
ComponentOne

PDF for .NET

Copyright © 2012 ComponentOne LLC. All rights reserved.

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 PDF for .NET Overview	1
Installing PDF for .NET	1
PDF for .NET Setup Files	1
System Requirements	2
Installing Demonstration Versions	2
Uninstalling PDF for .NET	2
End-User License Agreement	3
Licensing FAQs	3
What is Licensing?	3
How does Licensing Work?	3
Common Scenarios	4
Troubleshooting	6
Technical Support	7
Redistributable Files	8
About this Documentation	8
Namespaces	9
Creating a .NET Project	10
Adding the C1Pdf Component to a Project	10
Migrating a PDF for .NET Project to Visual Studio 2005	11
Key Features	13
Using ComponentOne PDF for .NET	14
Creating Documents	14
Adding Text	16
Drawing Text	16
Measuring Text	17
Making Text Flow from Page to Page	18
Drawing RTF Text	19
Adding Images	20
Adding Graphics	21
Creating Pages and Overlays	23

Adding Bookmarks and Annotations.....	26
Adding Bookmarks to a PDF Document	26
Adding Links to a PDF Document.....	27
Attaching Files to a PDF Document	29
Applying Security and Permissions.....	30
Using Metafiles	31
Rendering RTF Text.....	33
PDF for .NET Samples	35
PDF for .NET Task-Based Help.....	37
Setting Document Properties	37
Setting Document Restrictions	39
Setting Paper Size	42
Setting Landscape Page Orientation	42
Setting Image Quality.....	43
Printing PDF Files.....	44
Writing PDF for .NET Content in Web Applications.....	44
Using PDF for .NET with Other ComponentOne Tools	45
Adding ComponentOne FlexGrid for WinForms Data into PDF for .NET.....	45
Creating a PDF from a Report Rendered by VSPrinter	45
Removing PDF Bookmarks from a C1Report Document	46

ComponentOne PDF for .NET Overview

ComponentOne PDF for .NET is a library that allows you to create PDF documents from your applications.

The PDF (Portable Document Format) format was created by Adobe in 1993 and has since become the industry standard for the representation of printed material in electronic systems. PDF is popular because it is high-quality, portable, device-independent, and because there are many tools available for creating, manipulating, and viewing PDF documents. These tools include the free multi-platform document viewer Adobe Acrobat. The portability and high quality of the PDF documents make PDF an excellent choice for Web publishing. For more information on the PDF format and PDF utilities from Adobe and other sources, visit the Adobe Web site (www.adobe.com).

For a list of the latest features added to **ComponentOne Studio for WinForms**, visit [What's New in Studio for WinForms](#).

Getting Started

To get started, review the following topics:

- [Key Features](#) (page 13)
- [Creating Documents](#) (page 14)
- [Samples](#) (page 35)

Installing PDF for .NET

This section describes the setup files and system requirements for **ComponentOne PDF for .NET**. It also explains how to install demonstration versions of ComponentOne products and uninstall **PDF for .NET**.

PDF for .NET Setup Files

The ComponentOne Studio for WinForms installation program will create the directory C:\Program Files\ComponentOne\Studio for WinForms. This directory contains the following subdirectories:

bin	Contains copies of all ComponentOne binaries (DLLs, EXEs).
C1PDF	Contains samples and tutorials for ComponentOne PDF for .NET .

The **ComponentOne Studio for WinForms Help Setup** program installs integrated Microsoft Help Viewer help to the C:\Program Files\ComponentOne\Studio for WinForms\HelpViewer folder.

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\\My Documents\ComponentOne Samples

Windows 7/Vista path: C:\Users\\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.
C1Pdf	Contains samples and tutorials for PDF for .NET .

Samples can be accessed from the **ComponentOne Sample Explorer**. To view samples, on your desktop, click the **Start** button and then click **All Programs | ComponentOne | Studio for WinForms (or Studio for ASP.NET) | Samples | Pdf Samples**.

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 **PDF for .NET** 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 PDF for .NET

To uninstall **ComponentOne PDF for .NET**:

1. Open **Control Panel** and select **Add or Remove Programs (Programs and Features in Vista/Windows 7)**.
2. Select **ComponentOne Studio for WinForms** and click the **Remove** button.
3. Click **Yes** to remove the program.

To uninstall **ComponentOne PDF for .NET** integrated help:

1. Open the Control Panel and select **Add or Remove Programs (Programs and Features in Windows 7/Vista)**.
2. Select **ComponentOne Studio for WinForms Help** and click the **Remove** button.
3. Click **Yes** to remove the integrated help.

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 .NET 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 .NET 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 app 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 affected project.
2. Select an instance of the updated component.
3. In the Visual Studio Properties window, change any property. It does not matter which property you change; you can change it back to the previous value.
4. Rebuild the project using the **Rebuild All** option (not just **Rebuild**) and run the solution.

Alternative 1: Follow these steps:

1. Open a new Visual Studio.NET project.
2. Add the updated component to the form.
3. Compile and run the new project.
4. Open the licenses.licx file in the new project.
5. Copy the line that starts with the namespace of the updated component (for example, C1.Win.C1Report) and ends with a public key token.
6. Open the existing, incorrectly licensed project.
7. Open the licenses.licx file in the new project.
8. Paste the line from step 5 into this file (replace the old licensing information with the new).
9. Rebuild the project using the **Rebuild All** option (not just **Rebuild**) and run the solution.

Alternative 2: Follow these steps:

1. Open the affected project.
2. Delete the licenses.licx file from the project.
3. Add a new instance of the updated component to the form.
4. Rebuild and run the solution. The nag screen should not appear.
5. Remove the newly added component from the form.

Try each of these options multiple times, if necessary. If that still does not help, are you creating any of the controls in code rather than design-time? If so, you must add an entry for the control in the licenses.licx file (see <http://helpcentral.componentone.com/PrintableView.aspx?ID=1886> for more information). Also if this is a website, as opposed to an ASP.NET web application, please try right-clicking the licenses.licx file and selecting "Build Runtime Licenses" from the context menu.

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 .NET user controls with constituent licensed controls, the run-time license is embedded in the .NET 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.

- **Product Forums**
ComponentOne's [product forums](#) are available for users to share information, tips, and techniques regarding ComponentOne products. ComponentOne developers will be available on the forums to share insider tips and technique and answer users' questions. 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.

Redistributable Files

ComponentOne PDF for .NET 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). You may also distribute, free of royalties, the following 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:

- C1.C1Pdf.2.dll
- C1.Pdf.4.dll
- C1.Pdf.4.Design.dll

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

About this Documentation

Acknowledgements

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

Adobe Acrobat and Acrobat Reader are either registered trademarks or trademarks of Adobe Systems Incorporated 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®](#).

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's products is **C1.Win**. The namespace for the **C1PdfDocument** component is **C1.C1Pdf**. The following code fragment shows how to declare a **C1PdfDocument** component using the fully qualified name for this class:

- Visual Basic

```
Dim pdf As C1.C1Pdf.C1PdfDocument
```

- C#

```
C1.C1Pdf.C1PdfDocument pdf;
```

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 **C1PdfDocument**, you can use it inside your project without qualification. However, the **C1PDF** assembly also implements a class called **C1PdfDocument**. So, if you want to use the **C1PdfDocument** 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 object based on your C1PdfDocument class.  
Dim MyC1PdfDocument as C1PdfDocument  
  
' Define a new C1Pdf.C1PdfDocument object.  
Dim Pdf as C1.C1Pdf.C1PdfDocument
```

- C#

```
// Define a new object based on your C1PdfDocument class.  
MyC1PdfDocument as C1PdfDocument;  
  
// Define a new C1Pdf.C1PdfDocument object.  
Pdf as C1.C1Pdf.C1PdfDocument;
```

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 C1PdfDocument = C1.C1Pdf.C1PdfDocument  
Imports MyC1PdfDocument = MyProject.Objects.C1PdfDocument  
  
Dim c1 As C1PdfDocument  
Dim c2 As MyC1PdfDocument
```

- C#

```
using C1PdfDocument = C1.C1Pdf.C1PdfDocument;
using MyC1PdfDocument = MyProject.Objects.C1PdfDocument;

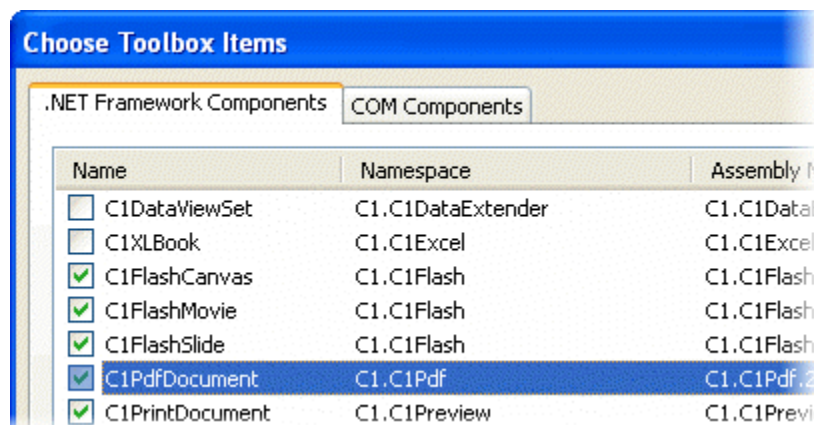
C1PdfDocument c1;
MyC1PdfDocument c2;
```

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 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 new application in the **Name** field or you can use the default name, `WindowsApplication1`.
A new Microsoft Visual Studio .NET project is and a new `Form1` is displayed in the Designer view.
4. Double-click the **C1PdfDocument** component from the Toolbox to add it to `Form1`. For information on adding a component to the Toolbox, see [Adding the C1Pdf Component to a Project](#) (page 10).

Adding the C1Pdf Component to a Project

When you install ComponentOne Studio for WinForms, the **Create a ComponentOne Visual Studio Toolbox Tab** check box is selected, by default, in the installation wizard. When you open Visual Studio, you will notice a **ComponentOne Studio for WinForms** tab containing the ComponentOne controls has automatically been added to the Toolbox.

If you decide to uncheck the **Create a ComponentOne Visual Studio Toolbox Tab** check box during installation, you can manually add ComponentOne controls to the Toolbox at a later time.

ComponentOne PDF for .NET provides the **C1PdfDocument** control. To use **C1Pdf**, add it to the form or add a reference to the `C1.C1Pdf` assembly to your project.

Manually Adding C1Pdf to the Toolbox

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

- C1PdfDocument

To manually add the **C1PdfDocument** 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 then right-click it to open the context menu.
2. To make the **C1PdfDocument** component appear on its own tab in the Toolbox, select **Add Tab** from the context menu and type in the tab name, **C1Pdf**, 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, go to the **.NET Framework Components** tab. Sort the list by Namespace (click the Namespace column header) and check the check box for the component belonging to namespace C1.C1Pdf. Note that there may be more than one component for each namespace, but there is only one for C1.C1Pdf.

Adding C1PdfDocument to the Form

To add **C1PdfDocument** to the form:

1. Add it 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 **C1Pdf** assembly:

1. Select the **Add Reference** option from the **Project** menu of your project.
2. Select the **ComponentOne C1Pdf** assembly from the list on the **.NET** tab or browse to find the C1.C1Pdf.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** statement (**using** in C#):

```
Imports C1.C1Pdf
```

Note: This makes the objects defined in the **C1Pdf** assembly visible to the project. See [Namespaces](#) (page 9) for more information.

Migrating a PDF for .NET 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**.
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.C1Pdf the same way.
10. Right-click the **References** node and select **Add Reference**.
11. Locate and select **C1.C1Pdf.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

ComponentOne PDF for .NET supports most of the advanced features included in the PDF specification, including security, compression, outlining, hyperlinking, and attachments.

But the main feature in **PDF for .NET** is ease of use. The commands provided for adding content to documents are similar to the ones available in the .NET **Graphics** class. If you know how to display text and graphics in .NET, you already know how to use **PDF for .NET**.

The following are some of the features of **PDF for .NET** that you may find useful:

- **Fast rendering and compression of images in Metafiles**

Metafiles are not converted into bitmaps; they are parsed and converted into vector graphics commands and thus retain the best possible resolution. If you want to add charts or technical drawings to your PDF document, metafiles are better than bitmap images.
- **AcroForms support**

Use the new `AddField` method to add Acrobat form fields to your PDF documents. **PDF for .NET** supports the following field types: textbox, checkbox, radio button, push button, combo box, and list box.
- **HTML Support**

Use the new `DrawStringHtml` method to render HTML into your PDF documents. You can flow HTML content into multiple pages or columns, use existing style sheets, and mix HTML with other types of content (images, RTF, plain text, form fields, and so on).
- **Familiar syntax using the `DrawImage` method**

Adding images to PDF documents is easy; all the work is done by the `DrawImage` method. `DrawImage` draws a given image at a specified location and has parameters that provide control over the image alignment and scaling. You can render any regular .NET Image object, including metafiles.
- **Manage document restrictions**

Allow users to copy and edit content, restrict users from printing the document, set annotation edit permission to the user, and more.
- **Add attachments to PDF files**

Attachments can contain any kind of file, including spreadsheets with detailed information that would clutter the main document, multimedia files with movies and sound, sample code, and more. Adding an attachment to your PDF file is easy. Simply specify which file you want to attach, what area of the page should contain the attachment, and optionally, the appearance of the attachment.
- **Owner and user password protection**

If your PDF documents contain sensitive information, you can encrypt them so that only authorized users can access it. There is a separate password for the owner of the document and for all other users. The user's access can be selectively restricted to allow only certain operations, such as viewing, printing, or editing the document.
- **Add graphical elements**

Add lines, rectangles, ellipses, pies, arcs, rounded rectangles, polygons, Bezier curves, and more.
- **Create an Outline structure**

Most long PDF documents contain an outline structure that is displayed on a pane on the left of the reader. The outline makes it easy to browse through a document's structure and find specific topics. With **PDF for .NET**, you can build this outline structure by adding outline entries (bookmarks).

- **Add hyperlinks and local links**

PDF provides methods for adding hyperlinks and hyperlink targets to your PDF documents. You can also add local links, that when clicked take the user to another location within the same PDF document. This type of link is useful when you want to implement some type of cross-referencing within the document, such as a table of contents or an index.

- **Control document information and viewer preferences**

PDF allows you to add meta data to the PDF documents you create. Specify author, creation date, keywords, and so on. You can also provide default viewer preferences to be applied when the document is opened in the Adobe Reader. Specify the initial page layout, window position, as well as reader toolbar and menu visibility.

Using ComponentOne PDF for .NET

The following topics provide details on creating documents using **PDF for .NET**, as well as how to add text, images, graphics, pages and overlays, bookmarks and annotations, and security and permissions to **PDF for .NET** documents.

Creating Documents

To create PDF documents using **PDF for .NET**, you simply create a `C1PdfDocument` object, add content to the document, and save the document. To follow tradition, here's how to create a "hello world" document using **PDF for .NET**.

1. Add the following **Imports** or **using** directive to your code.

- Visual Basic

```
Imports System.Drawing
```

- C#

```
using System.Drawing;
```

2. Create a `C1PdfDocument` object.

- Visual Basic

```
' Create the C1PdfDocument object.  
Dim pdf As New C1.C1Pdf.C1PdfDocument()
```

- C#

```
// Create the C1PdfDocument object.  
C1.C1Pdf.C1PdfDocument pdf = new C1.C1Pdf.C1PdfDocument();
```

3. Add your content to the **Form_Load** event. This usually involves calling the `DrawString` method.

- Visual Basic

```
' Add content to the page.  
Dim rect As RectangleF = pdf.PageRectangle  
rect.Inflate(-72, -72)  
Dim font As Font = New Font("Arial",12)  
pdf.DrawString("Hello World!", font, Brushes.Black, rect)
```

- C#

```
// Add content to the page.  
RectangleF rect = pdf.PageRectangle;  
rect.Inflate(-72, -72);  
Font font = new Font("Arial", 12);  
pdf.DrawString("Hello World!", font, Brushes.Black, rect);
```

4. Save the document to a file or to a stream using the **Save** method.

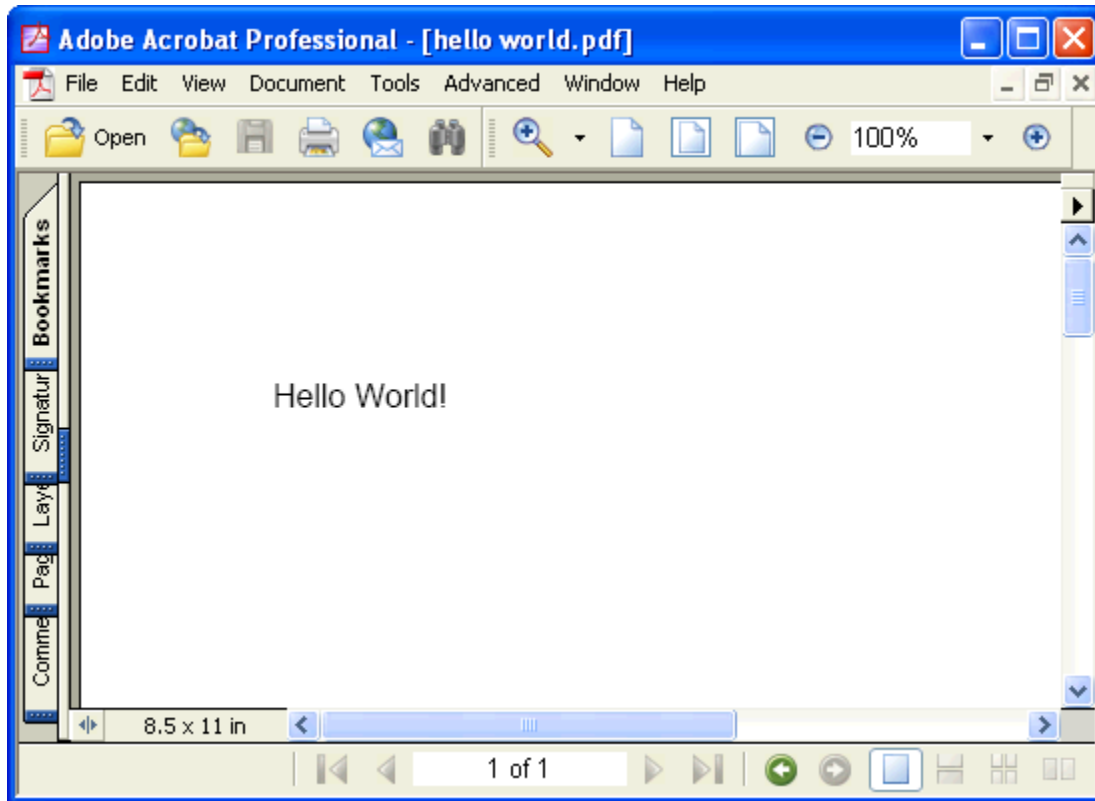
- Visual Basic

```
' Save the document to a file.  
pdf.Save("c:\hello world.pdf")
```

- C#

```
// Save the document to a file.  
pdf.Save(@"c:\hello world.pdf");
```

After the application is run, the hello world.pdf document will look like this:



Step 3 is the most interesting one. The code starts by retrieving the page rectangle, expressed in points. It then adds a one-inch margin around the page (72 points). Finally, the code creates a **Font** object and calls the `DrawString` method to write "Hello World!" on the page. This is exactly what you would do if you were writing to a **Graphics** object in .NET and is what makes **PDF for .NET** so easy to use.

One important thing to remember is that **PDF for .NET** uses a point-based coordinate system with the origin at the top-left corner of the page. This is similar to the default coordinate system used by .NET, but is different from the default PDF coordinate system (where the origin is on the bottom-left corner of the page). In this example, the top left point of the "H" in "Hello World" is located at [1,1].

Because the coordinate system is based on points, rather than pixels, **PDF for .NET** uses **RectangleF**, **SizeF**, and **PointF** structures, which have members of type **float**, rather than **Rectangle**, **Size**, and **Point**, which have members of type **int**.

Adding Text

The following topics provide information on drawing, measuring, and managing the flow of text, as well as drawing RTF text.

Drawing Text

Adding text to **PDF for .NET** documents is easy – all the work is done by the `DrawString` method.

`DrawString` draws a given string at a specified location using a given font and brush. For example:

- Visual Basic

```
pdf.DrawString("Hello World!", font, Brushes.Black, rect)
```

- C#

```
pdf.DrawString("Hello World!", font, Brushes.Black, rect);
```

By default, `DrawString` will align the text to the left and to the top of the given rectangle, will wrap the string within the rectangle, and will not clip the output to the rectangle. You can change all these options by specifying a *StringFormat* parameter in the call to `DrawString`. The **StringFormat** has members that allow you to specify the horizontal alignment (**Alignment**), vertical alignment (**LineAlignment**), and flags that control wrapping and clipping (**FormatFlags**).

For example, the code below creates a **StringFormat** object and uses it to align the text to the center of the rectangle, both vertically and horizontally:

- Visual Basic

```
Dim font As New Font("Arial", 12)
Dim rect As New RectangleF(72, 72, 100, 50)
Dim text As String = "Some long string to be rendered into a small
rectangle. "
text = text & text & text & text & text & text

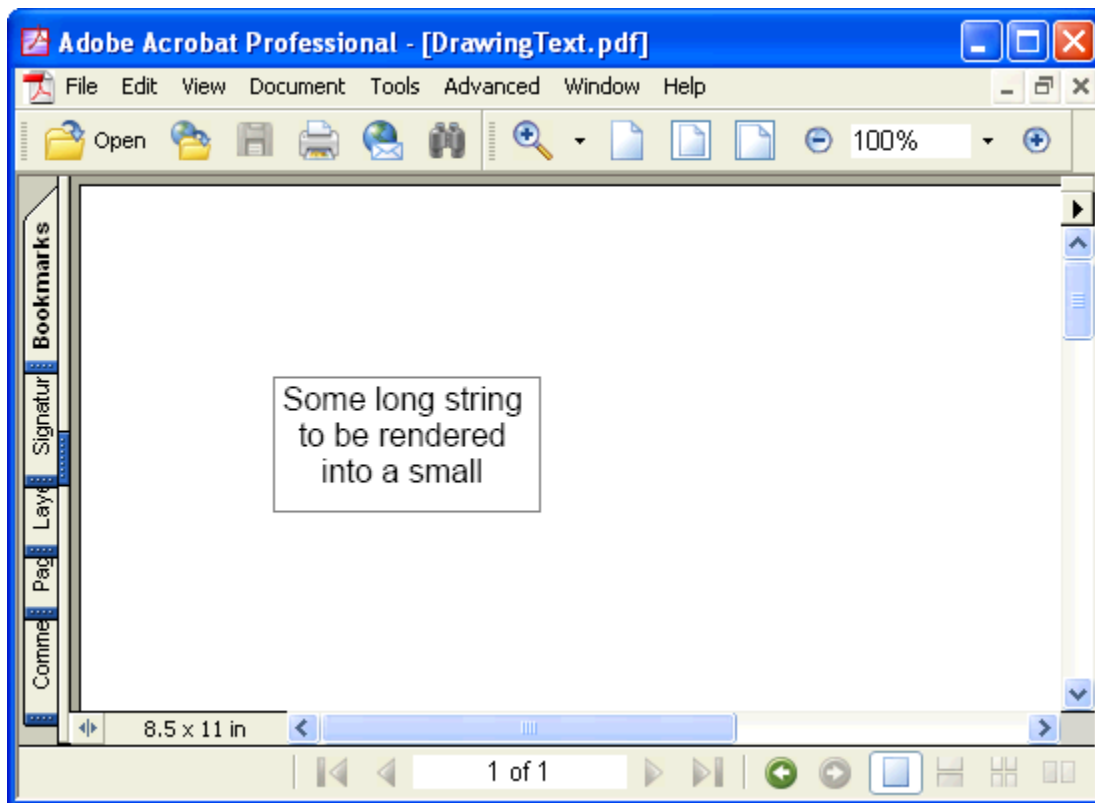
' Center align string.
Dim sf As New StringFormat()
sf.Alignment = StringAlignment.Center
sf.LineAlignment = StringAlignment.Center
pdf.DrawString(text, font, Brushes.Black, rect, sf)
pdf.DrawRectangle(Pens.Gray, rect)
```

- C#

```
Font font = new Font("Arial", 12);
RectangleF rect = new RectangleF(72, 72, 100, 50);
string text = "Some long string to be rendered into a small rectangle. ";
text = text + text + text + text + text + text;

// Center align string.
StringFormat sf = new StringFormat();
sf.Alignment = StringAlignment.Center;
sf.LineAlignment = StringAlignment.Center;
pdf.DrawString(text, font, Brushes.Black, rect, sf);
pdf.DrawRectangle(Pens.Gray, rect);
```

Here is the resulting PDF document:



Measuring Text

In many cases, you will need to check whether the string will fit on the page before you render it. You can use the `MeasureString` method for that. `MeasureString` returns a `SizeF` structure that contains the width and height of the string (in points) when rendered with a given font.

For example, the code below checks to see if a paragraph will fit on the current page and creates a page break if it has to. This will keep paragraphs together on a page:

- Visual Basic

```
Private Function RenderParagraph(text As String, font As Font, rect As RectangleF, rectPage As RectangleF) As RectangleF

    ' Calculate the necessary height.
    Dim sz As SizeF = _clpdf.MeasureString(text, font, rect.Width)
    rect.Height = sz.Height

    ' If it won't fit this page, do a page break.
    If rect.Bottom > rectPage.Bottom Then
        _clpdf.NewPage()
        rect.Y = rectPage.Top
    End If

    ' Draw the string.
    _clpdf.DrawString(text, font, Brushes.Black, rect)

    ' Update rectangle for next time.
    Rect.Offset(0, rect.Height)
    Return rect
```

```

End Function

' Use the RenderParagraph method.
Dim font As New Font("Arial", 10)
Dim rectPage As RectangleF = _clpdf.PageRectangle
rectPage.Inflate(-72, -72)
Dim rect As RectangleF = rectPage
Dim s As String
For Each s In myStringList
    rect = RenderParagraph(s, font, rect, rectPage)
Next s

```

- C#

```

private RectangleF RenderParagraph(string text, Font font, RectangleF
rect, RectangleF rectPage)
{
    // Calculate the necessary height.
    SizeF sz = _clpdf.MeasureString(text, font, rect.Width);
    rect.Height = sz.Height;

    // If it won't fit this page, do a page break.
    If (rect.Bottom > rectPage.Bottom)
    {
        _clpdf.NewPage();
        rect.Y = rectPage.Top;
    }

    // Draw the string.
    _clpdf.DrawString(text, font, Brushes.Black, rect);

    // Update rectangle for next time.
    Rect.Offset(0, rect.Height);
    return rect;
}

// Use the RenderParagraph method.
Font font = new Font("Arial", 10);
RectangleF rectPage = _clpdf.PageRectangle;
rectPage.Inflate(-72, -72);
RectangleF rect = rectPage;
foreach (string s in myStringList)
{
    rect = RenderParagraph(s, font, rect, rectPage);
}

```

Making Text Flow from Page to Page

The DrawString method returns an integer. This is the index of the first character that was not printed because it did not fit the output rectangle. You can use this value make text flow from page to page or from one frame to another within a page. For example:

- Visual Basic

```

' Render a string spanning multiple pages.
While True

    ' Render as much as will fit into the rectangle.
    Dim nextChar As Integer
    nextChar = _clpdf.DrawString(text, font, Brushes.Black, rectPage)

```

```

' Break when done.
If nextChar >= text.Length Then
    Exit While
End If

' Get rid of the part that was rendered.
Text = text.Substring(nextChar)

' Move on to the next page.
_clpdf.NewPage()
End While

```

- C#

```

// Render a string spanning multiple pages.
While (true)
{
    // Render as much as will fit into the rectangle.
    Int nextChar = _clpdf.DrawString(text, font, Brushes.Black, rectPage);

    // Break when done.
    If (nextChar >= text.Length)
    {
        break;
    }

    // Get rid of the part that was rendered.
    Text = text.Substring(nextChar);

    // Move on to the next page.
    _clpdf.NewPage();
}

```

By combining the `MeasureString` and `DrawString` methods, you can develop rendering routines that provide extensive control over how paragraphs are rendered, including keeping paragraphs together on a page, keeping with the next paragraph, and controlling widows and orphans (single lines that render on the current or next page).

Drawing RTF Text

`DrawString` provides all the functionality you need for rendering paragraphs using a single font and color.

If you want to render paragraphs with rich format, mixing fonts and colors, you should use the `DrawStringRtf` method instead. `DrawStringRtf` is similar to `DrawString`, except the string is interpreted as RTF.

For example, the code below renders a line of text with some bold and italic characters in it:

- Visual Basic

```

Dim font As New Font("Arial", 12)
Dim rect As RectangleF = pdf.PageRectangle
rect.Inflate(- 72, - 72)
Dim s As String
s = "To {\b boldly} go where {\i no one} has gone before!"
pdf.DrawStringRtf(s, font, Brushes.Black, rect)

```

- C#

```

Font font = new Font("Arial", 12);
RectangleF rect = pdf.PageRectangle;
rect.Inflate(-72, -72);

```

```
pdf.DrawStringRtf(@"To {\b boldly} go where {\i no one} has gone before!",  
font, Brushes.Black, rect);
```

Notice that the string does not contain a full RTF document. It has no `\rtf1` header, font or color tables. The `DrawStringRtf` method recognizes that and builds the necessary RTF header automatically, based on the *Font* and *Brush* parameters provided.

If the string contained a complete RTF header, that would be used in lieu of the *font* and *brush* parameters specified in the call `DrawStringRtf`. This would be the case if the string were copied from a **RichTextBox** control or loaded from an RTF file.

Adding Images

Adding images to **PDF for .NET** documents is also easy, all the work is done by the `DrawImage` method.

`DrawImage` draws a given image at a specified location and has parameters that provide control over the image alignment and scaling. In the following example, this image is:

- Stretched to fill the given rectangle
- Center-aligned within the rectangle, scaled to keep the aspect ratio
- Aligned to the top-left corner of the rectangle, with the original size

This code is used to draw the same image three times:

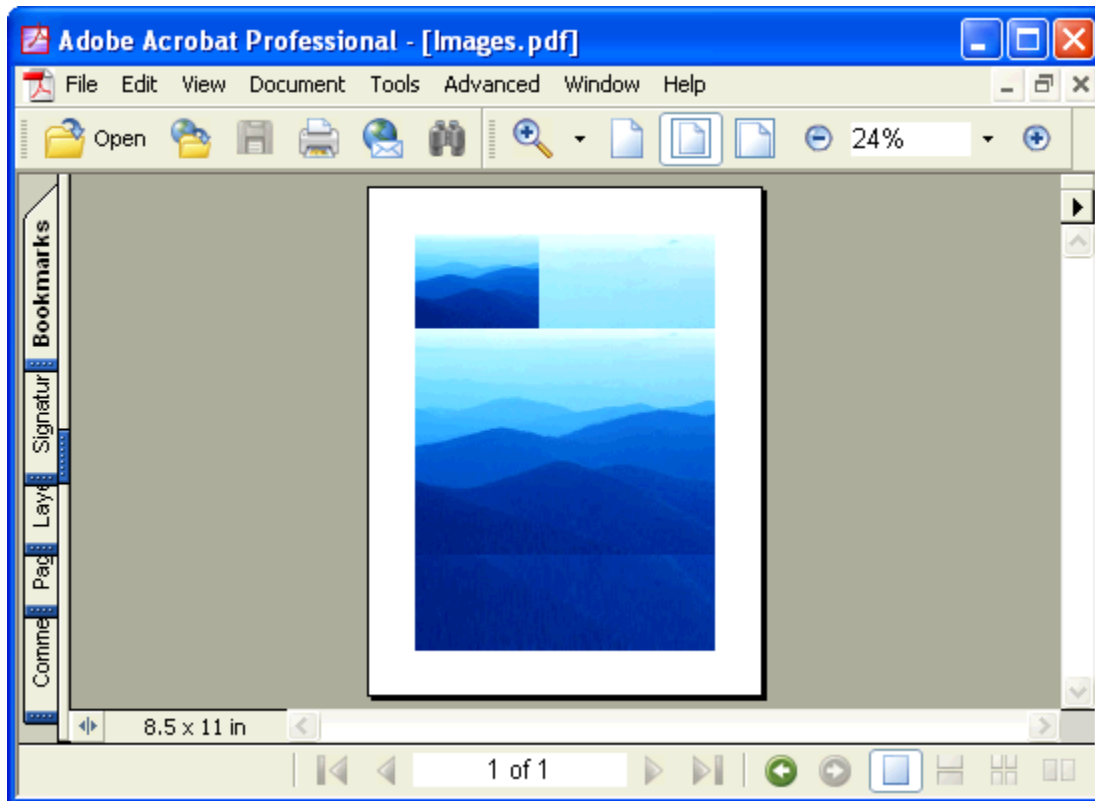
- Visual Basic

```
Dim rect As RectangleF = pdf.PageRectangle  
rect.Inflate(- 72, - 72)  
  
' Stretch image to fill the rectangle.  
pdf.DrawImage(pictureBox1.Image, rect)  
  
' Center image within the rectangle, scale keeping aspect ratio.  
pdf.DrawImage(pictureBox1.Image, rect, ContentAlignment.MiddleCenter,  
C1.C1Pdf.ImageSizeModeEnum.Scale)  
  
' Render the image to the top left corner of the rectangle.  
pdf.DrawImage(pictureBox1.Image, rect, ContentAlignment.TopLeft,  
C1.C1Pdf.ImageSizeModeEnum.Clip)
```

- C#

```
RectangleF rect = pdf.PageRectangle;  
rect.Inflate(-72, -72);  
  
// Stretch image to fill the rectangle.  
pdf.DrawImage(pictureBox1.Image, rect);  
  
// Center image within the rectangle, scale keeping aspect ratio.  
pdf.DrawImage(pictureBox1.Image, rect, ContentAlignment.MiddleCenter,  
C1.C1Pdf.ImageSizeModeEnum.Scale);  
  
// Render the image to the top left corner of the rectangle.  
pdf.DrawImage(pictureBox1.Image, rect, ContentAlignment.TopLeft,  
1.C1Pdf.ImageSizeModeEnum.Clip);
```

The PDF document will look similar to this:



Notice that you can render any regular .NET Image object, including Metafiles. Metafiles are not converted into bitmaps; they are played into the document and thus retain the best possible resolution. If you want to add charts or technical drawings to your PDF document, Metafiles are better than bitmap images. **PDF for .NET** also supports EMF+ metafiles.

Bitmap images are managed automatically by **PDF for .NET**. If you render the same image several times (in a page header for example), only one copy of the image is saved into the PDF file. You can control the quality of the images using the **C1PdfDocumentBase.ImageQuality** property, which allows you to trade-off image quality versus file size.

Adding Graphics

The **C1PdfDocument** class exposes several methods that allow you to add graphical elements to your documents, including lines, rectangles, ellipses, pies, arcs, rounded rectangles, polygons, Bezier curves, and so on.

The methods are a subset of those found in the .NET **Graphics** class, and use the same **Brush** and **Pen** classes to control the color and style of the lines and filled areas.

It is important to remember that **PDF for .NET** uses a coordinate system based on points, with the origin located at the top left of the page. (The default coordinate system for the .NET **Graphics** class is pixel-based.)

The example below illustrates how similar the graphics methods are between **PDF for .NET** and the .NET **Graphics** class. The sample declares a **C1PdfDocument** class called 'g' and calls methods to draw pies, splines, and other graphical elements.

The point of the sample is that if you replaced the **C1PdfDocument** class with a regular .NET **Graphics** object, you would be able to compile the code and get the same results:

- Visual Basic

```
' Create PDF document.
Dim g As New C1.C1Pdf.C1PdfDocument()
```

```

' Set up to draw.
Dim rect As New RectangleF(0, 0, 300, 200)
Dim text As String = "Hello world of .NET Graphics and PDF." +
ControlChars.Cr + ControlChars.Lf + "Nice to meet you."
Dim font As New Font("Times New Roman", 12, FontStyle.Italic Or
FontStyle.Underline)
Dim bezierPoints() As PointF = {New PointF(10F, 100F), New PointF(20F,
10F), New PointF(35F, 50F), New PointF(50F, 100F), New PointF(60F, 150F),
New PointF(65F, 100F), New PointF(50F, 50F)}

' Draw some pie slices.
Dim penWidth As Integer = 0
Dim penRGB As Integer = 0
g.FillPie(Brushes.Red, rect, 0, 20F)
g.FillPie(Brushes.Green, rect, 20F, 30F)
g.FillPie(Brushes.Blue, rect, 60F, 12F)
g.FillPie(Brushes.Gold, rect, - 80F, - 20F)

' Draw some arcs.
Dim startAngle As Single
For startAngle = 0 To 360 - 40 Step 40
    Dim penColor As Color = Color.FromArgb(penRGB, penRGB, penRGB)
    penWidth = penWidth + 1
    Dim pen As New Pen(penColor, penWidth)
    penRGB = penRGB + 20
    g.DrawArc(pen, rect, startAngle, 40F)
Next

' Draw a rectangle and some bezier splines.
g.DrawRectangle(Pens.Red, rect)
g.DrawBeziers(Pens.Blue, bezierPoints)
g.DrawString(text, font, Brushes.Black, rect)

```

- **C#**

```

// Create PDF document.
C1.C1Pdf.C1PdfDocument g = new C1.C1Pdf.C1PdfDocument();

// Set up to draw.
Rectangle rect = new RectangleF(0,0,300,200);
string text = "Hello world of .NET Graphics and PDF.\r\n" + "Nice to meet
you.";
Font font = new Font("Times New Roman", 12, FontStyle.Italic |
FontStyle.Underline);
PointF[] bezierPoints = new PointF[]
{
    new PointF(10f, 100f), new PointF(20f, 10f), new PointF(35f, 50f), new
PointF(50f, 100f), new PointF(60f, 150f), new PointF(65f, 100f), new
PointF(50f, 50f)
};

// Draw some pie slices.
int penWidth = 0;
int penRGB = 0;
g.FillPie(Brushes.Red, rect, 0, 20f);
g.FillPie(Brushes.Green, rect, 20f, 30f);
g.FillPie(Brushes.Blue, rect, 60f, 12f);

```

```

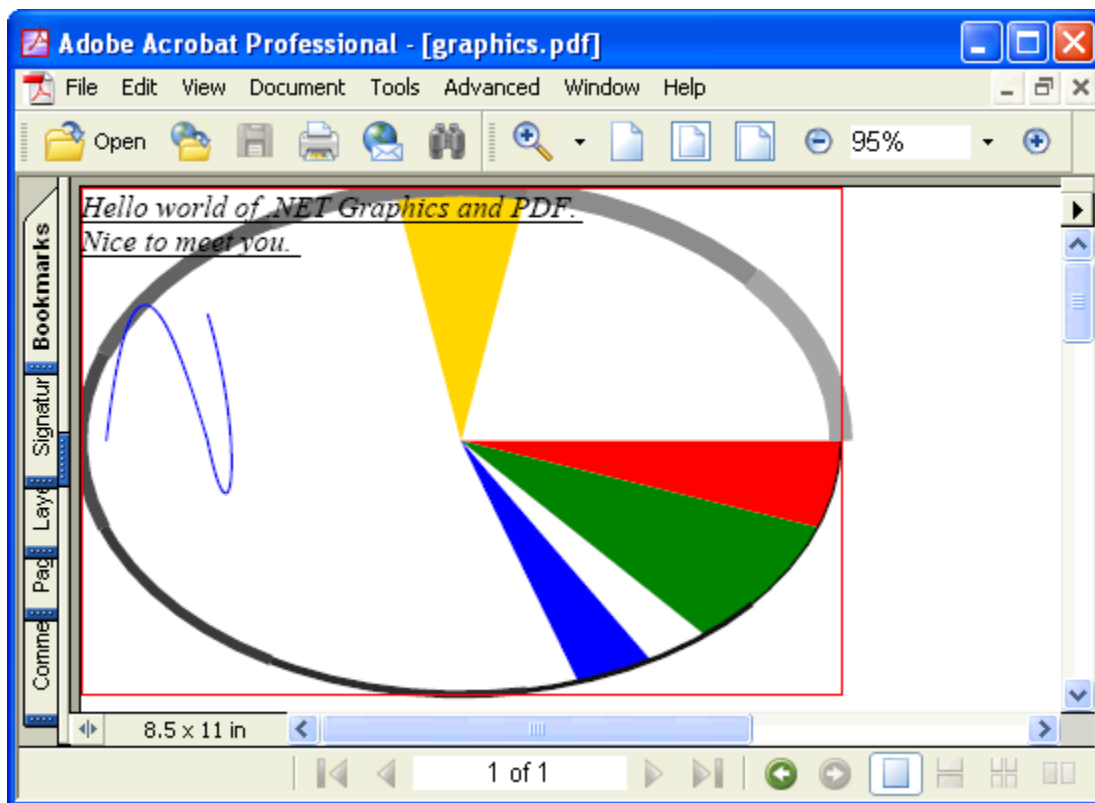
g.FillPie(Brushes.Gold, rect, -80f, -20f);

// Draw some arcs.
for (float startAngle = 0; startAngle < 360; startAngle += 40)
{
    Color penColor = Color.FromArgb(penRGB, penRGB, penRGB);
    Pen pen = new Pen(penColor, penWidth++);
    penRGB = penRGB + 20;
    g.DrawArc(pen, rect, startAngle, 40f);
}

// Draw a rectangle and some bezier splines.
g.DrawRectangle(Pens.Red, rect);
g.DrawBeziers(Pens.Blue, bezierPoints);
g.DrawString(text, font, Brushes.Black, rect);

```

Here is the resulting PDF document:



Creating Pages and Overlays

You may have noticed that in the previous examples, we started adding content to the document right after creating the `C1PdfDocument` object. This is possible because when you create the `C1PdfDocument`, it automatically adds an empty page to the document, ready to receive any type of content.

When you are done filling up the first page, you can add a new one using the `C1PdfDocumentBase.NewPage` method.

By default, all pages in the document have the same size and orientation. These parameters can be specified in the `C1PdfDocument` constructor. You can also change the page size and orientation at any time by setting the `PaperKind`, `PageSize`, and `Landscape` properties. For example, the code below creates a document with all paper sizes defined by the **PaperKind** enumeration:

- Visual Basic

```
Dim font As New Font("Arial", 9)
Dim sf As New StringFormat()
sf.Alignment = StringAlignment.Center
sf.LineAlignment = StringAlignment.Center

' Create one page with each paper size.
Dim firstPage As Boolean = True
Dim pk As PaperKind
For Each pk In System.Enum.GetValues(GetType(PaperKind))

    ' Skip custom size.
    If pk = PaperKind.Custom Then
        GoTo ContinueForEach1
    End If

    ' Add new page for every page after the first one.
    If Not firstPage Then
        _clpdf.NewPage()
    End If
    firstPage = False

    ' Set paper kind.
    _clpdf.PaperKind = pk

    ' Draw some content on the page.
    _clpdf.DrawString("PaperKind: " + pk.ToString(), font, Brushes.Black,
        _clpdf.PageRectangle, sf)

    ContinueForEach1:
Next pk
```

- C#

```
Font font = new Font("Arial", 9);
StringFormat sf = new StringFormat();
sf.Alignment = StringAlignment.Center;
sf.LineAlignment = StringAlignment.Center;

// Create one page with each paper size.
bool firstPage = true;
foreach (PaperKind pk in Enum.GetValues(typeof(PaperKind)))
{

    // Skip custom size.
    if (pk == PaperKind.Custom)
    {
        continue;
    }

    // Add new page for every page after the first one.
    if (!firstPage)
    {
```

```

        _clpdf.NewPage();
    }
    firstPage = false;

    // Set paper kind.
    _clpdf.PaperKind = pk;

    // Draw some content on the page.
    _clpdf.DrawString("PaperKind: " + pk.ToString(), font, Brushes.Black,
        _clpdf.PageRectangle, sf);
}

```

You are not restricted to writing on the last page that was added to the document. You can use the `CurrentPage` property to select which page you want to write to, and then use the regular drawing commands as usual. This is useful for adding content to pages after you are done rendering a document. For example, the code below adds footers to each page containing the current page number and the total of pages in the document (page *n* of *m*):

- Visual Basic

```

Private Sub AddFooters()
    Dim font As New Font("Tahoma", 7, FontStyle.Bold)
    Dim sf As New StringFormat()
    sf.Alignment = StringAlignment.Center
    Dim page As Integer
    For page = 0 To _clpdf.Pages.Count - 1

        ' Select page.
        _clpdf.CurrentPage = page

        ' Build rectangle for rendering the footer.
        Dim rect As RectangleF = _clpdf.PageRectangle
        rect.Y = rect.Bottom

        ' Write the footer.
        Dim text As String
        text = String.Format("Page {0} of {1}", page + 1,
            _clpdf.Pages.Count)
        _clpdf.DrawString(text, font, Brushes.Gray, rect, sf)
    Next page
End Sub

```

- C#

```

private void AddFooters()
{
    Font font = new Font("Tahoma", 7, FontStyle.Bold);
    StringFormat sf = new StringFormat();
    sf.Alignment = StringAlignment.Center;
    for (int page = 0; page < _clpdf.Pages.Count; page++)
    {
        // Select page.
        _clpdf.CurrentPage = page;

        // Build rectangle for rendering the footer.
        RectangleF rect = _clpdf.PageRectangle;
        rect.Y = rect.Bottom - 36;

        // Write the footer.
        string text = string.Format("Page {0} of {1}", page+1,
            _clpdf.Pages.Count);
    }
}

```

```

        _clpdf.DrawString(text, font, Brushes.Gray, rect, sf);
    }
}

```

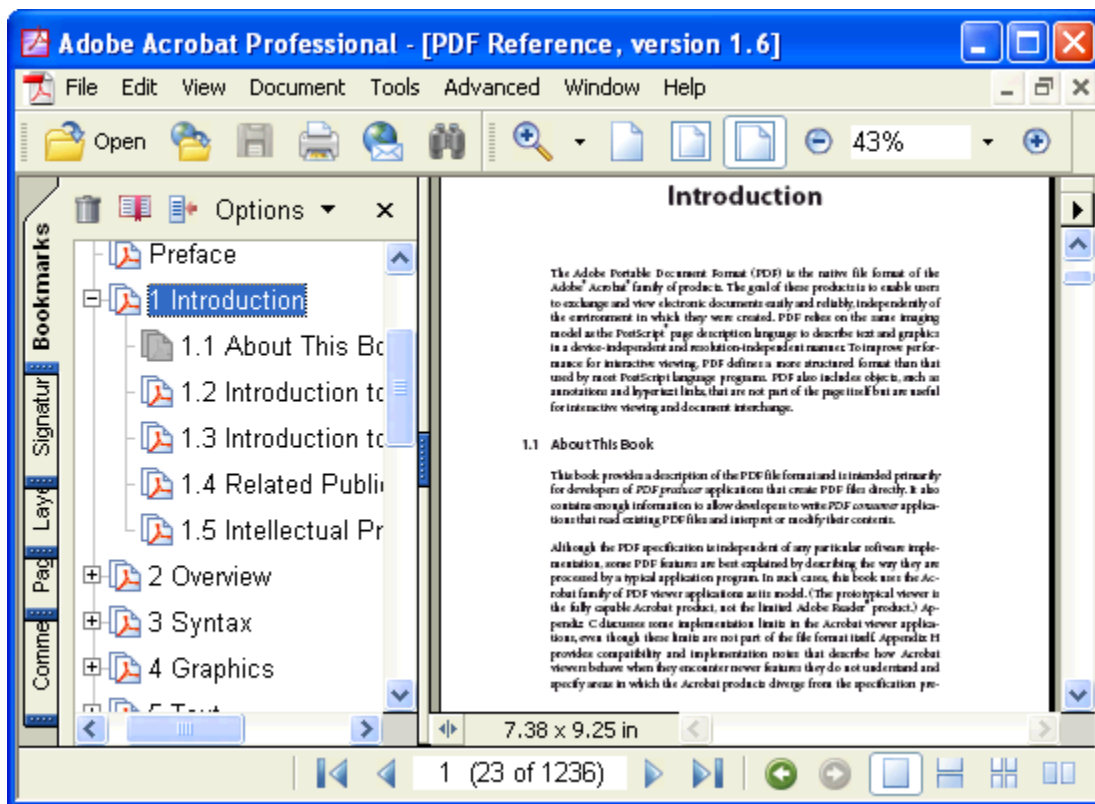
Note that the code uses the **C1PdfDocumentBase.Pages** property to get the page count. **Pages** is a collection based on the **ArrayList** class, and has methods that allow you to count and enumerate pages, as well as add and remove pages at specific positions. You can use the **Pages** collection to remove pages from certain locations in the document and re-insert them elsewhere.

Adding Bookmarks and Annotations

The following topics describe how to add bookmarks and annotations to a PDF document.

Adding Bookmarks to a PDF Document

When you open a PDF document using Adobe's Acrobat Reader application, you will notice that most long documents contain an outline structure that is displayed on a pane on the left of the reader. The outline makes it easy to browse through a document's structure and find specific topics. The picture below shows a PDF document with an outline:



The outline entries are called Bookmarks, and you can add them to your **PDF for .NET** documents using the **AddBookmark** method. The **AddBookmark** method takes three parameters: the title of the outline entry, the outline level, and the 'y' position of the entry on the current page (measured in points from the top of the page).

For example, the routine below adds a paragraph to a document and optionally marks it as a level-zero outline entry:

- Visual Basic

```

Private Function RenderParagraph(text As String, font As Font, rect As
RectangleF, rectPage As RectangleF, outline As Boolean) As RectangleF

    ' If it doesn't fit on this page, add a page break.
    rect.Height = _clpdf.MeasureString(text, font, rect.Width).Height
    If rect.Bottom > rectPage.Bottom Then
        _clpdf.NewPage()
        rect.Y = rectPage.Top
    End If

    ' Draw the string.
    _clpdf.DrawString(text, font, Brushes.Black, rect)

    ' Add headings to outline.
    If outline Then
        _clpdf.DrawLine(Pens.Black, rect.X, rect.Y, rect.Right, rect.Y)
        _clpdf.AddBookmark(text, 0, rect.Y)
    End If

    ' Update rectangle for next time.
    rect.Offset(0, rect.Height)
    Return rect
End Function

```

- **C#**

```

private RectangleF RenderParagraph(string text, Font font, RectangleF
rect, RectangleF rectPage, bool outline)
{
    // If it doesn't fit on this page, add a page break.
    rect.Height = _clpdf.MeasureString(text, font, rect.Width).Height;
    if (rect.Bottom > rectPage.Bottom)
    {
        _clpdf.NewPage();
        rect.Y = rectPage.Top;
    }

    // Draw the string.
    _clpdf.DrawString(text, font, Brushes.Black, rect);

    // Add headings to outline.
    if (outline)
    {
        _clpdf.DrawLine(Pens.Black, rect.X, rect.Y, rect.Right, rect.Y);
        _clpdf.AddBookmark(text, 0, rect.Y);
    }

    // Update rectangle for next time.
    rect.Offset(0, rect.Height);
    return rect;
}

```

Adding Links to a PDF Document

The PDF specification allows you to add several types of annotations to your documents. Annotations are often added by hand, as highlights and notes. But they can also be added programmatically. C1PdfDocument provides methods for adding hyperlinks, hyperlink targets, and file attachments to your PDF documents.

To add a hyperlink to your document, use the AddLink method. AddLink method takes two parameters: a string that specifies a *url* and a *RectangleF* that specifies the area on the current page that should behave as a link.

Note that the AddLink method does not add any visible content to the page, so you will usually need another command along with AddLink to specify some text or an image that the user can see. For example, the code below adds a string that says "Visit ComponentOne" and a link that takes the user to the ComponentOne home page:

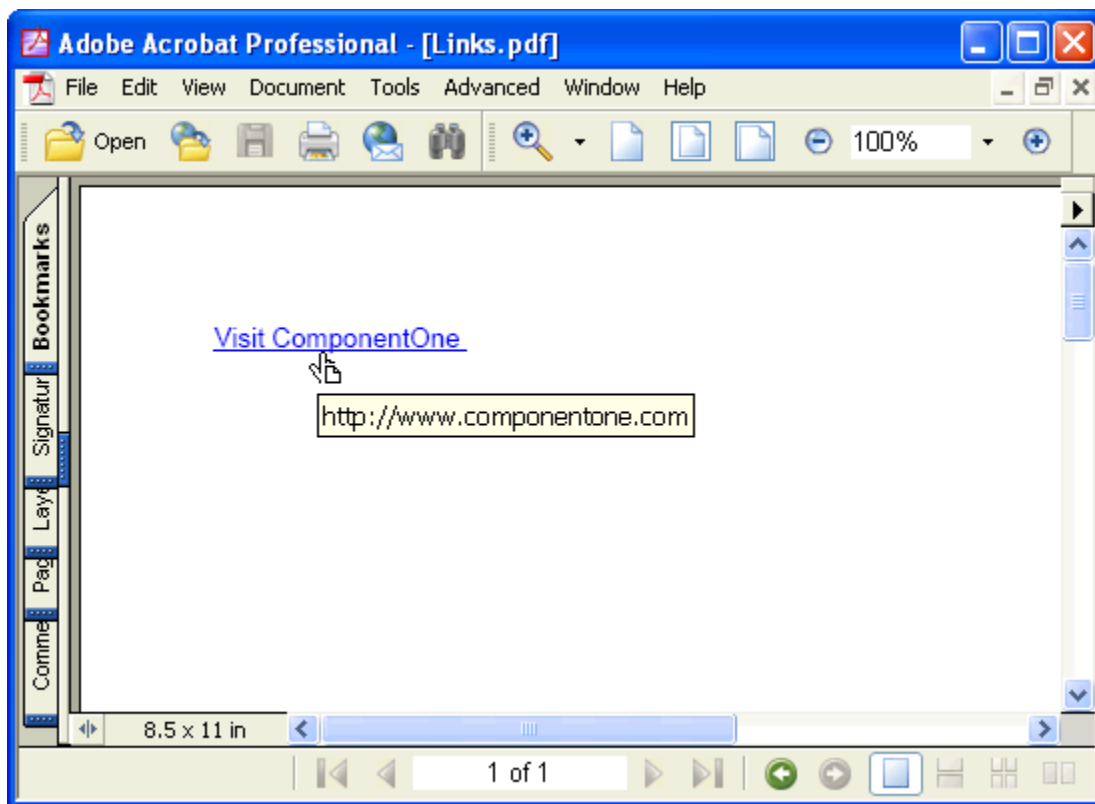
- Visual Basic

```
Dim rect As New RectangleF(50, 50, 100, 15)
Dim font As New Font("Arial", 10, FontStyle.Underline)
_clpdf.AddLink("http://www.componentone.com", rect)
_clpdf.DrawString("Visit ComponentOne", font, Brushes.Blue, rect)
```

- C#

```
RectangleF rect = new RectangleF(50, 50, 100, 15);
Font font = new Font("Arial", 10, FontStyle.Underline);
_clpdf.AddLink("http://www.componentone.com", rect);
_clpdf.DrawString("Visit ComponentOne", font, Brushes.Blue, rect);
```

Here is the resulting PDF document:



You can also add local links, which when clicked take the user to another location within the same PDF document. This type of link is useful when you want to implement some type of cross-referencing within the document, such as a table of contents or an index.

Local links are identical to regular hyperlinks, except for two things:

- The *url* parameter must start with a "#".

- You must specify the target location for the link using the AddTarget method. The AddTarget method takes the same parameters as AddLink, a string that specifies the name of the target and a rectangle that marks the area on the page that will be displayed when the user selects the link.

Attaching Files to a PDF Document

Adding file attachments to PDF files is often a useful feature. Attachments can contain any kind of file, including spreadsheets with detailed information that would clutter the main document, multimedia files with movies and sound, sample code, and so on.

Adding file attachments to your **PDF for .NET** documents is easy. All you have to do is call the AddAttachment method and specify which file you want to attach, what area of the page should contain the attachment, and optionally, the appearance of the attachment.

For example, the following code attaches all files in the application directory to the PDF document:

- Visual Basic

```
Dim rect As New RectangleF(100, 100, 60, 10)
Dim font As New Font("Arial", 9)

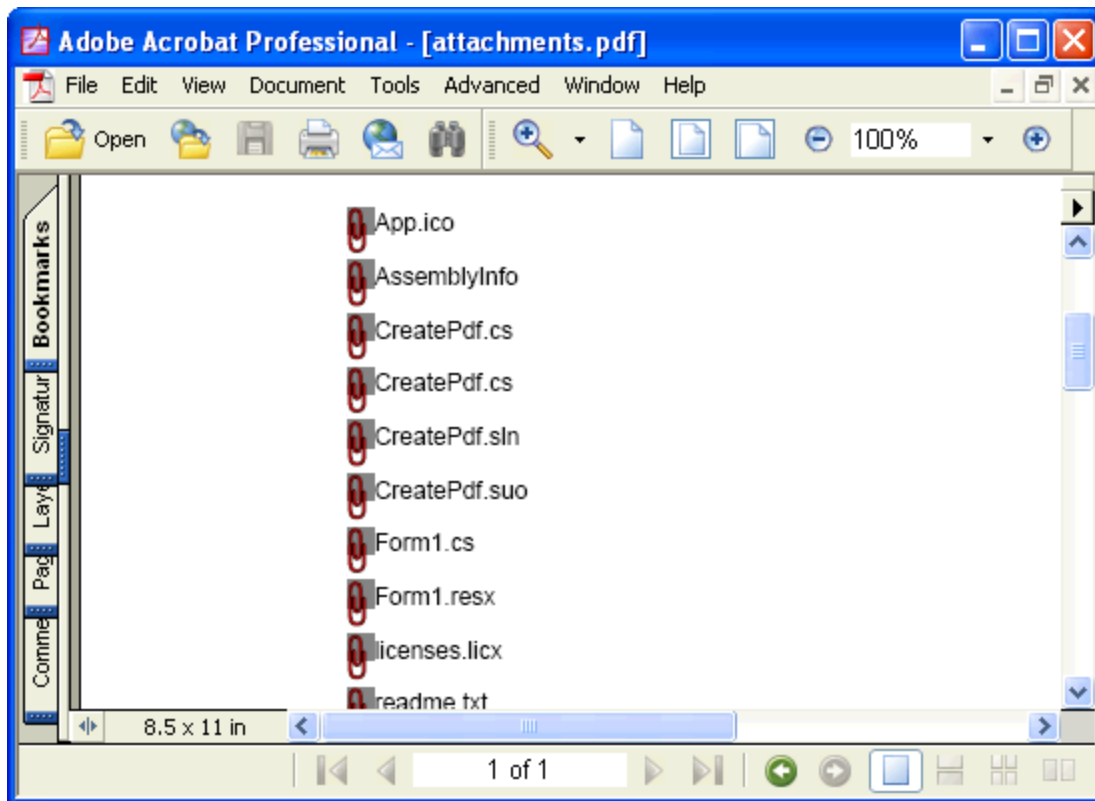
' Attach some files.
Dim path As String = "c:\temp\files"
Dim file As String
For Each file In Directory.GetFiles(path)
    Dim width As Single = rect.Width
    rect.Width = rect.Height
    _clpdf.FillRectangle(Brushes.Gray, rect)
    _clpdf.AddAttachment(file, rect)
    rect.Width = width
    rect.X += rect.Height
    _clpdf.DrawString(Path.GetFileName(file), font, Brushes.Black, rect)
    rect.X -= rect.Height
    rect.Y += 2 * rect.Height
Next file
```

- C#

```
RectangleF rect = new RectangleF(100, 100, 60, 10);
Font font = new Font("Arial", 9);

// Attach some files.
string path = @"c:\temp\files";
string[] files = Directory.GetFiles(path);
foreach (string file in files)
{
    float width = rect.Width;
    rect.Width = rect.Height;
    _clpdf.FillRectangle(Brushes.Gray, rect);
    _clpdf.AddAttachment(file, rect);
    rect.Width = width;
    rect.X += rect.Height;
    _clpdf.DrawString(Path.GetFileName(file), font, Brushes.Black, rect);
    rect.X -= rect.Height;
    rect.Y += 2 * rect.Height;
}
```

Here's what the PDF document looks like in Adobe's Acrobat Reader:



The attachments are displayed as icons (you can select from four predefined icons in the AttachmentIconEnum enumeration and you can also select the icon color). When the user moves the mouse over the attachment, the file name is displayed and the cursor changes to indicate there is an attachment available. The user can then right-click the attachment name to open the attachment or save it to a file.

Applying Security and Permissions

By default, anyone can open, copy, print, and edit PDF files. If your PDF documents contain sensitive information, however, you can encrypt them so that only authorized users can access it.

There is a separate password for the owner of the document and for all other users. The user's access can be selectively restricted to allow only certain operations, such as viewing, printing, or editing the document.

PDF for .NET provides a **C1PdfDocumentBase.Security** property that returns a PdfSecurity object. This object has properties that allow you to specify the **owner password** (required to change passwords and permissions for the document) and the **user password** (required to open the document). Additionally, the PdfSecurity object allows you to specify what permissions a regular user should have. For example you may allow users to see the document but not to print or edit it.

To use the **PDF for .NET** security features, simply set the passwords before you save the document. For example:

- Visual Basic

```
' Create the document as usual.
CreateDoc()

' Set passwords.
_clpdf.Security.OwnerPassword = "2mds%dffgd"
_clpdf.Security.UserPassword = "anyone"
_clpdf.Security.AllowEditAnnotations = False
_clpdf.Security.AllowEditContent = False
```

```

_clpdf.Security.AllowPrint = False

' Save the encrypted document.
_clpdf.Save("")

```

- C#

```

// Create the document as usual.
CreateDoc();

// Set passwords.
_clpdf.Security.OwnerPassword = "2mds%dfgd";
_clpdf.Security.UserPassword = "anyone";
_clpdf.Security.AllowEditAnnotations = false;
_clpdf.Security.AllowEditContent = false;
_clpdf.Security.AllowPrint = false;

// Save the encrypted document.
_clpdf.Save(@"c:\reports\secure.pdf");

```

Note that you can specify permissions and set only the owner password, leaving the user password empty. In this case, anyone will be allowed to open the document, but only the owner will be allowed to change the permissions.

Note also that the encryption scheme used by **PDF for .NET** is public and is not 100% secure. There are ways to crack encrypted PDF documents. The security provided is adequate to protect your documents from most casual attacks, but if your data is truly sensitive you should not rely on PDF encryption alone.

Using Metafiles

PDF for .NET makes it very easy to create documents, mainly because the object model mimics the well-known .NET **Graphics** model. However, not all methods available in the **Graphics** class are available in **PDF for .NET**. Plus, you may have existing code that draws to a **Graphics** object and that you do not want to rewrite even if most methods are very similar.

In these cases, you can reuse your existing .NET code by sending the drawing commands to a **Metafile**, then rendering the **Metafile** into **PDF for .NET** using the `DrawImage` command. This method allows you to expose any graphics you create as images or as PDF documents.

For example, suppose you have an application that generates documents using the **PrintDocument** pattern of drawing each page into a **Graphics** object. You could then use the same methods to create a collection of metafiles, one per page, and then convert the list into a PDF document using the following code:

- Visual Basic

```

' Get the document as a list of Metafiles, one per page.
Dim pages As ArrayList = GetMetafiles()

' Loop through the pages and create a PDF document.
_clpdf.Clear()
Dim i As Integer
for i = 0 to pages.Count - 1

    ' Get ith page.
    Dim page As Metafile = CType(Metafile.FromFile(pages[i]), Metafile)
    If Not (page Is Nothing) Then

        ' Calculate the page size.
        Dim sz As SizeF = page.PhysicalDimension
        sz.Width = Math.Round(sz.Width * 72.0F / 2540.0F, 2)
        sz.Height = Math.Round(sz.Height * 72.0F / 2540.0F, 2)
    End If
end for

```

```

        ' Add a page and set the size.
        If i > 0 Then
            _clpdf.NewPage()
        End If
        _clpdf.PageSize = sz

        ' Draw the page into the PDF document.
        _clpdf.DrawImage(page, _clpdf.PageRectangle)
    End If
Next

' Save to file.
_clpdf.Save("c:\temp\mydoc.pdf")

```

- **C#**

```

// Get the document as a list of Metafiles, one per page.
ArrayList pages = GetMetafiles();

// Loop through the pages and create a PDF document.
_clpdf.Clear();
for (int i = 0; i < pages.Count; i++)
{
    // Get ith page.
    Metafile page = (Metafile)Metafile.FromFile(pages[i]);
    if (page == null)
    {
        continue;
    }

    // Calculate the page size.
   .SizeF sz = page.PhysicalDimension;
    sz.Width = (float)Math.Round(sz.Width * 72f / 2540f, 2);
    sz.Height = (float)Math.Round(sz.Height * 72f / 2540f, 2);

    // Add a page and set the size.
    if (i > 0) _clpdf.NewPage();
    _clpdf.PageSize = sz;

    // Draw the page into the PDF document.
    _clpdf.DrawImage(page, _clpdf.PageRectangle);
}

// Save to file.
_clpdf.Save(@"c:\temp\mydoc.pdf");

```

The code gets each metafile on the list, calculates its size in points (each page could have a different size), then draws the metafile into the page. The metafiles could be generated by a reporting engine, drawing or charting program, or any application that can create metafile images.

PDF for .NET also supports EMF+ metafiles. [According to Microsoft](#), EMF+ is an extension to EMF that allows GDI+ records to be stored. Previously, all metafiles were transformed to EFM Only. EMF+ metafiles are no longer transformed; therefore, they do not lose their context due to transformation.

Rendering RTF Text

Another powerful feature of **ComponentOne PDF for .NET** is the ability to render rich text format (RTF) text using the `DrawStringRtf` method. This is an extremely useful feature mainly because there are so many tools for creating RTF, and it's easy to create templates in RTF and have your application customize, then render the RTF strings. This makes it easy to create richly formatted documents practically without writing any code.

For example, the code below uses an RTF template created with WordPad. It scans a directory and builds an RTF string based on the template, then renders the RTF using the `DrawStringRtf` method.

- Visual Basic

```
' Get RTF templates.
Dim rtfHdr As String = "" & _
    "{\rtf1\ansi\ansicpg1252\deff0\deflang1033" & _
    "{\fonttbl{\f0\fswiss\fcharset0 " & _
    "Arial;}{\f1\froman\fprq2\fcharset0 Book Antiqua;}}}" & _
    "{\colortbl ;\red0\green0\blue0;}" & _
    "\viewkind4\uc1\pard\f0\fs20\par" & _
    "\pard\tx1440\tx2880\tx4320\tx5760\cf1\b\f1\fs24 " & _
    "Directory Report created on <<TODAY>>\par" & _
    "\ul\par Name\tab Extension\tab Size\tab Date\tab " & _
    "Attributes\par"
Dim rtfEntry As String = "" & _
    "\cf0\ulnone\b0\f0\fs16 <<NAME>>\tab " & _
    "<<EXT>>\tab <<SIZE>>\tab <<DATE>>\tab <<ATTS>>\par"

' Build RTF string.
Dim sb As New StringBuilder()
sb.Append(rtfHdr.Replace("<<TODAY>>", DateTime.Today.ToShortDateString()))
Dim file As String
For Each file in Directory.GetFiles(@"c:\winnt", "*.bmp")
    Dim s As String = rtfEntry
    Dim fi As FileInfo = New FileInfo(file)
    s = s.Replace("<<NAME>>", Path.GetFileNameWithoutExtension(file))
    s = s.Replace("<<EXT>>", fi.Extension)
    s = s.Replace("<<SIZE>>", String.Format("{0:#,##0}", fi.Length))
    s = s.Replace("<<DATE>>", fi.LastWriteTime.ToShortDateString())
    s = s.Replace("<<ATTS>>", fi.Attributes.ToString())
    sb.Append(s)
Next
sb.Append("{}")

' Render it.
Dim pdf As New ClPdfDocument()
Dim rect As RectangleF = _clpdf.PageRectangle
rect.Inflate(-72, -72)
_clpdf.DrawStringRtf(sb.ToString(), Font, Brushes.Black, rect)

' Save.
_clpdf.Save("c:\temp\dir.pdf")
```

- C#

```
// Get RTF templates.
string rtfHdr = @"{\rtf1\ansi\ansicpg1252\deff0\deflang1033" +
    @"{\fonttbl{\f0\fswiss\fcharset0 " +
    @"Arial;}{\f1\froman\fprq2\fcharset0 Book Antiqua;}}}" +
    @"{\colortbl ;\red0\green0\blue0;}" +
    @" \viewkind4\uc1\pard\f0\fs20\par" +
```

```

        @"\pard\tx1440\tx2880\tx4320\tx5760\cf1\b\f1\fs24 " +
        @"Directory Report created on <<TODAY>>\par" +
        @"\ul\par Name\tab Extension\tab Size\tab Date\tab " +
        @"Attributes\par";
string rtfEntry = @"\cf0\ulnone\b0\f0\fs16 <<NAME>>\tab " +
        @"\<<EXT>>\tab <<SIZE>>\tab <<DATE>>\tab <<ATTS>>\par";

// Build RTF string.
StringBuilder sb = new StringBuilder();
sb.Append(rtfHdr.Replace("<<TODAY>>",
    DateTime.Today.ToShortDateString()));
foreach (string file in Directory.GetFiles(@"c:\winnt", "*.bmp"))
{
    string s = rtfEntry;
    FileInfo fi = new FileInfo(file);
    s = s.Replace("<<NAME>>", Path.GetFileNameWithoutExtension(file));
    s = s.Replace("<<EXT>>", fi.Extension);
    s = s.Replace("<<SIZE>>", string.Format("{0:#,##0}", fi.Length));
    s = s.Replace("<<DATE>>", fi.LastWriteTime.ToShortDateString());
    s = s.Replace("<<ATTS>>", fi.Attributes.ToString());
    sb.Append(s);
}
sb.Append("}");

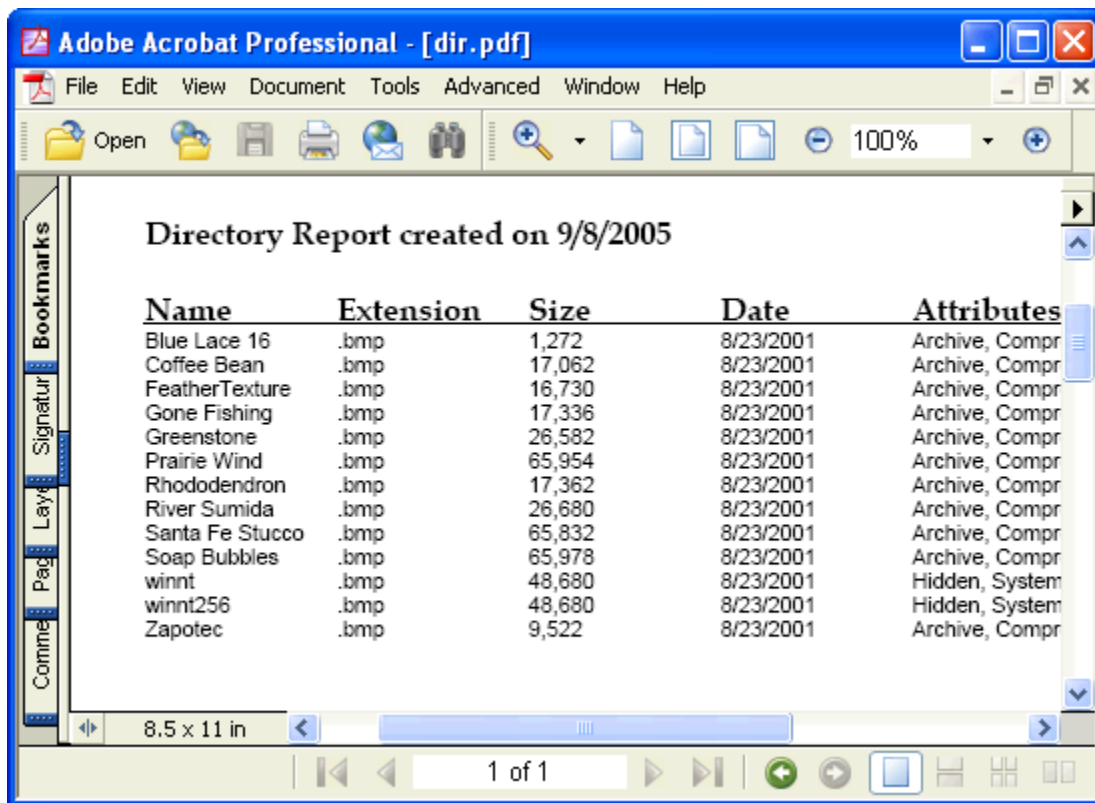
// Render it.
C1.C1Pdf.C1PdfDocument pdf = new C1.C1Pdf.C1PdfDocument();
RectangleF rect = pdf.PageRectangle;
rect.Inflate(-72, -72);
_clpdf.DrawStringRtf(sb.ToString(), Font, Brushes.Black, rect);

// Save.
_clpdf.Save(@"c:\temp\dir.pdf");

```

The code is very simple. The only complicated part is the RTF definition, but that was copied and pasted from an RTF file created with WordPad. The RTF template string contains tags (for example, "<<NAME>>") that are then replaced with the actual contents.

When the RTF string is built, it is rendered into the PDF document using the DrawStringRtf command. The result looks like this:



PDF for .NET 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 the desktop, click the **Start** button and then click **All Programs | ComponentOne | Studio for WinForms (or Studio for ASP.NET) | Samples | Pdf Samples**.

- Visual Basic Samples 

Sample	Description
PdfForm	Creates a PDF version of a paper form. This sample loads an empty fax cover form (created in Visio and saved as an enhanced metafile), and then uses the DrawString method to fill in the fields based on information you enter on the main form.
PrintDocument	Renders a PrintDocument into a PDF file. This sample uses the C1PdfDocument control.

- C# Samples 

Sample	Description
CharTable	Creates a PDF document containing a character table. The sample shows a simple

	way to create tables. It creates a rectangle and then moves it across to create each cell in a row. At the end of the row, it moves the rectangle back to the left and one row down. The sample also shows all ASCII characters rendered in PDF.
Charts	Creates a PDF document containing charts created with the C1Chart control. The sample creates metafiles for several 2 and 3D charts, and then renders the metafiles into a PDF document using the DrawImage method. The charts are rendered as a sequence of graphics commands (they are not translated into bitmaps). As a result, the charts are rendered in a high-quality, resolution-independent way. This sample uses the C1PdfDocument, C1Chart and C1Chart3D controls.
CreatePdf	Creates several PDF documents that show different aspects of the C1Pdf library. This sample uses the C1PdfDocument control.
FlowHtml	This sample shows how to render HTML into PDF documents and uses the C1PdfDocument control.
FlowRtf	This sample shows how to render Rtf text that flows onto several pages. This sample uses the C1PdfDocument control.
ManualSamples	Contains a variety of examples, such as how to render a long string into a short string, how to add links, and how to draw lines. Each example is described in the PDF created. This sample uses the C1PdfDocument control.
Metafile	Renders several metafiles into one PDF document. This sample uses the C1PdfDocument control.
MixedOrientation	This sample renders two C1Reports (one portrait, one landscape) and then merges them into a single C1PdfDocument using the C1Report.PageImages property and the DrawImage method.
PdfAcroform	This sample shows how to create a PDF AcroForm with a variety of fields and uses the C1PdfDocument control.
PdfAnnotations	This sample creates a PDF document with a form containing several notes, including rectangle, circle, line and text box. This sample uses the C1PdfDocument control.
PdfChart	The sample creates a pie chart, converts it into a metafile, and then renders the metafile into a PDF document using the DrawImage method. The chart is rendered as a sequence of graphics commands (it is not translated into a bitmap). As a result, the chart is rendered in a high-quality, resolution-independent way. This sample uses C1Chart .
PdfPrintPreview	This sample implements an enhanced PrintPreviewDialog with PDF support. The standard PrintPreviewDialog class is handy and easy to use, but it is limited and looks outdated. This samples shows an updated version of the PrintPreviewDialog class using a drop-in replacement class called CoolPrintPreviewDialog. This class adds enhancements, such as PDF output support, it shows preview pages as the document is generated, it shows the current page number and total number of pages, and more. This sample uses the C1PdfDocument control.
PdfThreaded	This sample allows you to select the number of threads to create. Each thread generates a PDF file. This sample uses the C1PdfDocument control.
Security	Creates a PDF document with password protection and permissions. This sample uses the C1PdfDocument control.

PDF for .NET Task-Based Help

The task-based help section assumes that you are familiar with programming in the Visual Studio .NET environment, and know how to use the **PDF for .NET** control in general. If you are a novice to the PDF for .NET, please see [Using ComponentOne PDF for .NET](#) (page 14) first.

Each topic provides a solution for specific tasks using the **PDF for .NET** product. By following the steps outlined in the help, you will be able to utilize the features of **ComponentOne PDF for .NET**.

Each task-based help topic also assumes that you have created a new .NET project. For additional information on this topic, see [Creating a .NET Project](#) (page 10).

Setting Document Properties

Setting the document properties for a PDF document allows you to customize the **Title**, **Author**, **Subject**, **Keywords**, and **Application** under **Description**, as well as **PDF Producer** under **PDF Information**. To set the document properties, enter information in the Title, Author, Subject, Keywords, Creator, and Producer properties either in the designer or in code.

The image shows a screenshot of a software interface for setting PDF document properties. It is divided into two main sections: "Description" and "PDF Information".

Description Section:

- Title:** PDF Creation
- Author:** I. Author
- Subject:** PDF
- Keywords:** PDF, creating PDF, PDF files, eBooks
- Created:** 5/19/2006 5:48:40 AM
- Modified:** 5/19/2006 5:48:40 AM
- Application:** My New Application

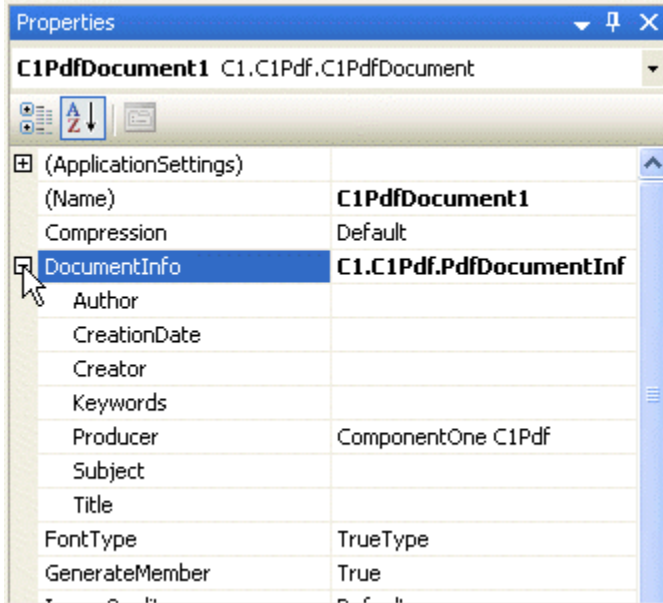
PDF Information Section:

- PDF Producer:** PDF Creation Industries
- PDF Version:** 1.3 (Acrobat 4.x)
- Path:** C:\temp\mydoc.pdf
- File Size:** 0.89 KB (916 Bytes)
- Page Size:** 8.5 x 11 in
- Number of Pages:** 1
- Tagged PDF:** No
- Fast Web View:** No

Note: To set the **Application** under **Description**, use the Creator property. The document properties appear in the **Document Properties** dialog box in the PDF document.

In the Designer

In the Properties window, locate the **C1PdfDocumentBase.DocumentInfo** property and expand the property node.



Type in the information for a property and press ENTER when finished to set the property.

In Code

Add the following code to the **Form_Load** event:

1. Create the C1PdfDocument object:

- Visual Basic

```
' Create the C1PdfDocument object.  
Dim pdf As New C1.C1Pdf.C1PdfDocument()
```

- C#

```
// Create the C1PdfDocument object.  
C1.C1Pdf.C1PdfDocument pdf = new C1.C1Pdf.C1PdfDocument();
```

2. Set the document properties:

To set the Title property, add the following code:

- Visual Basic

```
pdf.DocumentInfo.Title = "PDF Creation"
```

- C#

```
pdf.DocumentInfo.Title = "PDF Creation";
```

To set the Author property, add the following code:

- Visual Basic

```
pdf.DocumentInfo.Author = "I. Author"
```

- C#

```
pdf.DocumentInfo.Author = "I. Author";
```

To set the Subject property, add the following code:

- Visual Basic

```
pdf.DocumentInfo.Subject = "PDF"
```

- C#

```
pdf.DocumentInfo.Subject = "PDF";
```

To set the Keywords property, add the following code:

- Visual Basic

```
pdf.DocumentInfo.Keywords = "PDF, creating PDF, PDF files, eBooks"
```

- C#

```
pdf.DocumentInfo.Keywords = "PDF, creating PDF, PDF files, eBooks";
```

To set the Creator property, add the following code:

- Visual Basic

```
pdf.DocumentInfo.Creator = "My New Application"
```

- C#

```
pdf.DocumentInfo.Creator = "My New Application";
```

To set the Producer property, add the following code. The default is "ComponentOne C1Pdf".

- Visual Basic

```
pdf.DocumentInfo.Producer = "PDF Creation Industries"
```

- C#

```
pdf.DocumentInfo.Producer = "PDF Creation Industries";
```

3. Save the document to a file:

- Visual Basic

```
' Save the document to a file.  
pdf.Save("c:\mydoc.pdf")
```

- C#

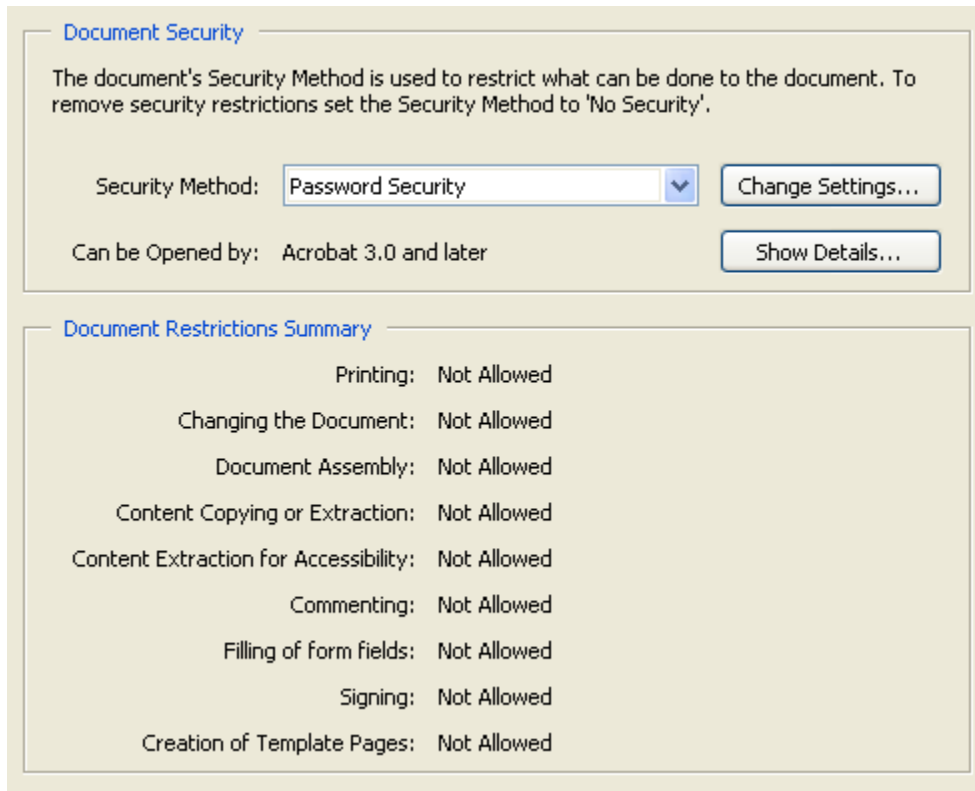
```
// Save the document to a file.  
pdf.Save(@"c:\mydoc.pdf");
```

Setting Document Restrictions

Setting document restrictions allows you to customize if a user can copy content, edit annotations, edit content, or print the PDF file. To enable document restrictions, set the following properties to **False** either in the designer or in code:

- AllowCopyContent
- AllowEditAnnotations
- AllowEditContent
- AllowPrint

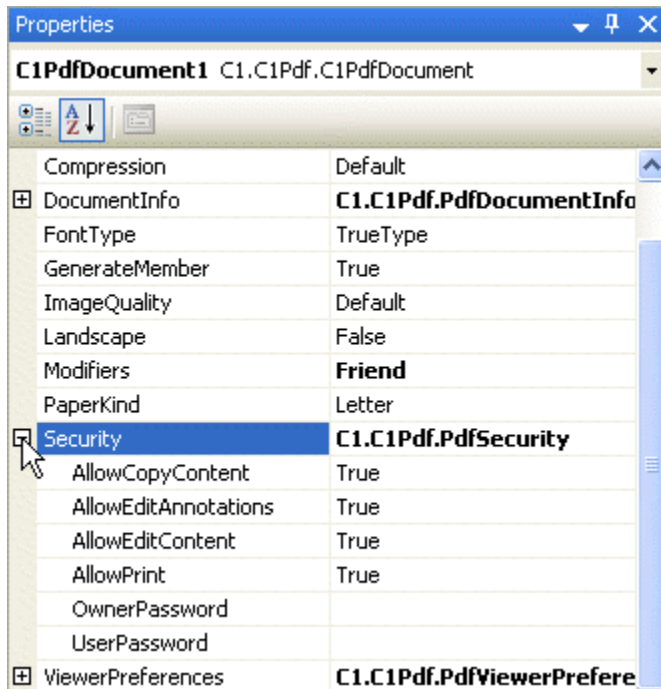
For more information on setting passwords and permissions, see [Applying Security and Permissions](#) (page 30).



Note: The document restrictions appear in the **Document Properties** dialog box in the PDF document.

In the Designer

1. In the Properties window, locate the **C1PdfDocumentBase.Security** property and expand the property node.



2. Set the OwnerPassword property by typing in a password and pressing ENTER when finished.
3. Set the AllowCopyContent, AllowEditAnnotations, AllowEditContent, and AllowPrint properties to **False** to not allow copying, editing, and printing.

In Code

Add the following code to the **Form_Load** event.

1. Create the C1PdfDocument object:

- Visual Basic

```
' Create the C1PdfDocument object.
Dim pdf As New C1.C1Pdf.C1PdfDocument()
```

- C#

```
// Create the C1PdfDocument object.
C1.C1Pdf.C1PdfDocument pdf = new C1.C1Pdf.C1PdfDocument();
```

2. Add the following code to set the OwnerPassword property:

- Visual Basic

```
pdf.Security.OwnerPassword = "Password"
```

- C#

```
pdf.Security.OwnerPassword = "Password";
```

3. Add the following code to set the AllowCopyContent, AllowEditAnnotations, AllowEditContent, and AllowPrint properties to **False** to not allow copying, editing, and printing:

- Visual Basic

```
pdf.Security.AllowCopyContent = False
pdf.Security.AllowEditAnnotations = False
pdf.Security.AllowEditContent = False
pdf.Security.AllowPrint = False
```

- C#

```
pdf.Security.AllowCopyContent = false;
pdf.Security.AllowEditAnnotations = false;
pdf.Security.AllowEditContent = false;
pdf.Security.AllowPrint = false;
```

4. Save the document to a file:

- Visual Basic

```
' Save the document to a file.
pdf.Save("c:\mydoc.pdf")
```

- C#

```
// Save the document to a file.
pdf.Save(@"c:\mydoc.pdf");
```

Setting Paper Size

To set the paper size, use the PaperKind property. This property can be set either in the designer or in code. For more information on paper size, see [Creating Pages and Overlays](#) (page 23).

In the Designer

Locate the PaperKind property in the Properties window and choose the new paper size.

In Code

Add the following code to the **Form_Load** event to set the PaperKind property to Legal size paper:

- Visual Basic

```
' Create the C1PdfDocument object.
Dim pdf As New C1.C1Pdf.C1PdfDocument()

' Use Legal size paper.
pdf.PaperKind = Printing.PaperKind.Legal

' Save the document to a file.
pdf.Save("c:\mydoc.pdf")
```

- C#

```
// Create the C1PdfDocument object.
C1.C1Pdf.C1PdfDocument pdf = new C1.C1Pdf.C1PdfDocument();

// Use Legal size paper.
pdf.PaperKind = Printing.PaperKind.Legal;

// Save the document to a file.
pdf.Save(@"c:\mydoc.pdf");
```

Setting Landscape Page Orientation

To set the page orientation, use the Landscape property. This property can be set either in the designer or in code. For more information on page settings, see [Creating Pages and Overlays](#) (page 23).

In the Designer

Locate the Landscape property in the Properties window and set it to **True**.

In Code

Add the following code to the **Form_Load** event to set the Landscape property:

- Visual Basic

```
' Create the C1PdfDocument object.
Dim pdf As New C1.C1Pdf.C1PdfDocument()

' Use landscape page orientation.
pdf.Landscape = True

' Save the document to a file.
pdf.Save("c:\mydoc.pdf")
```

- C#

```
// Create the C1PdfDocument object.
C1.C1Pdf.C1PdfDocument pdf = new C1.C1Pdf.C1PdfDocument();

// Use landscape page orientation.
pdf.Landscape = true;

// Save the document to a file.
pdf.Save(@"c:\mydoc.pdf");
```

Setting Image Quality

To set the image quality in the PDF document, use the **C1PdfDocumentBase.ImageQuality** property. This property can be set either in the designer or in code. For more information on images and image quality, see [Adding Images](#) (page 20).

In the Designer

Locate the **C1PdfDocumentBase.ImageQuality** property in the Properties window and set it to one of the following ImageQualityEnum settings: Low, Medium, Default, or High.

In Code

Add the following code to the **Form_Load** event to set the **C1PdfDocumentBase.ImageQuality** property to high quality images:

- Visual Basic

```
' Create the C1PdfDocument object.
Dim pdf As New C1.C1Pdf.C1PdfDocument()

' Use high quality images.
pdf.ImageQuality = C1.C1Pdf.ImageQualityEnum.High

' Save the document to a file.
pdf.Save("c:\mydoc.pdf")
```

- C#

```
// Create the C1PdfDocument object.
C1.C1Pdf.C1PdfDocument pdf = new C1.C1Pdf.C1PdfDocument();

// Use high quality images.
pdf.ImageQuality = C1.C1Pdf.ImageQualityEnum.High;

// Save the document to a file.
pdf.Save(@"c:\mydoc.pdf");
```

Printing PDF Files

The following code allows you to add print functionality to **PDF for .NET**. In this example, the code was added to the **Button1_Click** event. Note that you must use the **Imports System.Diagnostics** (Visual Basic) or **using System.Diagnostics** (C#) directive at the top of your form in order for the following code to work.

- Visual Basic

```
Process.Start("C:\Program Files\Adobe\Acrobat 5.0\Acrobat\Acrobat.exe",  
"/p /h " & "c:\hello world.pdf")
```

- C#

```
Process.Start(@"C:\Program Files\Adobe\Acrobat 5.0\Acrobat\Acrobat.exe",  
"/p /h " + @"c:\hello world.pdf");
```

Writing PDF for .NET Content in Web Applications

This topic provides an example of how to write **PDF for .NET** content directly into a Web page's output buffer with no temporary files.

1. In your Web application project, select **View | Component Designer**.
2. Add the C1PdfDocument control to your form.
3. Select **View | Code**, and add the following directives to your project:

- Visual Basic

```
Imports C1.C1Pdf  
Imports System.IO
```

- C#

```
using C1.C1Pdf;  
using System.IO;
```

4. Then add the following code:

- Visual Basic

```
Public Sub RenderPdf(ByVal doc As C1PdfDocument)  
    ' Render PDF document into memory-based PDF stream.  
    Dim ms As New MemoryStream()  
    doc.Save(ms)  
  
    ' Get response object.  
    Dim length As Integer  
    Dim rsp As HttpResponse = Me.Page.Response  
  
    ' Clear it.  
    rsp.Clear()  
  
    ' Write PDF stream into response buffer.  
    rsp.ContentType = "Application/pdf"  
    length = CInt(ms.Length)  
    rsp.OutputStream.Write(ms.GetBuffer(), 0, length)  
  
    ' Finish.  
    rsp.Flush()  
  
    ' This is required!  
    rsp.SuppressContent = True  
End Sub
```

- C#

```
protected void RenderPDF(C1PdfDocument doc)
{
    // Render PDF document into memory-based PDF stream.
    MemoryStream ms = new MemoryStream();
    doc.Save(ms);

    // Get response object.
    int length;
    HttpResponse rsp = this.Page.Response;

    // Clear it.
    rsp.Clear();

    // Write PDF stream into response buffer.
    rsp.ContentType = "Application/pdf";
    length = (int)ms.Length;
    rsp.OutputStream.Write(ms.GetBuffer(), 0, length);

    // Finish.
    rsp.Flush();

    // This is required.
    rsp.SuppressContent = true;
}
```

Using PDF for .NET with Other ComponentOne Tools

The following topics discuss using the **PDF for .NET** control with other development tools included with the ComponentOne Studios.

Adding ComponentOne FlexGrid for WinForms Data into PDF for .NET

To add **FlexGrid for WinForms** data into **PDF for .NET**, use **C1FlexGrid's CreateImage** method and render the images using the **DrawImage** method. The images created by the grid are metafiles.



Sample Project Available

For a complete sample using this method, download the PdfExport sample located on <http://our.componentone.com/samples/>.

Creating a PDF from a Report Rendered by VSPrinter

To convert documents created with the **VSPrinter** ActiveX control to PDF, you can retrieve the page images using the **VSPrinter.Picture** property, then render the images into **PDF for .NET** using the **DrawImage** method.

The code below creates a VSPrinter document and then converts it into PDF:

- Visual Basic

```
' Initialize PDF document.
_pdff.Clear()

' Create a VSPrinter document (or load it from an existing document).
.StartDoc()
For i = 1 To 1000
    .Paragraph = "This is paragraph " & i.ToString()
Next
```

```

.EndDoc()

' Scan the document page by page (1-based).
For i = 1 To .PageCount

    ' Set the current preview page.
    .PreviewPage = i

    ' Retrieve the image.
    Dim img As Image = .Picture

    ' Save image into PDF document.
    If i > 1 Then
        _pdf.NewPage()
    End If
    _pdf.DrawImage(img, Me._pdf.PageRectangle)
Next

' Return preview to first page.
.PreviewPage = 0

```

- **C#**

```

// Initialize PDF document.
_pdff.Clear();

// Create a VSPrinter document (or load it from an existing document).
.StartDoc();
for (i = 1; i <= 1000; i++)
{
    .Paragraph = "This is a paragraph " + i.ToString() ;
}
.EndDoc();

// Scan the document page by page (1-based).
for (i = 1; i <= .PageCount; i++)
{

    // Set the current preview page.
    .PreviewPage = i;

    // Retrieve the image.
    Image img = .Picture;

    // Save the image into PDF document.
    if (i > 1)
    {
        _pdff.NewPage();
    }
    _pdff.DrawImage(img, this._pdff.PageRectangle);
}

// Return preview to first page.
.PreviewPage = 0;

```

Removing PDF Bookmarks from a C1Report Document

You can remove a PDF bookmark from a **C1Report** document by completing one of the following tasks:

- Set the OutlineRootLevel property to -1.
OR
- Create a PDF filter and disable the outline.

The following code shows both alternatives.

- Visual Basic

```
' Set the OutlineRootLevel to -1.  
Me.C1Report1.OutlineRootLevel = - 1  
Me.C1Report1.RenderToFile("report.pdf", C1.C1Report.FileFormatEnum.PDF)  
  
' OR: Create a PDF filter and disable the outline.  
Dim f As New C1.C1Report.PdfFilter("report.pdf")  
f.Outline = False  
Me.C1Report1.RenderToFilter(f)
```

- C#

```
// Set the OutlineRootLevel to -1.  
this.c1Report1.OutlineRootLevel = -1;  
this.c1Report1.RenderToFile("report.pdf", C1.C1Report.FileFormatEnum.PDF);  
  
// OR: Create a PDF filter and disable the outline.  
C1.C1Report.PdfFilter f = new C1.C1Report.PdfFilter("report.pdf");  
f.Outline = false;  
this.c1Report1.RenderToFilter(f);
```