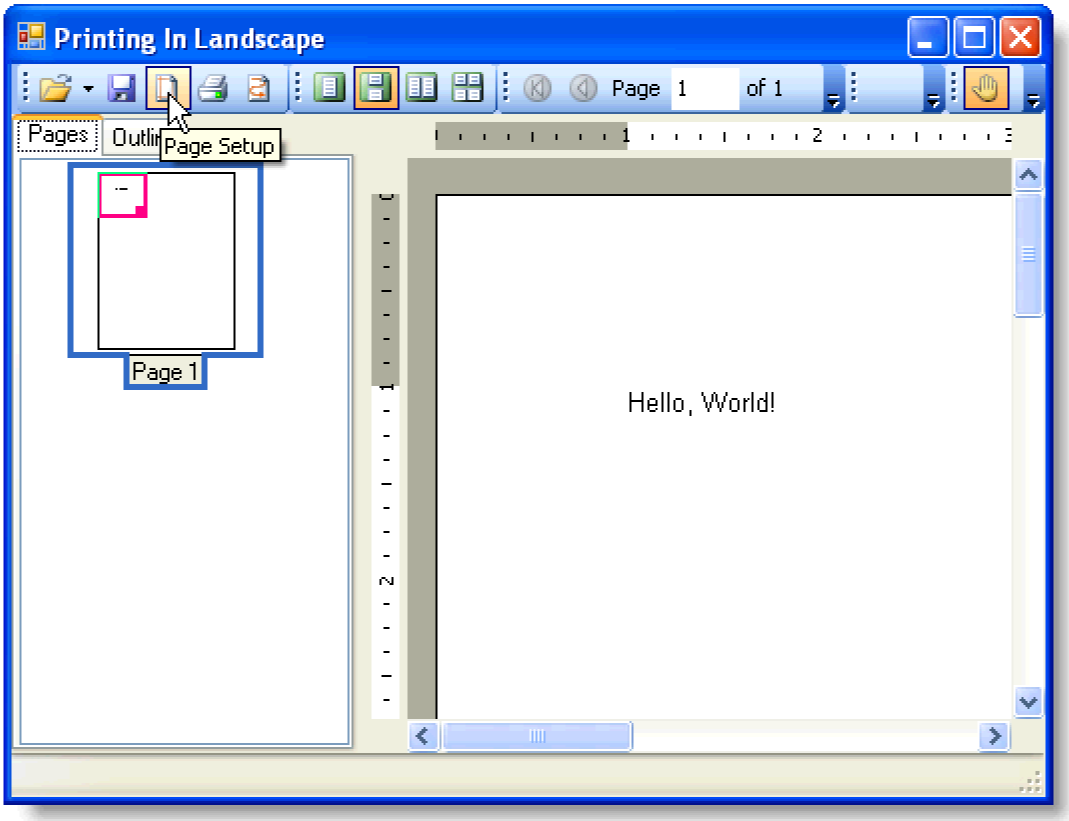
- C#

```
this.c1PrintDocument1.PageLayout.PageSettings.Landscape = true;
```

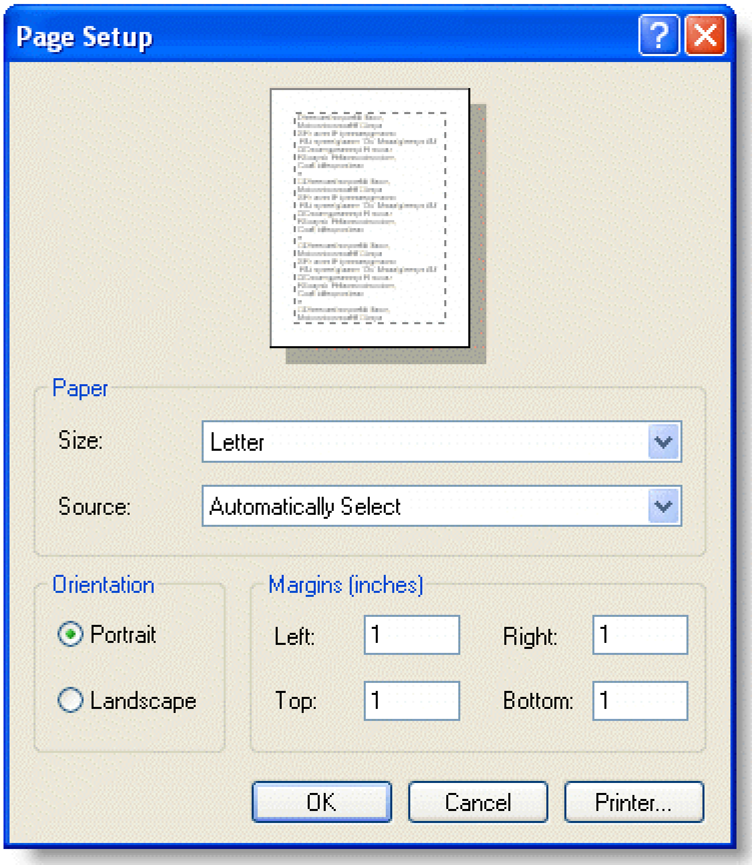
### At Run Time

You can also change the page settings for your C1PrintDocument in the preview at run time using the **Page Setup** dialog box.

1. Click the **Page Setup** button located in the toolbar.



The **Page Setup** dialog box opens.



Size: Letter

Source: Automatically Select

● Portrait

○ Landscape

Left: 1 Right: 1

Top: 1 Bottom: 1

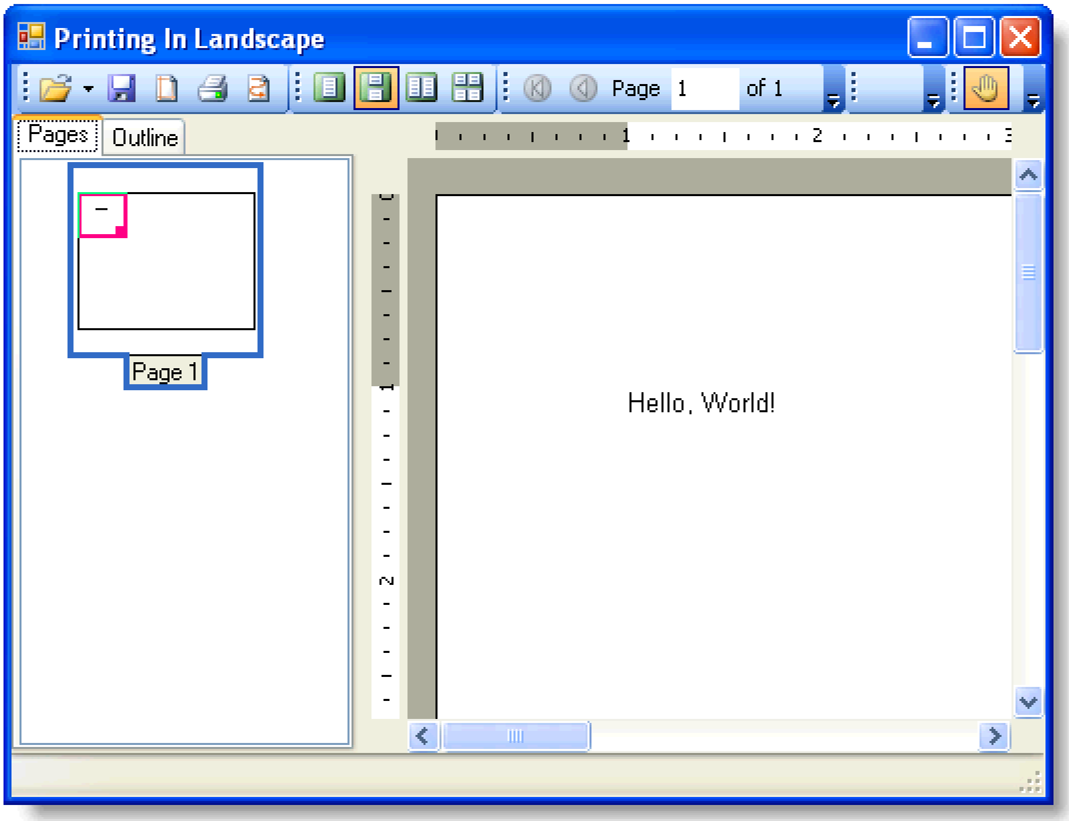
The **Page Setup** dialog box contains the following fields:

| Field Name | Description |
|---|---|
| Paper | Select the size and the source of the paper. The available sizes and sources depend on the selected printer. |
| Orientation | Choose from **Portrait** or **Landscape** page orientation. |
| Margins | Customize the Left, Right, Top, and Bottom margins of your C1PrintDocument. |

The **Page Setup** dialog box has the following Command Buttons:

| Button Name | Description |
|---|---|
| **OK** | Apply the settings to your C1PrintDocument. |
| **Cancel** | Cancel the modified Page Settings for your document. |
| **Printer** | Modify or view the settings for your printer. |

2. Locate the **Orientation** field and select the **Landscape** option. Your C1PrintDocument now appears and will print in the Landscape orientation.



Hello, World!

### *Setting the Page Size*

To set the page size for the document, use the PaperKind property.

1. From the Toolbox, add the **C1PrintPreviewControl** and **C1PrintDocument** controls to your project.
2. Click **C1PrintPreviewControl1** to select it and in the Properties window set its **Document** property to **C1PrintDocument1**.
3. Add the following code to the **Form_Load** event:
   - Visual Basic
     ```
     ' Make the document.
     MakeDoc()

     ' Generate the document.
     Me.C1PrintDocument1.Generate()
     ```
   - C#
     ```
     // Make the document.
     MakeDoc();

     // Generate the document.
     this.c1PrintDocument1.Generate();
     ```
4. Add the following MakeDoc subroutine, which uses the PaperKind property to set the page size to **Legal**:
   - Visual Basic
     ```
     Private Sub MakeDoc()
     ```

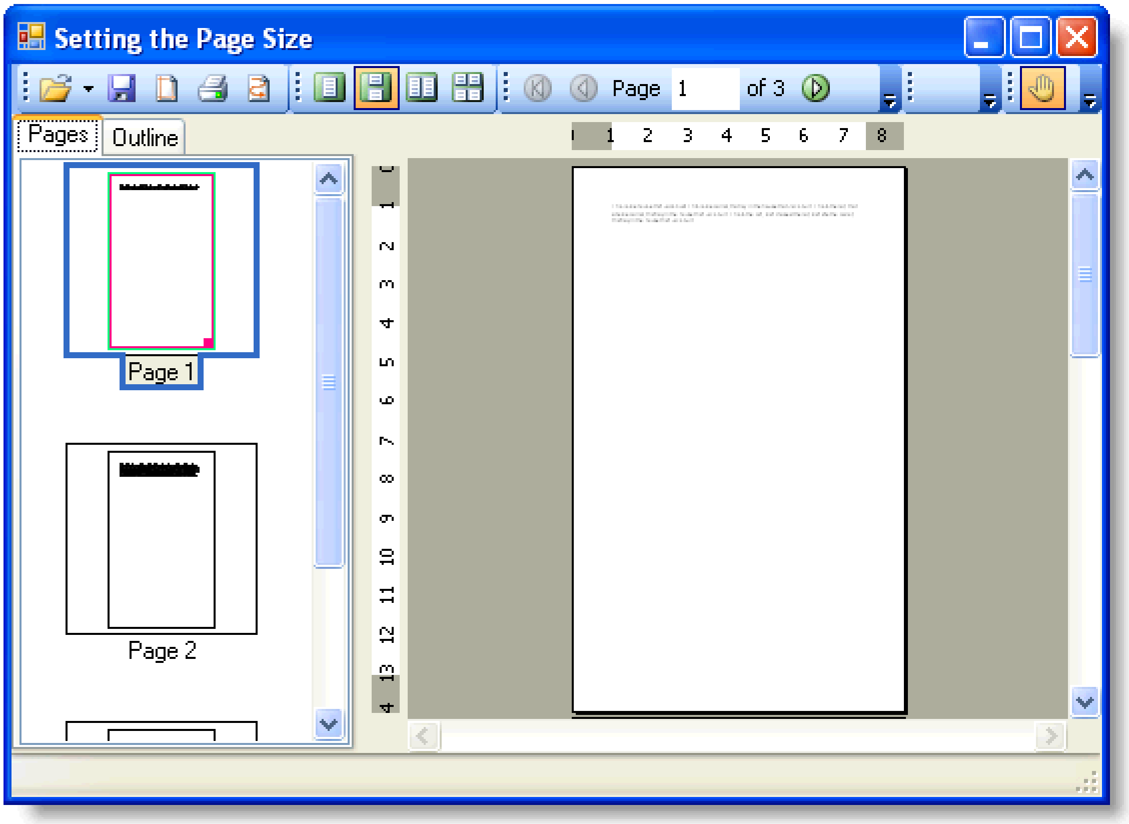
```
    ' Define the page layout for the document.
    Dim pl As New C1.C1Preview.PageLayout()
    pl.PageSettings = New C1.C1Preview.C1PageSettings()
    pl.PageSettings.PaperKind = System.Drawing.Printing.PaperKind.Legal
    Me.C1PrintDocument1.PageLayouts.Default = pl
End Sub
```

- C#

```
private void MakeDoc()
{

    // Define the page layout for the document.
    C1.C1Preview.PageLayout pl = new C1.C1Preview.PageLayout();
    pl.PageSettings = new C1.C1Preview.C1PageSettings();
    pl.PageSettings.PaperKind =
System.Drawing.Printing.PaperKind.Legal;
    this.c1PrintDocument1.PageLayouts.Default = pl;
}
```

### What You've Accomplished

The default page size is set to **Legal**:



## *Resizing or Scaling an Image*

You can easily resize or scale a RenderImage to 50% by completing the following steps:

1. From the Toolbox, add the **C1PrintPreviewControl** and **C1PrintDocument** components to your project.

2. Click **C1PrintPreviewControl1** to select it, and in the Properties window set its **Document** property to **C1PrintDocument1**.

3. Switch to Code view and add the following namespace declaration:

   - Visual Basic

   ```
   Imports C1.C1Preview
   ```

   - C#

   ```
   using C1.C1Preview;
   ```

4. Add the following **Form_Load** event, which adds an image to the page and scales that image to 50% of the available width with the height scaling automatically, replacing *c1logo.png* with your image name and location:

   - Visual Basic

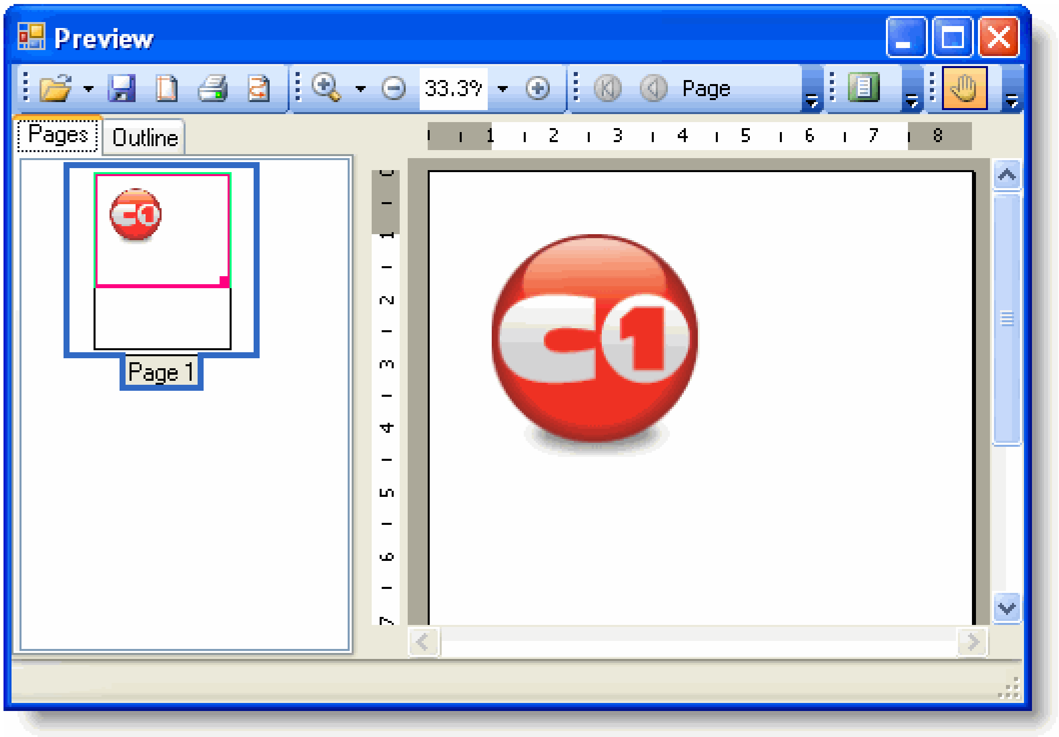
   ```
   Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
   System.EventArgs) Handles MyBase.Load
       ' Create a new Render Image, replacing c1logo.png with your image
   name and location.
       Dim img As New C1.C1Preview.RenderImage
       img.Image = Image.FromFile("C:\c1logo.png")
       ' Scale the image to 50% of the available page width, the height of
   the image will scale automatically here.
       img.Width = "50%"
       ' Create the document.
       C1PrintDocument1.StartDoc()
       C1PrintDocument1.RenderBlock(img)
       C1PrintDocument1.EndDoc()
   End Sub
   ```

   - C#

   ```
   private void Form1_Load(object sender, EventArgs e)
   {
       // Create a new Render Image, replacing c1logo.png with your image
   name and location.
       C1.C1Preview.RenderImage img = new C1.C1Preview.RenderImage();
       img.Image = Image.FromFile("C:\\c1logo.png");
       // Scale the image to 50% of the available page width, the height
   of the image will scale automatically here.
       img.Width = "50%";
       // Create the document.
       c1PrintDocument1.StartDoc();
       c1PrintDocument1.RenderBlock(img);
       c1PrintDocument1.EndDoc();
   }
   ```

### ✓ What You've Accomplished

A image appears scaled on the page:



### Adding a Watermark Image

To set the page size for the document, use the Watermark property.

1. From the Toolbox, add the **C1PrintPreviewControl** and **C1PrintDocument** components to your project.
2. Click **C1PrintPreviewControl1** to select it, and in the Properties window set its **Document** property to **C1PrintDocument1**.
3. Switch to Code view and add the following namespace declaration:
   - Visual Basic
     ```
     Imports C1.C1Preview
     ```
   - C#
     ```
     using C1.C1Preview;
     ```
4. Add the following **Form_Load** event, which uses the Watermark property to add a watermark to the page, and replace *c1logo.png* with your image's name and location:
   - Visual Basic
     ```
     Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
     System.EventArgs) Handles MyBase.Load
         ' Create watermark and layout.
         Dim waterMark As New RenderImage
         Dim pl As New C1.C1Preview.PageLayout()

         ' Set the watermark image; replace c1logo.png with your image's
     name.
         waterMark.Image = Image.FromFile("c:\c1logo.png")
         waterMark.Y = New Unit(2, UnitTypeEnum.Inch)
         pl.Watermark = waterMark
         Me.C1PrintDocument1.PageLayout = pl

         ' Generate the document.
     ```

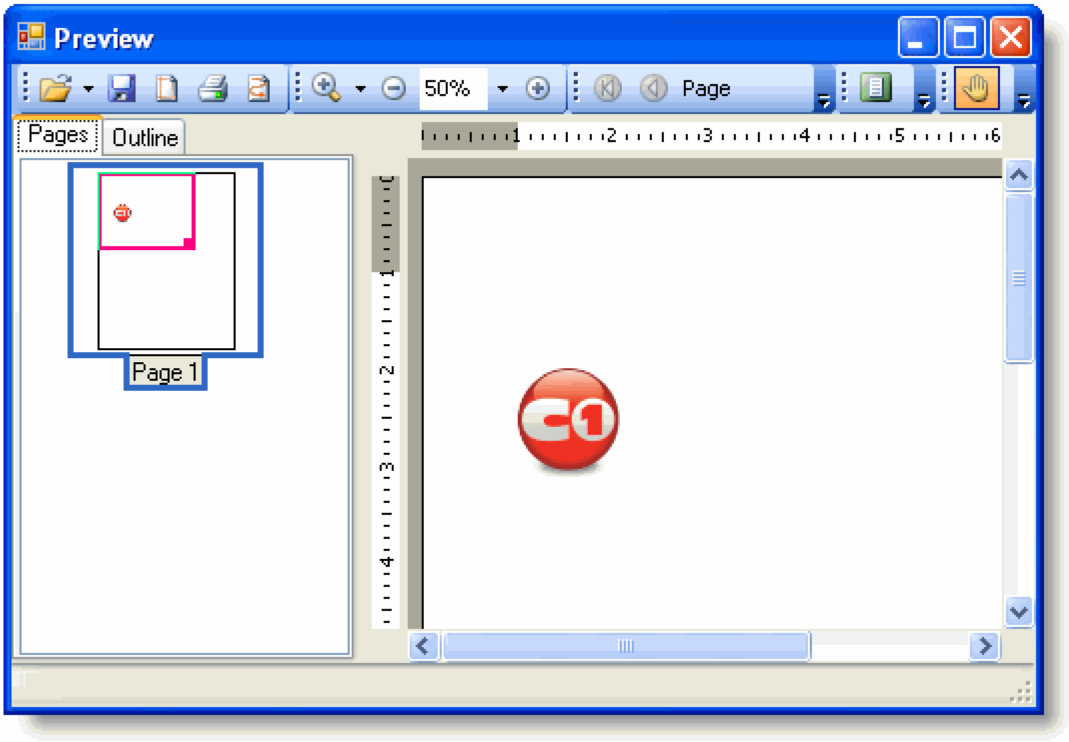
```
    Me.C1PrintDocument1.Generate()
End Sub
```

- C#

```
private void Form1_Load(object sender, EventArgs e)
{
    // Create watermark and layout.
    RenderImage waterMark = new RenderImage();
    C1.C1Preview.PageLayout pl = new C1.C1Preview.PageLayout();

    // Set the watermark image; replace c1logo.png with your image's
name.
    waterMark.Image = Image.FromFile("c:\\c1logo.png");
    waterMark.Y = new Unit(2, UnitTypeEnum.Inch);
    pl.Watermark = waterMark;
    this.c1PrintDocument1.PageLayout = pl;

    // Generate the document.
    this.C1PrintDocument1.Generate();
}
```

### What You've Accomplished

A watermark is added to the page:

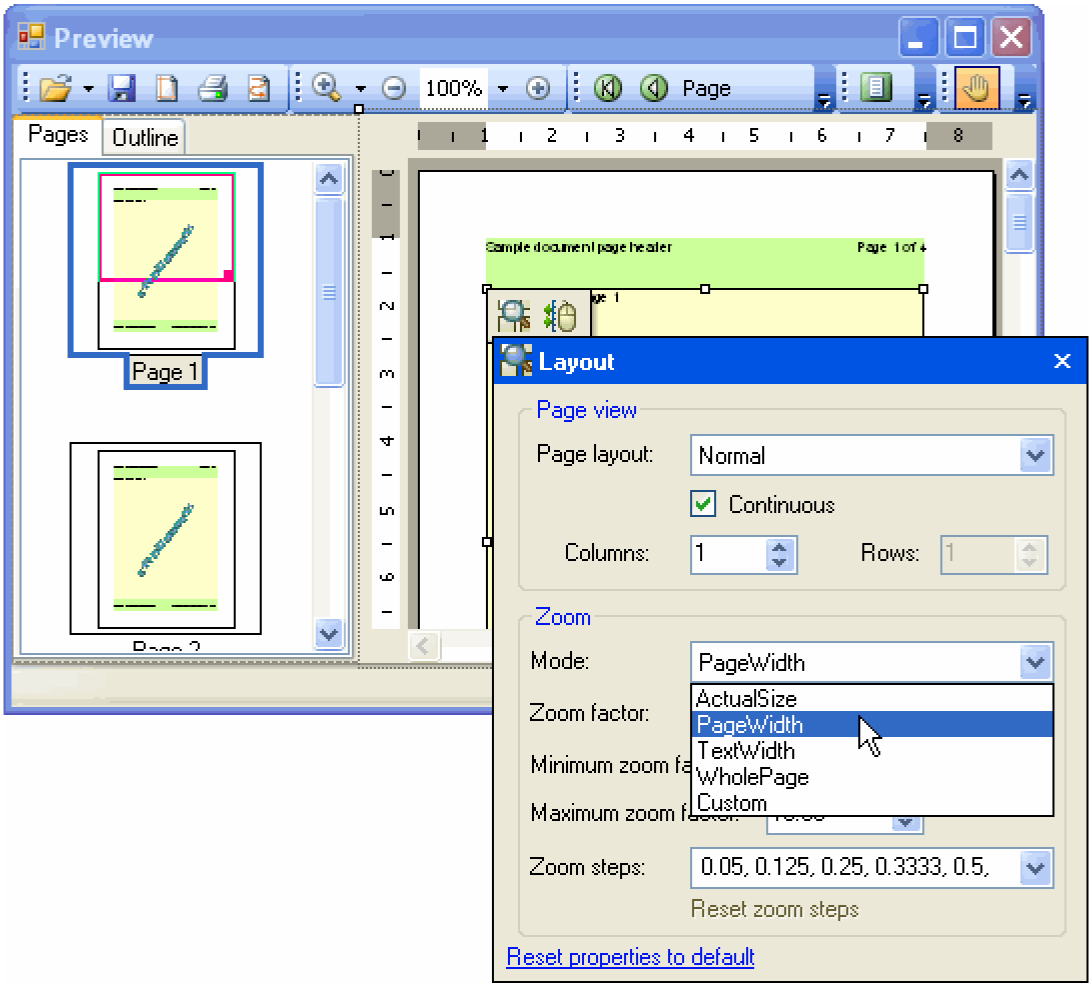


## *Setting the Initial Zoom Mode*

To set the initial zoom mode, set the ZoomMode property to **ActualSize**, **PageWidth**, **TextWidth**, **WholePage** (default), or **Custom**. This property can be set at either in the designer or in code.

### In the Smart Designer

1. Click the **Layout** button in the **PreviewPane** floating toolbar to open **Layout** dialog box.
2. Select **PageWidth** from the drop-down menu next to the Zoom Mode:



Layout

Page view

Page layout: Normal

☑ Continuous

Columns: 1 Rows: 1

Zoom

Mode: PageWidth

ActualSize
PageWidth
TextWidth
WholePage
Custom

Zoom factor:

Minimum zoom fa

Maximum zoom f

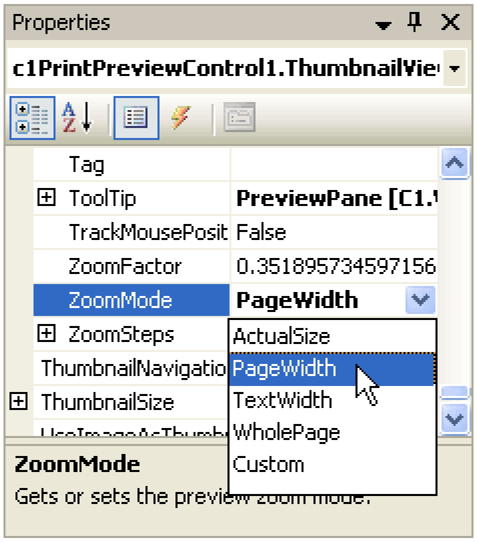
Zoom steps: 0.05, 0.125, 0.25, 0.3333, 0.5,

Reset zoom steps

Reset properties to default

**In the Properties window**

1. Locate the **PreviewPane** property for C1PrintPreviewControl in the Properties window and expand the properties node.
2. Set the **ZoomMode** property to **PageWidth**.



c1PrintPreviewControl1.ThumbnailVie

Tag
ToolTip PreviewPane [C1.\
TrackMousePosit False
ZoomFactor 0.351895734597156
ZoomMode PageWidth
ZoomSteps
ThumbnailNavigatio
ThumbnailSize

ZoomMode
Gets or sets the preview zoom mode.

### In Code

Add the following code to the **Form_Load** event to set the ZoomMode to **PageWidth**:

- Visual Basic

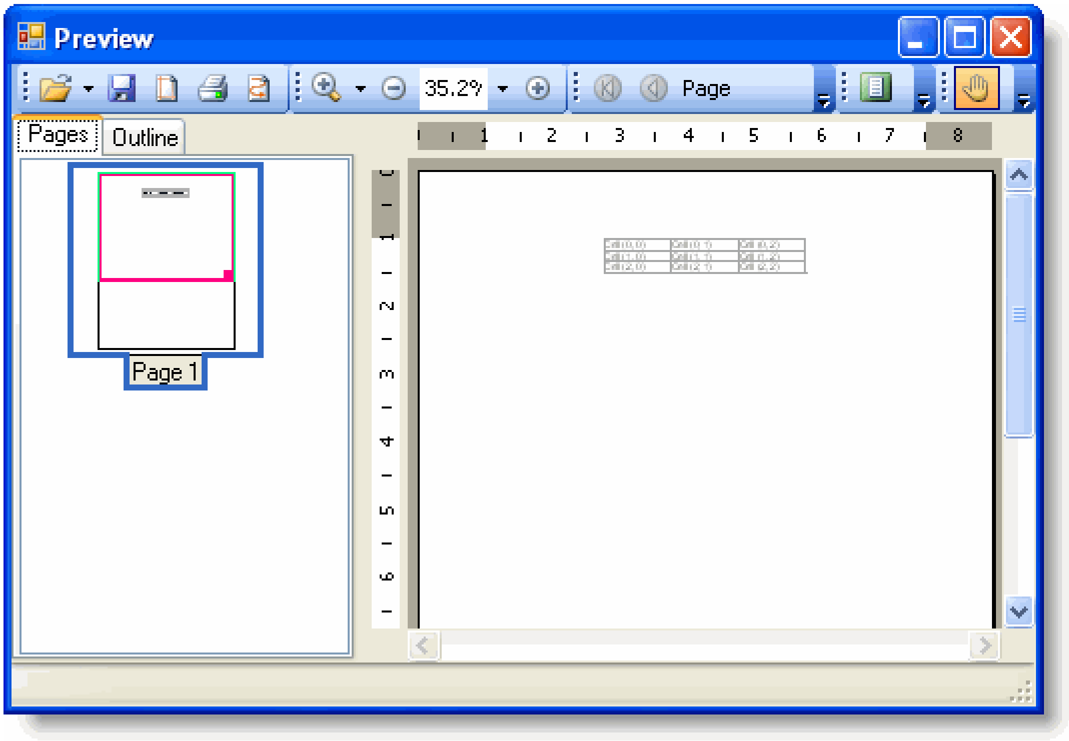
```
Me.C1PrintPreviewControl1.PreviewPane.ZoomMode =
C1.Win.C1Preview.ZoomModeEnum.PageWidth
```

- C#

```
this.c1PrintPreviewControl1.PreviewPane.ZoomMode =
C1.Win.C1Preview.ZoomModeEnum.PageWidth;
```

### What You've Accomplished

When the document loads, the page zooms into the page width:



### Removing an Item from the Context Menu

By default a context menu appears when the C1PreviewPane is right-clicked at run time. This **ContextMenuStrip** includes settings for manipulating the preview including items from the File, Zoom, and Text toobars. You can create your own context menu by adding a **ContextMenuStrip** control and assigning it to the PreviewPane.**ContextMenuStrip** property. You can also customize the existing **ContextMenuStrip** by adding additional items or removing existing items.

The following example removes the standard "Copy" item from the context menu. Complete the following steps:

1. In the **Form_Load** event attach a handler to the **Opening** event of the **ContextMenuStrip** on the C1PreviewPane:

   - Visual Basic

   ```
   AddHandler PreviewPane.ContextMenuStrip.Opening, AddressOf
   ContextMenuStrip_Opening
   ```

   - C#

   ```
   PreviewPane.ContextMenuStrip.Opening += new
   CancelEventHandler(ContextMenuStrip_Opening);
   ```

2. Create the **ContextMenuStrip_Opening** event, and add the following code to remove the "Copy" item from the context menu:

   - Visual Basic

   ```
   Private Sub ContextMenuStrip_Opening(ByVal sender As Object, ByVal e As
   CancelEventArgs)
       Dim cms As System.Windows.Forms.ContextMenuStrip =
   DirectCast(sender, System.Windows.Forms.ContextMenuStrip)
       For Each item As ToolStripItem In cms.Items
           If item.Tag = ContextMenuTags.Copy Then
               item.Visible = False
           End If
       Next
   ```

```
End Sub
```

- C#

```
void ContextMenuStrip_Opening(object sender, CancelEventArgs e)
{
  System.Windows.Forms.ContextMenuStrip cms =
(System.Windows.Forms.ContextMenuStrip)sender;
  foreach (ToolStripItem item in cms.Items)
    if (item.Tag == ContextMenuTags.Copy)
      item.Visible = false;
}
```

### What You've Accomplished

When you right-click the preview pane in the C1PrintPreviewControl control, observe that the standard "Copy" item is not included on the context menu.

## *Disabling the Context Menu*

A context menu appears by default when the C1PreviewPane is right-clicked at run time. This **ContextMenuStrip** includes settings for manipulating the preview. You can disable this context menu in code by setting the **ContextMenuStrip** property of the C1PreviewPane to null. Note that this cannot be done in the designer – only in code – but, you can override the default context menu strip with your own in the designer (by dropping a **ContextMenuStrip** component on the form, and setting the **C1PreviewPane**'s **ContextMenuStrip** property to that component). When using the C1PrintPreviewControl control, the preview pane can be accessed via the PreviewPane property on the control.

Add the following code to the **Form_Load** event to disable this context menu in a C1PrintPreviewControl control:

- Visual Basic

```
Me.C1PrintPreviewControl1.PreviewPane.ContextMenuStrip = Nothing
```

- C#

```
this.c1PrintPreviewControl1.PreviewPane.ContextMenuStrip = null;
```

### What You've Accomplished

The default context menu does not appear when you right-click the preview pane in the C1PrintPreviewControl control.