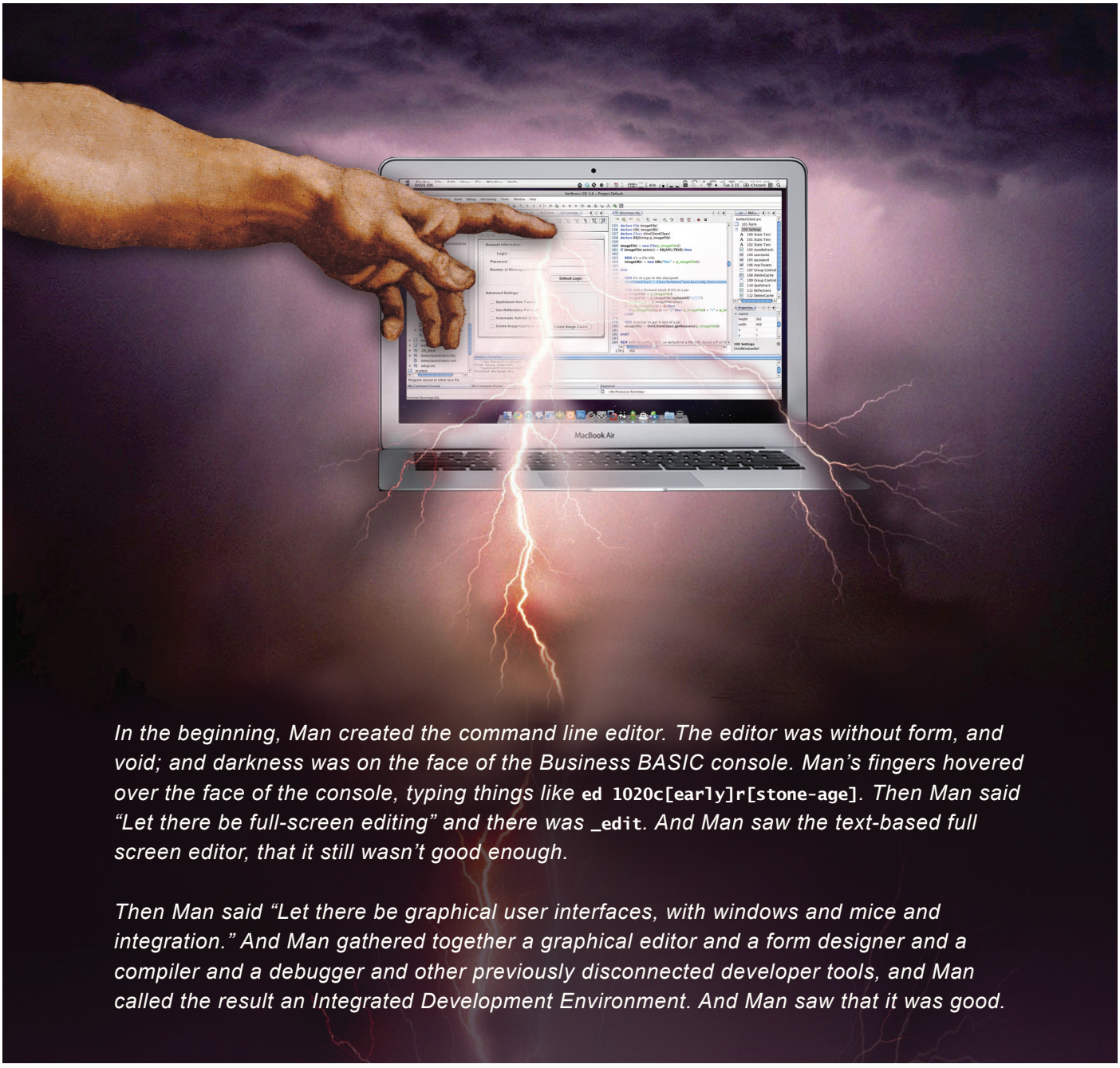
<<< previous article



In the beginning, Man created the command line editor. The editor was without form, and void; and darkness was on the face of the Business BASIC console. Man's fingers hovered over the face of the console, typing things like `ed 1020c[early]r[stone-age]`. *Then Man said "Let there be full-screen editing" and there was* `_edit`. *And Man saw the text-based full screen editor, that it still wasn't good enough.*

Then Man said "Let there be graphical user interfaces, with windows and mice and integration." And Man gathered together a graphical editor and a form designer and a compiler and a debugger and other previously disconnected developer tools, and Man called the result an Integrated Development Environment. And Man saw that it was good.

BBj IDE in the Beginning - Then and Now

Man is not yet finished with creation, of course. At BASIS we are constantly engaged in making improvements to our own Integrated Development Environment (IDE). Even though perfection is a long way off, we'd like to show you the latest iteration of the BASIS IDE built on NetBeans from Sun Microsystems (now Oracle) and extended by BASIS with plug-in modules to give it the capability of developing Business BASIC applications.

By Mike Phelps
Software Programmer

Read on for the IDE's most helpful features, what it takes to get it running, and some insider tips. If you have already tried the IDE, you may pick up some useful information you hadn't heard before. If you haven't tried the IDE, you may find the time has come to give it a try!

Part I - Configuring the IDE for Your Development Project

To install the BASIS IDE, you need a Java 1.6 JDK and the latest version of BBj® (as of BBj 10, there is no longer any need to run the IDE with a Java 1.5 JDK). Running the IDE with only a JRE (Java >>

Runtime Environment) instead of a full JDK is possible but not a good idea, since you will be missing the Java debugging tools. The IDE will complain loudly about this condition. Likewise, it is not a good idea to attempt to use the IDE component from an older version of BBj with an installation of a newer version of BBj, or vice-versa. The BASIS-designed plug-in modules that make the IDE Business BASIC-capable are closely tied to the specific version of BBj they are released with. Mixing and matching different versions of the IDE and BBj will result in subtle bugs or even total failure of various features.

We are sometimes asked "can I install the BASIS IDE on a central server and run in a multi-user environment, eliminating the need to install it on individual developer's machines?" The answer is a definite *maybe*. Nothing in NetBeans prevents it outright, but the BASIS installation and plug-in modules configuration were not designed with this in mind. For better or worse, modern IDEs are large, complex *desktop* applications dedicated to individual users who each have their own relatively powerful machine. We have never attempted to use the IDE with a multi-user configuration, or received feedback from anyone who has, but our collected wisdom on the subject appears in the Knowledge Base Article #01149, called Multi-user installation of the BASIS IDE.

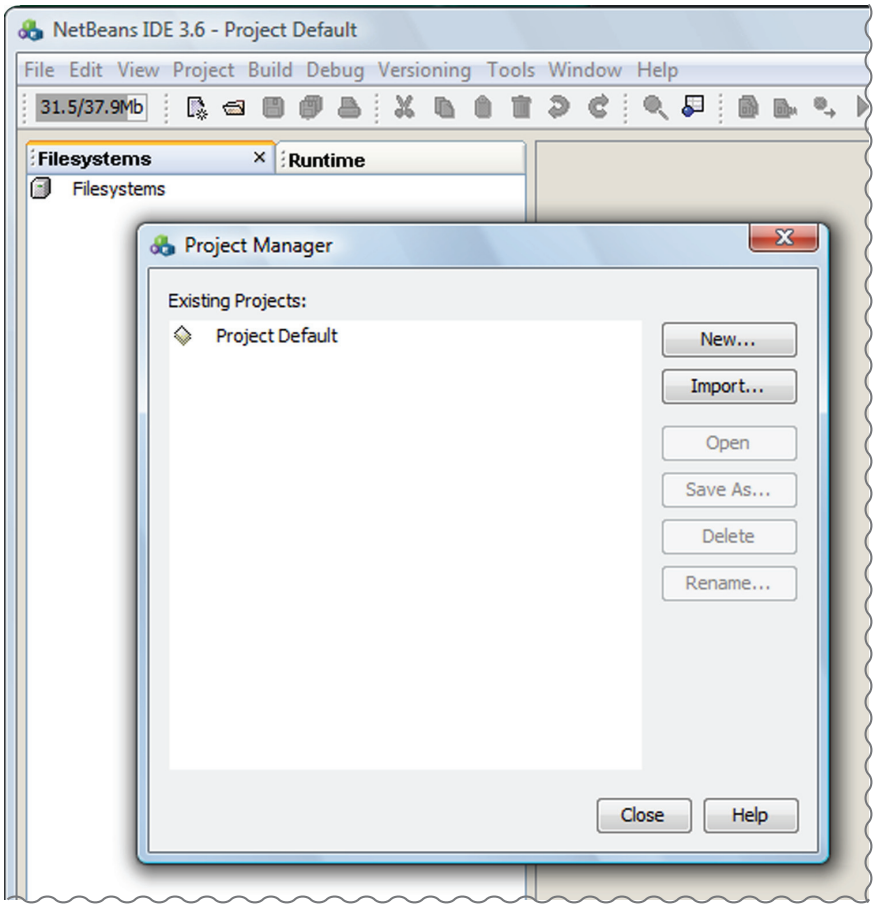


Figure 1. The "Create New Project" dialog

Another question we've been asked is "Can I download the NetBeans IDE from *netbeans.org* on the Web and then simply add BBj to it to get a working BASIS IDE?" Emphatically, this answer is no. The BASIS IDE uses an earlier version of NetBeans, which we've heavily customized with our plug-in modules, and which we maintain independently from the versions available from the NetBeans Web site. However, if you are already a NetBeans user with the latest version of the NetBeans IDE installed on your system, you can certainly install and use the BASIS IDE as well.

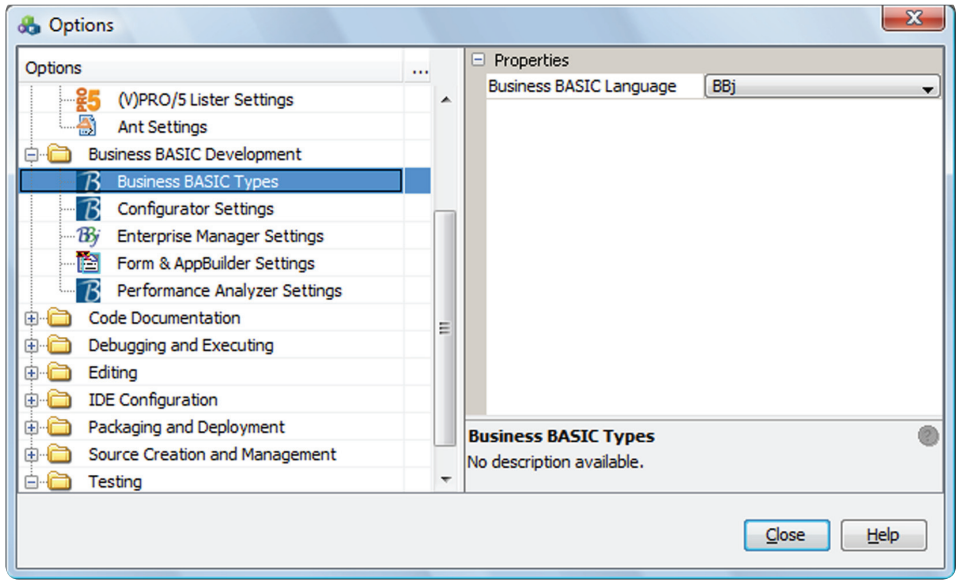


Figure 2. The Options window with Business BASIC Development expanded and Business BASIC Type selected

After successfully installing BBj and the IDE, what's next? The first major step is to define a Project that is a grouping of all the files and resources that belong to the application you are developing. Select **Project > Project Manager** from the main menu and press the [New…] button to create the new Project as shown in **Figure 1.** Every Project needs a unique name to distinguish it from all the others. (When you first start the IDE, you are placed in the *Default Project* automatically, but it is probably not the best idea to remain there.)

After entering a name, the IDE reconfigures to show a new empty project.

You need to specify what kind of BBx® project your application represents. Open the Options window via the Tools menu, then expand Business BASIC Development (see **Figure 2**).

The Business BASIC Language can be either BBj or (V)PRO/5 (Visual PRO/5® and PRO/5®). This setting controls which compiler, executor, and lister will be available to your project, and which set of icons will represent your files in the Explorer >>

Filesystems tab. Because the IDE cannot distinguish which program source files apply to (V)PRO/5 and which are meant for BBj based upon the files names or extensions alone, we recommend segregating BBj and (V)PRO/5 development into separate projects. If you mix the two different types of source in a single project, you must remember to change the setting of the Business BASIC Language property to the correct language type before compiling or running your code. The source files in the Explorer will all display the same icon to the left of their names (either all BBj or all PRO/5 icons), which may be confusing.

Next, start putting files into the new, empty project by associating directories with it. This is called "mounting filesystems." Right-click on the Filesystems node in the Explorer and select Mount as shown in **Figure 3**.

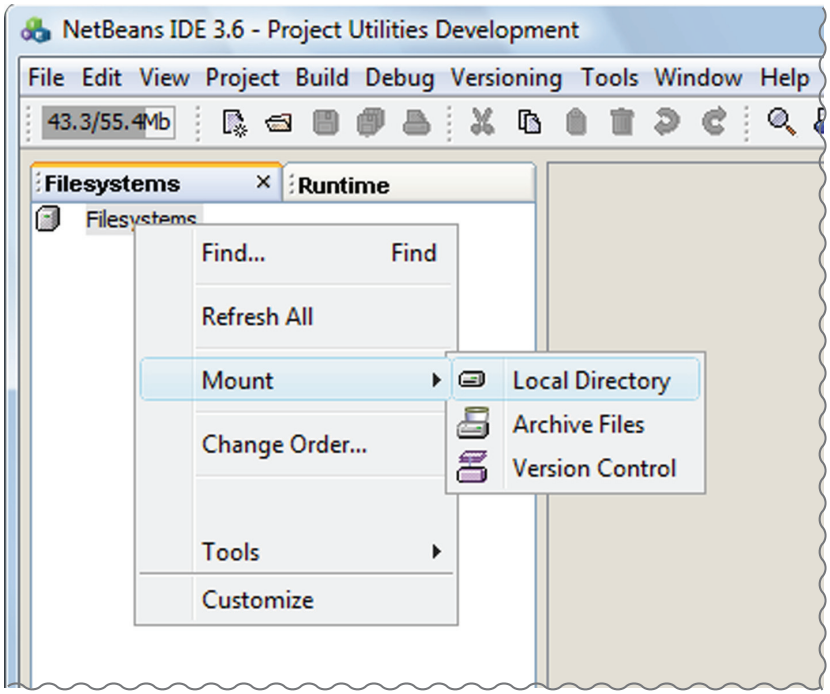


Figure 3. The Filesystems Mount submenu

The choices are *Local Directory*, *Archive Files*, and *Version Control*. A local directory is any directory containing your application's files that resides on a hard disk available to your computer. Archive files are Java .jar archives that your application may need. Version control refers to any local directories containing application files that are synchronized with a source code management (SCM) or version control system (VCS). The IDE can serve as an easy-to-use graphical interface for the CVS or Subversion SCM systems, sparing you the need to use a command line for the most common operations.

For the sake of performance, mount directories (see **Figure 4**) that contain only the source files and resources necessary for your application, and not directories full of other unrelated material (such as the root directory of the entire hard disk partition). The IDE examines each file in a mounted directory in order to assign a file type. If you mount drive C: or '/' as a single filesystem in the Explorer, this process will require a *lot of memory* and a *very long time* to finish.

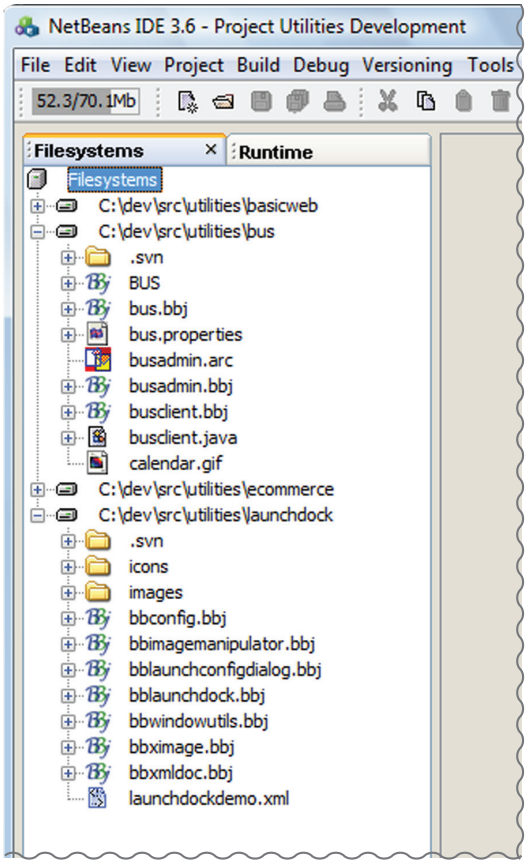


Figure 4. Mounted filesystems in the Explorer

If you are loading your new Project with an existing set of application files, there may be some additional configuration to do before the IDE will recognize your files as Business BASIC source files. The IDE distinguishes file types (and therefore what operations can be performed on those files) based on their file extensions. It is easy to tell if the IDE doesn't know what to do with your files: 1) You will see the 'empty page' icon just to the left of the file name nodes in the Explorer, and 2) when you right-click on a file name to open the popup menu, the first choice will be 'Treat as Text'.

Don't panic, this simply means the IDE doesn't have your file extension in its default list. You can fix this by going back to the Options window as shown in **Figure 5**. Go to I**DE Configuration > System > Object Types > BBj Files**, then select the Extensions and MIME Types property as shown in **Figure 5**. Click on the ellipsis '…' box to open a property editor listing all the file extensions (including no extension at all) that are recognized as Business BASIC file extensions, and then add the extensions used by your files. >>

After adding the file extensions and closing the dialog and Options window, the Explorer will repaint itself. Your files will be assigned BBj or PRO/5 icons, depending on the Business BASIC type you have selected, and the popup menu will show a new set of choices.

You can create as many unique projects as you have applications to develop. There is no artificial limit. You may find that you need the same directory of files to be a part of more than one project at the same time, which is perfectly fine. We mentioned earlier that a best practice is to avoid mixing (V)PRO/5 and BBj development in the same project.

What if your BBj and PRO/5 code is stored side by side in the same directory or directory tree, and must remain that way? Just create two separate projects, set one of them as the (V)PRO/5 and the other as the BBj language type, then mount the same directories in both of them.

If you are just beginning an application development project, rather than loading an existing project into the IDE, you may just be mounting empty directories. In that case, how do you go about creating new files in the IDE?

Begin by selecting the mounted directory in the Explorer where your new file should be placed. Next select **File > New…** from the main menu, or right-click on the directory and select **New…** from the popup menu to open a wizard that guides you through the creation of a new file. The wizard offers templates (see **Figure 6**) for different file types with which the IDE can work.

After selecting one of the templates, you'll be asked for a name. Closing the wizard by pressing the [Finish] button creates the new empty file and automatically opens it in the particular IDE tool with which it is associated.

Would you like every program source file that your company generates to have a copyright notice at the top? Or would it be handy to have some standard boilerplate code in every new file you open, or to have empty file templates that use other file extensions? You can add your own templates to the list shown in the New Wizard.

First, prepare a file containing whatever baseline text you would like included >>

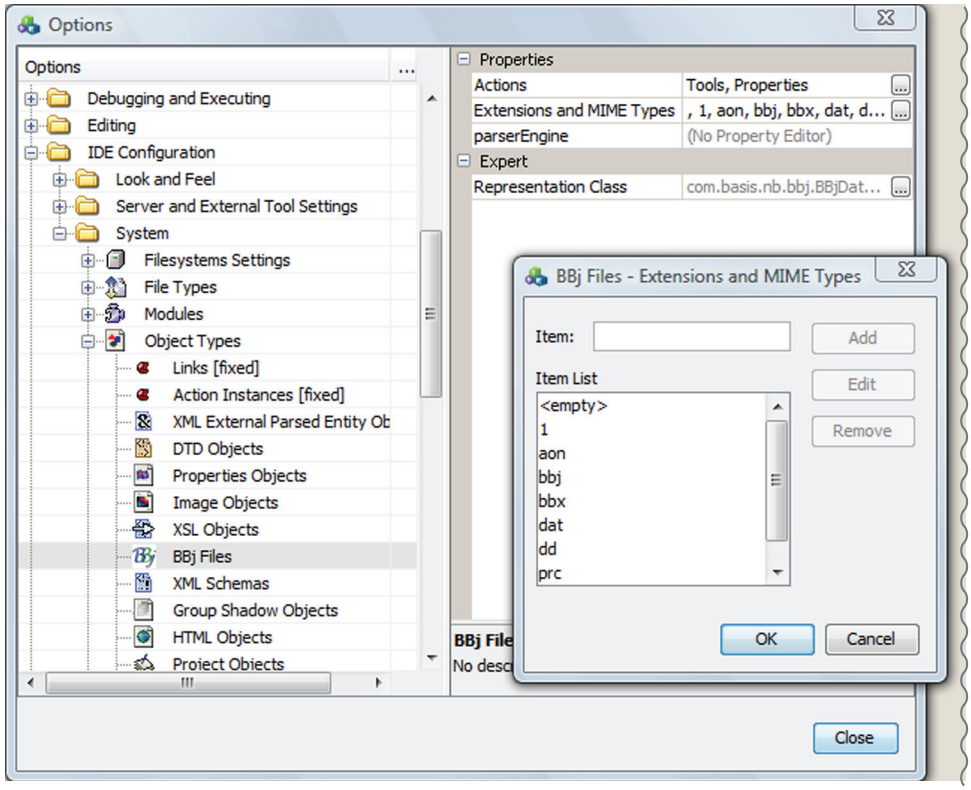


Figure 5. The Options window with the BBj Files – Extensions and Mime Types dialog

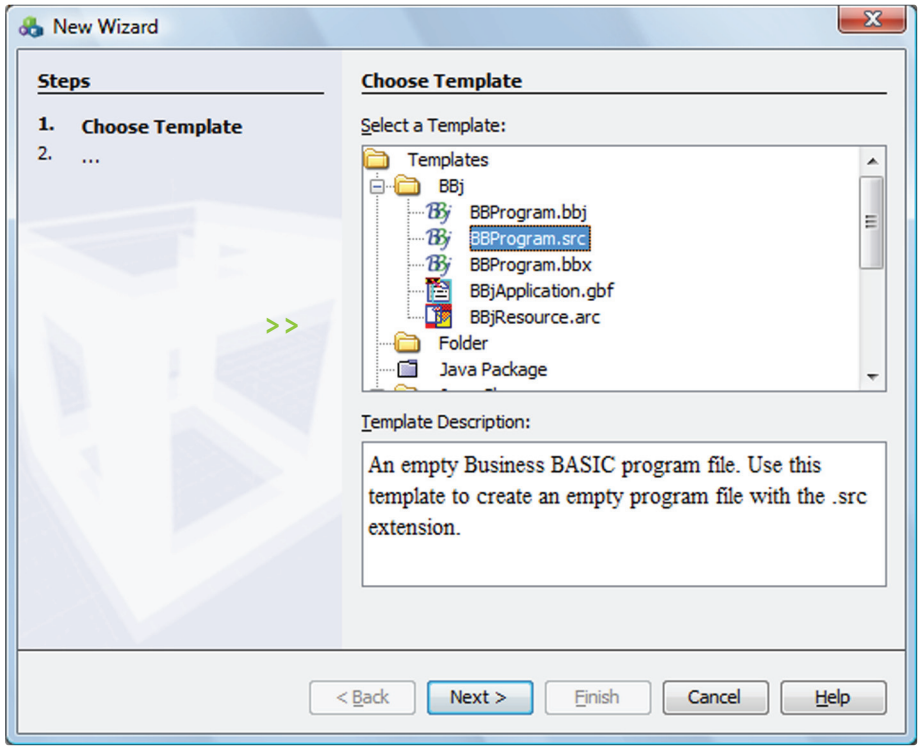


Figure 6. The New Wizard showing the BBj file templates

in your template, then give it a unique name and a file extension that all new examples made from the template should have. (Files meant to be used in FormBuilder or AppBuilder must have the .arc or .gbf extension, but program source files meant to be opened in the Source Editor can have any file extension that you have registered in the Options window Object Types, as we've already mentioned.)

When your file is ready, right-click on it in the Explorer and select **Save As Template…** This opens a dialog that lets you choose the folder (or in other words, the file type category) where your template will appear. Click [OK] to create a template based on your file. You can make changes to this template by going to the Options window and expanding Source Creation and **Management > Template**s. It is even possible to insert macro objects into a template, which can do interesting things like automatically inserting the current date and time or the name of the user who created the new file from the template.

Part II - Editing Source Code

You'll probably spend most of your time in the IDE editing your application's source files in the Source Editor. Think of the Source Editor as a word processor that is purposely designed to assist in writing computer code. It is outfitted with special features to help you work faster and avoid mistakes.

Syntax highlighting, where different types of BBx language entities are assigned different colors, fonts, or styles, helps you discern the structure of your code. Verbs, functions, keywords, strings, comments, and other entity types are each assigned their own color, font, and style to distinguish them from one another.

All the attributes in **Figure 7** are customizable to suit your personal preference. Open the Options window and expand **Editing > Editor Settings > BB Source Editor/Debugger**, then click on the property editor (ellipsis box) of the "Fonts and Colors" property to display the Fonts and Colors dialog.

If you are writing BBj object-oriented programs that take advantage of the BBj API, code completion might become your best friend and constant companion. As the name implies, code completion can automatically complete or fill-in the statement you are currently writing in the editor. When you type certain "trigger" characters such as the period ('.'), the IDE reviews all the text in your file and all the files it refers to, in a process we call *parsing*. This allows the editor to make an educated guess about what you will probably need to type next.

Figure 8 shows a code completion popup window listing the possible choices that would be valid for an instance of the custom object class BBImageManipulator. Selecting one of these choices with the mouse, or by scrolling through the list with the up/down arrow keys and pressing ENTER, causes that choice to be inserted in your text at the cursor position. Code completion spares you the trouble of constantly referring to outside documentation to find out what > >

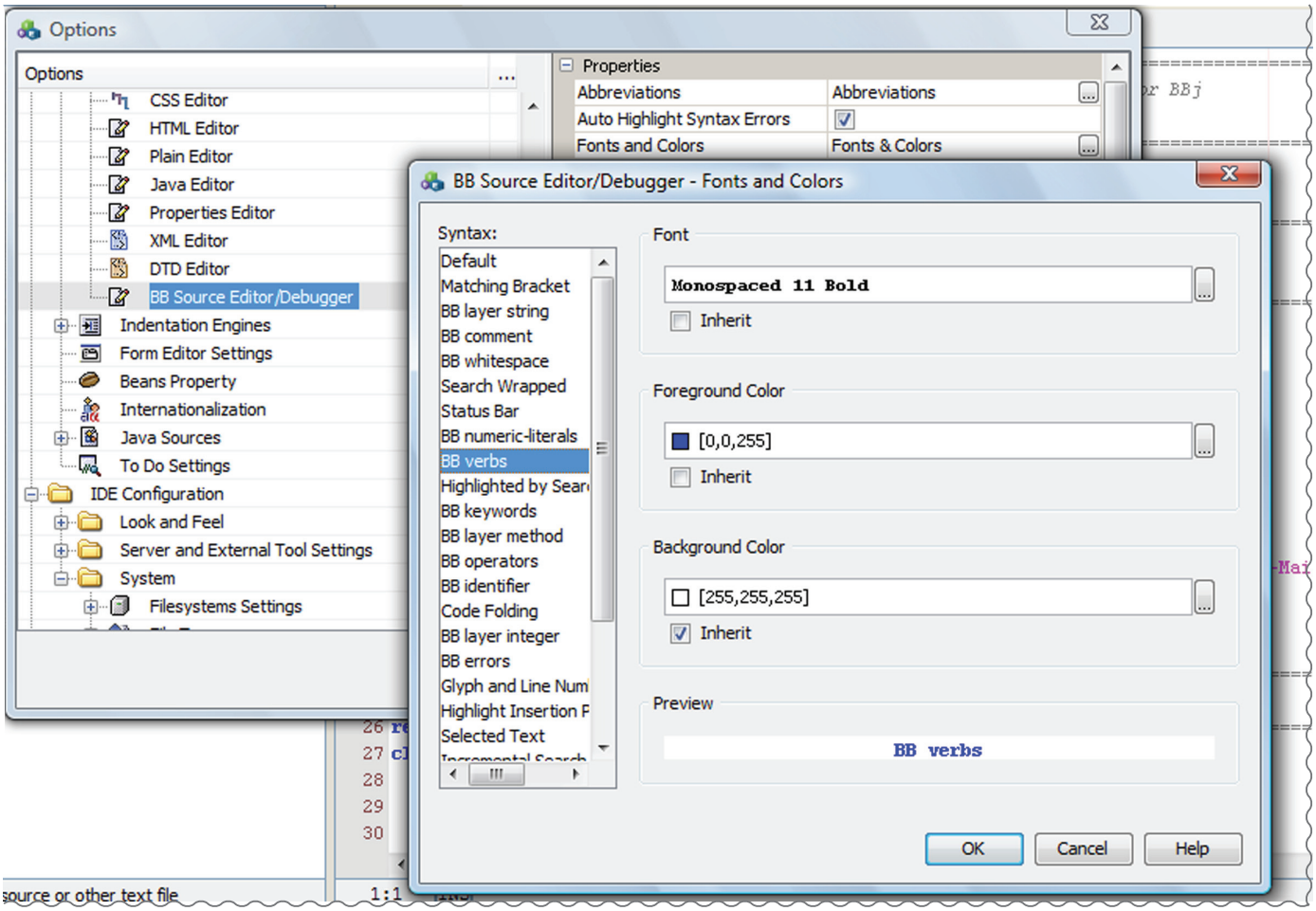


Figure 7. The Options window open to Editor Settings with the BB Source Editor/Debugger – Fonts and Colors dialog

methods or variables are available to a given class object, and also ensures correct syntax by preventing spelling or capitalization errors.

The Options window at **Editing > Editor Settings > BB Source Editor/Debugger** offers two properties that affect code completion (see **Figure 9**). You can turn code completion off by unchecking the Auto Popup Completion Window property. If you would like a longer delay before the code completion window pops up, increase the time shown in the Delay of Completion Window Auto Popup property.

As your application grows and the number of program source files proliferates, finding your way around in them becomes more difficult. It is impossible to remember the exact location of every specific subroutine or algorithm. There needs to be a way to quickly identify and move to a section of code in any given file without spending a lot of time scrolling up and down in the editor, or searching for patterns that may or may not produce the correct results.

The solution is the navigation capability built into the IDE's Explorer Filesystems tab, where you see a hierarchical view of your directories and files. As you would naturally expect, clicking on the little icon just to the left of a directory name causes it to expand and display a list of the files and subdirectories it contains. Did you know that program source files can also be expanded? > >

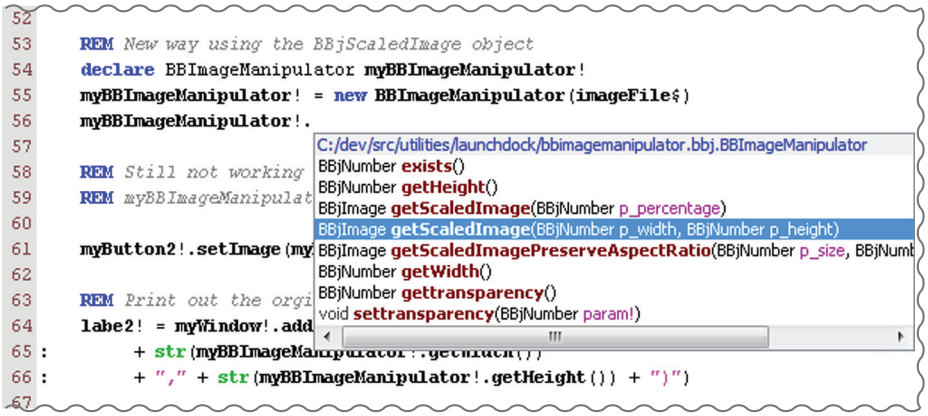


Figure 8. The code completion popup window

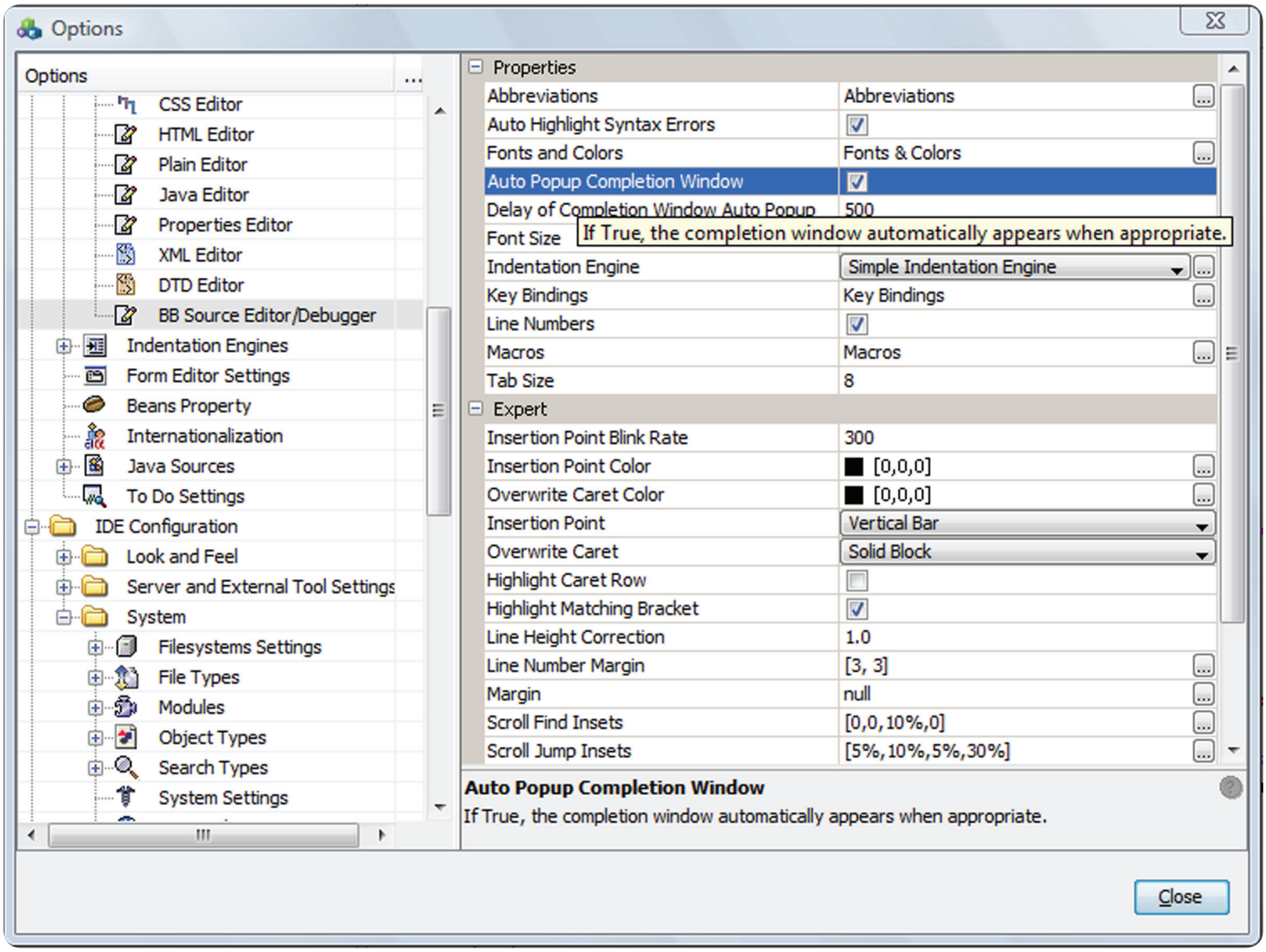


Figure 9. The Options window displaying BB Source Editor/Debugger properties affecting code completion

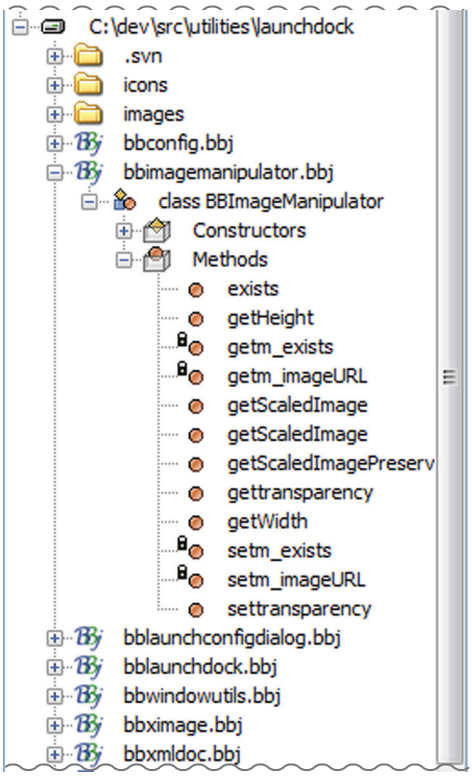


Figure 10. The Explorer showing an expanded BBj source file

Figure 10 displays a file called *bbimagemanipulator.bbj* and all the classes, methods and subroutine labels it contains (which are called "child nodes" of the file they belong to).

This file isn't currently opened in the Source Editor, but double-clicking on one of these child nodes will have two effects: 1) the file will open in the Source Editor, and 2) the cursor will be placed at the start of the class or method or label that was selected. Expanding files in the Explorer and double-clicking their child nodes instantly moves the Source Editor to the corresponding section of code, without your having to do any time-wasting searching.

The parsing process, which updates code completion information and the child nodes used for navigation, also enables the editor to display syntax errors. Each time the file is parsed, all the syntax errors in the file you are currently editing will be highlighted. A simple way to force reparsing of your file at any time is to press [Ctrl]+Space. If you find this syntax error highlighting distracting and would prefer not to be constantly reminded of your mistakes, you can turn it off by unchecking the Auto Highlight Syntax Errors property in the Options window's **Editing > Editor Settings > BB Source Editor/Debugger** page.

An even better way to catch syntax errors before attempting to run your code is do a test compile without writing any output. **Figure 11** shows the **Building > BBj Compiler Settings** page of the Options window. When the 'Compile Without Output' property is checked and the Error Log File property is left blank, no tokenized output file will be created and no error log file will be written to disk when you compile your source code.

When you compile your code, an output window opens at the bottom of the IDE that contains the list of errors that otherwise would have gone to the error log. Each error message is hyperlinked to the source file containing the error as shown in **Figure 12.** Double-clicking on a syntax error message opens the file in the Source Editor and positions the cursor on the line with the error.

Now that is a lot of what the IDE is capable of and how it can help you manage your projects and facilitate application development, but the IDE is extremely full-featured and can do ever so much more! Read on to explore more new features that improve the development experience with time saving techniques and powerful new capabilities. >>

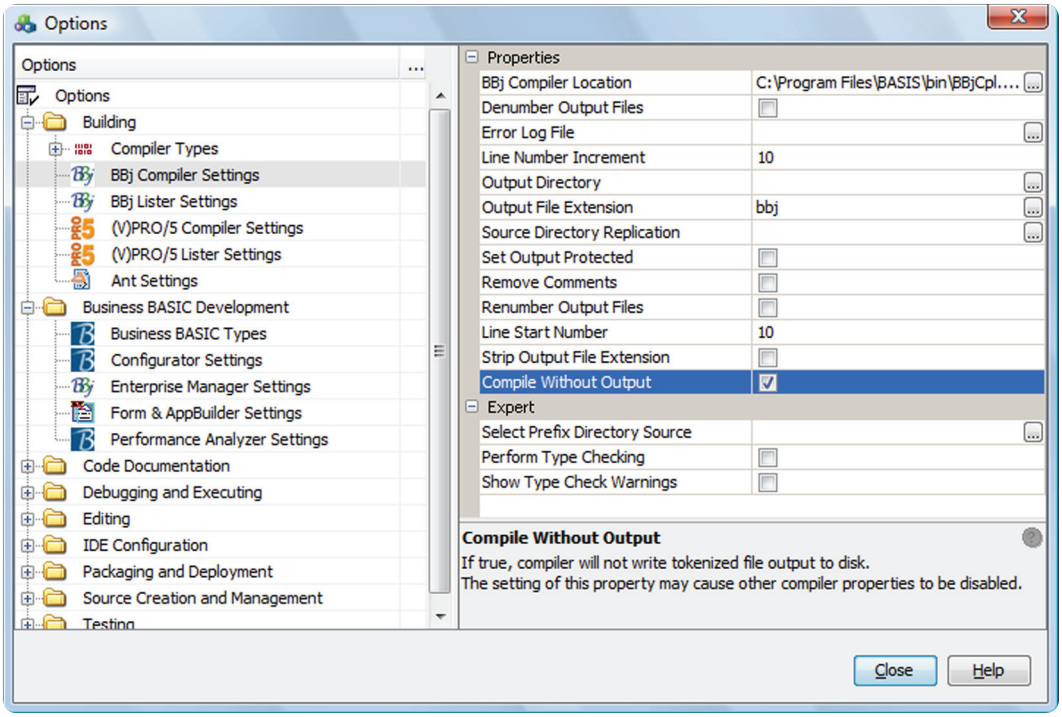


Figure 11. The Options window displaying the BBj Compiler Settings

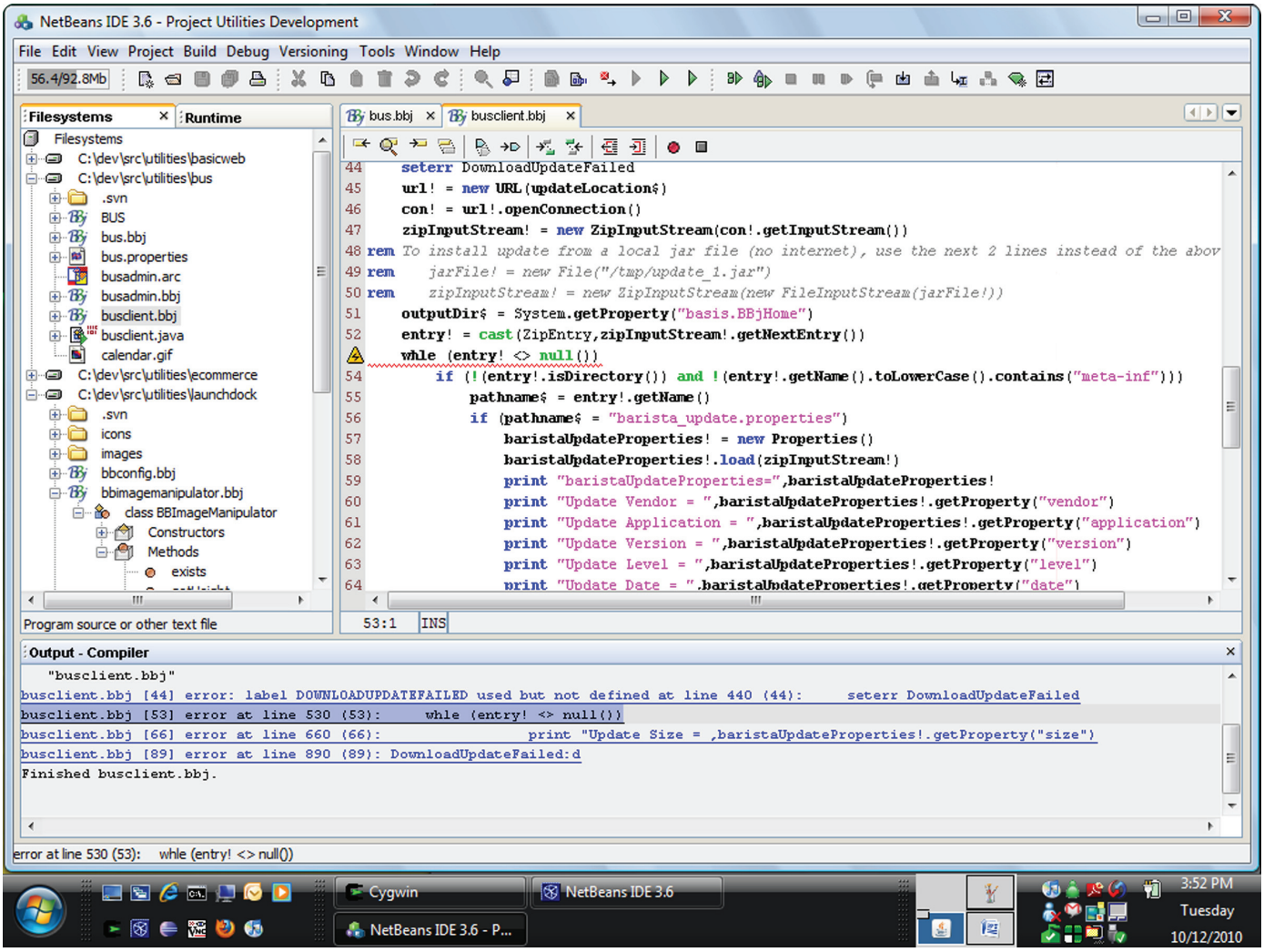


Figure 12. Syntax error hyperlinking

Part III - Compiling and Deploying

Compiling or *tokenizing* is the process of turning ASCII text source code into a binary format that can be executed by the Business BASIC interpreter. It is one of the final steps before turning your application over to your customers and users, and something you will want to do often during development for testing purposes.

As a PRO/5 or BBj developer, you are already familiar with the pro5cpl and BBjCpl compiler utilities and their various command line options. The BASIS IDE provides a convenient graphical front end for these utilities, which simplifies the process of selecting and compiling large numbers of files. The IDE itself doesn't actually contain any built-in compilers; it merely connects to the cpl compilers found in your installed version of BBj or BBx. At compile time, the IDE assembles all the selected source files, invokes the specified cpl compiler as a process in a new execution thread, feeds it the appropriate command line parameters garnered from the properties you have specified in the Options dialog, and then displays the results when the process is finished. Using the IDE for compiling spares you a lot of careful typing and shows you what's happening in a more comprehensible way. As we already mentioned in *Part 2 Editing Source Code*, any errors found during compilation are displayed in the Compiler Output Window as hyperlinks. Double-clicking on a hyperlink opens the original source file in the Source Editor, where the line containing the error is highlighted. >>

Open the Options window and expand **Building > BBj Compiler Settings** to display a list of the properties for compiling with BBjCpl, as shown in **Figure 13**.

Most of the properties you see here directly correspond to parameters you would enter on the command line if you were using the compiler in a shell outside of the IDE. Holding the mouse cursor over the name of the property causes a "hint" window to appear with a short explanation of what the property controls. Most of these properties are obvious, so we'll spend some time looking at the most critical or unusual ones.

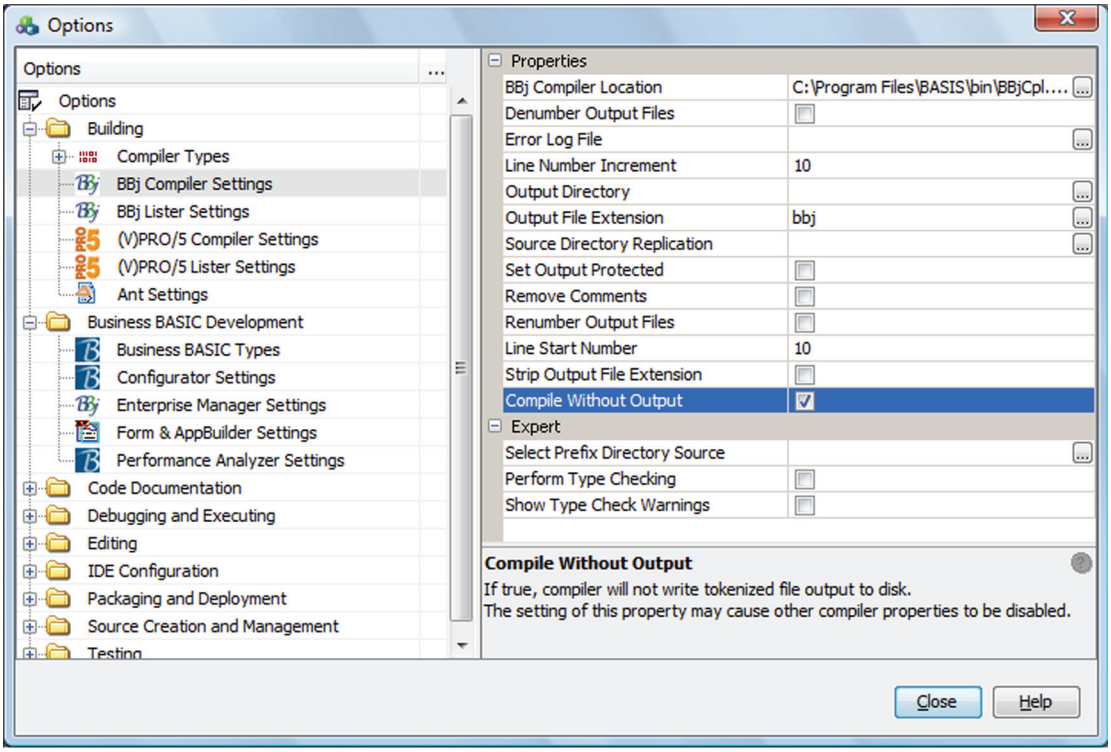


Figure 13. The Options window with the BBj compiler options.

BBj Compiler Location – This property comes pre-configured with BBj. It simply points to the installed BBjCpl compiler. If you move the utilities around or store them in unusual places, you will need to update this property.

Error Log File – This property corresponds to the **-e** command line parameter. It is worth mentioning only because if you specify an error log file, the compiler errors get written to a disk file instead of displaying in the IDE's Compiler Output Window. You must leave this property blank in order to display compiler errors as hyperlinks connected to the original source.

Output Directory – This of course represents the **-d** command line parameter and specifies the path to the directory where the compiler places the tokenized files. For best results, you should have previously created and mounted the output directory in the NetBeans Explorer. If this directory does not yet exist, the BBjCpl compiler will create and populate it with tokenized files, but you won't see the results in the Explorer Filesystems tab and may wonder if anything really happened. To create tokenized files in the same directory as the source files, leave this property blank.

Output File Extension – This is the **-x** command line parameter, and specifies the extension added to the end of tokenized files. The extension is not limited to a certain number of characters. If the extension is registered as a BBj module extension, the compiled files are assigned a special icon in the Explorer that distinguishes them from all other file types. Leaving this option blank triggers the file naming behavior discussed in the 'Strip Output File Extension' property below.

Source Directory Replication – This property has no command line equivalent; it represents a capability added by the IDE that you can't easily get with plain vanilla BBjCpl at the command prompt. This property represents the name of a specific directory from the directory path of the source file(s), which becomes the point at which source directories are replicated in the output directory structure. It modifies the directory path string that is forwarded to the compiler with the **-d** parameter so that the generated output files will exist in a directory tree hierarchy which is similar to the directory tree of the source files. Did you get all that? If not, don't worry. We're going to discuss this in more detail a little later.

This property is not essential for compiling; it can be safely ignored if you have no special concerns about the default handling of the output directory. If no output directory is specified in the 'Output Directory' property, this property is ignored because your compiled files will be created right alongside your source files in the same directory structure.

Strip Output File Extension – This property also has no corresponding BBjCpl command line parameter. When set to 'true' (when the box is checked), it removes the last file extension from the name of the compiled output file, provided that an output directory is specified and the 'Output File Extension' property is left blank. If no output directory is entered in the 'Output Directory' property, the 'Strip Output File Extension' property is ignored and has no effect on the naming of the file. To make this perfectly clear, here's a summary of the rules applying to tokenized file names and extensions: >>

Case 1: Output file extension supplied.
The tokenized file is named with the specified extension. The 'Output Directory' and 'Strip Output File Extension' properties do not affect file naming.

Case 2: No output file extension supplied. No output directory supplied.
The tokenized file is located in the same directory as the source file and has the same name as the source file, minus the last extension of the source file.

Case 3: No output file extension supplied. Output directory specified. Strip Output File Extension set to 'false' (unchecked).
The tokenized file is placed in the output directory, and has the same name and file extension as the source file.

Case 4: No output file extension supplied. Output directory specified. 'Strip Output File Extension' set to 'true' (checked).
The tokenized file is placed in the output directory, and has the same name as the source file, minus the last extension of the source file.

Compile Without Output – We touched on this property earlier in our discussion of editing source files, but this time the setting needs to be different. You are compiling source files with the intent of getting tokenized files written to the hard disk, so this time the property needs to be 'false' (unchecked).

Select Prefix Directory Source – This property represents the **-P** or **-c** command line parameters, depending on which of the four possible directory prefix list sources is selected. A prefix directory list is necessary when the compiler is directed to perform type checking of BBj object syntax, and therefore this property is ignored unless the 'Perform Type Checking' property is set to 'true'. The four possible settings are shown in the BBj Compiler Settings – Select Prefix Directory Source dialog in **Figure 14**.

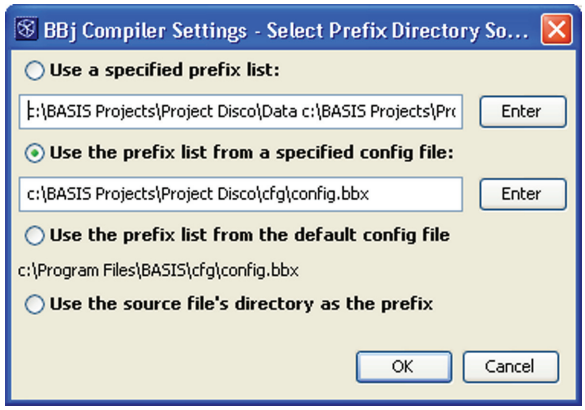


Figure 14. The Prefix Directory Source dialog

Use a specified prefix list. Select the radio button, type a space-separated list of directories in the text field and press the Enter button. The command line arguments forwarded to the compiler will include the **-P** parameter and the list of prefix directories from this text field.

Use the prefix list from a specified config file. Select the radio button, type the path/file name of the config file that contains the desired prefix directories and press the Enter button. The command line arguments forwarded to the compiler will include the **-c** parameter and the path/file name of the config file from this text field.

Use the prefix list from the default config file. Select the radio button. The command line arguments forwarded to the compiler will include the **-c** parameter and the path/file name of the config file specified in the BBj Execution and Debug Settings > Config File Location property.

Use the source file's directory as the prefix. Select the radio button. No **-P** or **-c** parameters are added to the command line forwarded to the compiler. If type checking is required (the 'Perform Type Checking' property is set to 'true'), the compiler will use the source file's directory as the directory prefix.

Perform Type Checking – This represents the **-t** command line parameter. When set to 'true', the compiler will perform type checking on BBj object syntax used in the source file.

Show Type Check Warnings – This is the **-W** command line parameter, which is ignored unless the 'Perform Type Checking' property is set to 'true'. When set to 'true', the compiler will display type check warnings as well as errors in the output results.

We ought to mention that certain BBjCpl command line parameters are not available when compiling BBj source code with the IDE. These include the ? parameter, which displays a usage summary, the @ parameter for specifying a text file containing a list of files to be compiled, and the **-R** parameter for recursive compiling of files in subdirectories. (As we will soon find out, in the IDE recursive compiling is performed by selecting a directory instead of a file in the NetBeans Explorer and choosing **Compile All** or **Build All** rather than **Compile** or **Build**.)

Now that we've covered the most important compiler options and you are all configured, let's get on with the actual compiling (the hard part is over). To compile a file in the IDE:

1. Select a program source file in the Explorer Filesystems tab.
2. Right click to open the popup menu (or open Build on the main menu).
3. Select **Compile** or **Build** (or use their short-cut keys) to kick off the process.

A quick word about the difference between '**Compile**' and '**Build**': When you select **Compile**, the IDE checks to see if a compiled version of the file already exists, and if the date/time of its creation or last modification is more recent than the last modification date/time of the source file. This is called the "up-to-date" check. If a compiled version already exists and is up-to-date, the source file is not recompiled. The **Build** choice does not check for pre-existing and up-to-date compiled versions of the source. All selected source files are compiled again, regardless of the status of any already compiled versions. Choosing Compile instead of Build may therefore save you some time, since only the changed files in your selection are recompiled rather than all of them. >>

You can compile more than one file at a time by holding down the [Ctrl] key and selecting multiple files, or you can select a directory in the Explorer instead of a file. Here's where we talk about recursive compiling, which is merely the difference between **Compile/Build** and **Compile All/Build All**. When you select a file or group of files in the Explorer, you are offered the choice to **Compile** or **Build**. If you select a directory rather than a file, the choices expand to include **Compile All** and **Build All**. The **Compile** and **Build** options work only with the files in the selected directory. Compile All and Build All process not only the files in the selected directory, but also the files in any subdirectories of the selected directory.

The **Clean** and **Clean All** menu choices (as you might expect) are there to wipe out the tokenized files if you no longer want them. **Clean** will delete only the files in the currently selected directory, while **Clean All** does a recursive delete through all the subdirectories of the selected directory. Use with care!

As promised, we need to step you through a more detailed description of the 'Source Directory Replication' compiler property and discuss its implications. Source directory replication is only going to be interesting if you keep your source text files in a different place than your tokenized files. If your preferred development style is to lump all the hundreds or thousands of files in the entire application into the same directory, you don't have any source directory structure to replicate and you should leave this property blank or set it to 'Use no source directory replication at all'. (However, if your preferred development style is to lump all the hundreds or thousands of files in the entire application into the same directory, we'd love to convince you there's a Better Way To Do Things.)

The 'Output Directory' property of the compiler options lets you set the destination of the generated tokenized output files. If you leave this property blank, the compiled files are created in the same directory as their corresponding source. If the 'Output Directory' property is filled in with a directory path, the IDE automatically derives a new output path based on the 'Output Directory' property and the location of the source files selected for compiling. This new output path is used as the **-d** parameter given to the cpl compiler, as shown in **Example A** below.

Example A
Setting of the Output Directory property:
```
C:\BBj Development\level A\level B\bin
```
Directory containing source files:
```
C:\BBj Development\level A\level B\src\level C\level D
```
Adjusted –d parameter sent to the compiler:
```
C:\BBj Development\level A\level B\bin\level C\level D
```

In this example, the source files are kept under **src** and the tokenized binary files under **bin**. The IDE computes the directory path for the **-d** command line parameter by comparing each segment of the source path with each segment of the output directory, starting from the beginning of the path. Each segment of the source path is identical with each segment of the output path until the **src** segment, so all segments after src in the source path are appended to the output path, which is then used for the **-d** parameter. This automatic path derivation algorithm insures the output from files in src will go in **bin**, the output from src\level C will go in **bin\level C**, the output from **src\level C\level D** will go in **bin\level C\level D**, etc. The automatic path derivation is in effect when the 'Source Directory Replication' property is set to 'Use default source directory replication'.

In **Example B**, this automatic derivation algorithm fails to make the expected output path:

Example B
Setting of the Output Directory property:
```
D:\Our Product\Release 4
```
Directory containing source files:
```
C:\Development\Our Product\beta\gui\main menu
```
Incorrectly adjusted –d parameter sent to the compiler:
```
D:\Our Product\Release 4\Development\Our Product\beta\gui\main menu
```
>>

The first non-matching segment is **C:**, so everything after the first segment is appended to the output path specified in the Output Directory property. This results in a long, awkward output path that probably isn't what was desired. The 'Source Directory Replication' property's 'Use a selected directory for replication' setting is designed to correct this problem and help the output path derivation algorithm do the right thing, as shown in **Example C**:

Example C
Setting of the Output Directory property:

```
D:\Our Product\Release 4
```

Directory containing source files:

```
C:\Development\Our Product\beta\gui\main menu
```

Setting of the Source Directory Replication property:

```
gui
```

Correctly adjusted –d parameter sent to the compiler:

```
D:\Our Product\Release 4\gui\main menu
```

Because a segment from the source directory path is identified in the 'Source Directory Replication' property, the automatic derivation algorithm can skip the segment comparison process. Everything in the source path starting with the **\gui** segment is appended to the output directory, which results in a much more appropriate **-d** parameter output path.

And so you reach the end of this rather long IDE discussion, we hope you find it helpful! ■

Download the latest version of BBj and enjoy these new enhancements:

Updated for Java 1.6:

- Enhanced NetBeans 3.6 codebase no longer requires Java 1.5 to run; now uses the same Java 1.6 version as the BBj server.
- Adds new Java 1.6-compatible code completion database for Java code development.

Enhanced code completion:

- Adds file navigation from the Explorer. Expand file nodes to see classes, methods and labels contained in the file; click on one of them to zoom to that spot in the file.
- Includes new BBjConstants; a BBj API class containing often-used constants for constructs like MSGBOX, making them accessible through code completion.

<<< table of contents