

Freedom of Choice: Using Object Code Completion in the IDE

By Nick Decker

Object-oriented syntax brings a great deal of power to BBx® with a certain amount of complexity. Mastering the power requires learning about dozens or even hundreds of artificial constructs called objects. Each object contains information and unique capabilities, called methods, for manipulating or sharing that information. In a well-written object-oriented application, objects represent items in the user interface, or important collections of data, or processes that work with data. To write that kind of code, developers need to know what object classes are available and what each of them can do. Therein lies the complexity. Do you have an extra monitor attached to your computer so to simultaneously display both the code you are editing and the online BBj® object documentation, or do you possess the gift of a photographic memory that helps you recall all the details about the objects with which you are working?

Regardless of the answer, the code completion feature of the BASIS IDE is the simplest and smartest solution. Code completion is a kind of online help system for custom BBj objects automatically built into the editor and functions. Simply put, the code completion feature evaluates the characters typed into the editor. When it senses that you need to enter the name of an object's class or one of its methods, it displays a list of the possible object classes or methods available for use in the current situation. This gives developers complete freedom to select an item from the list, pasting it automatically and instantly into the editor. Now there is no need to refer to the documentation, no need to type carefully, and no chance for careless mistakes.

Code Completion in BBj 5.0

The first phase of code completion was available with the BBj 5.0 release of the IDE. This version supported the BBj objects that represent classes either from the BBj API or from Java, but it did not have a programmatic syntax for assigning an object variable name to one of these types of objects. This limitation required some extra effort to get code completion working in the BBj 5.0 IDE. Without a variable type declaration syntax in the language, the editor in the IDE could not determine the class type a BBj object variable represented. The developer must identify to the IDE, one variable at a time while typing in the editor, or by embedding a list of the variables with their associated types before starting the IDE. Once the editor knows the class type of a BBj object variable, it is finally able to connect to the class definition and get the correct items to display in the code completion popup window.

Figure 1 shows the “undeclared BBj object” popup window that appears anytime the editor cannot resolve the type of BBj object variable name. Selecting a class from the list in this window causes the variable name to link to a class type. The editor remembers this association and writes it to a special file called **BBjObjectTypes.txt** when saving the edited file.

continued...

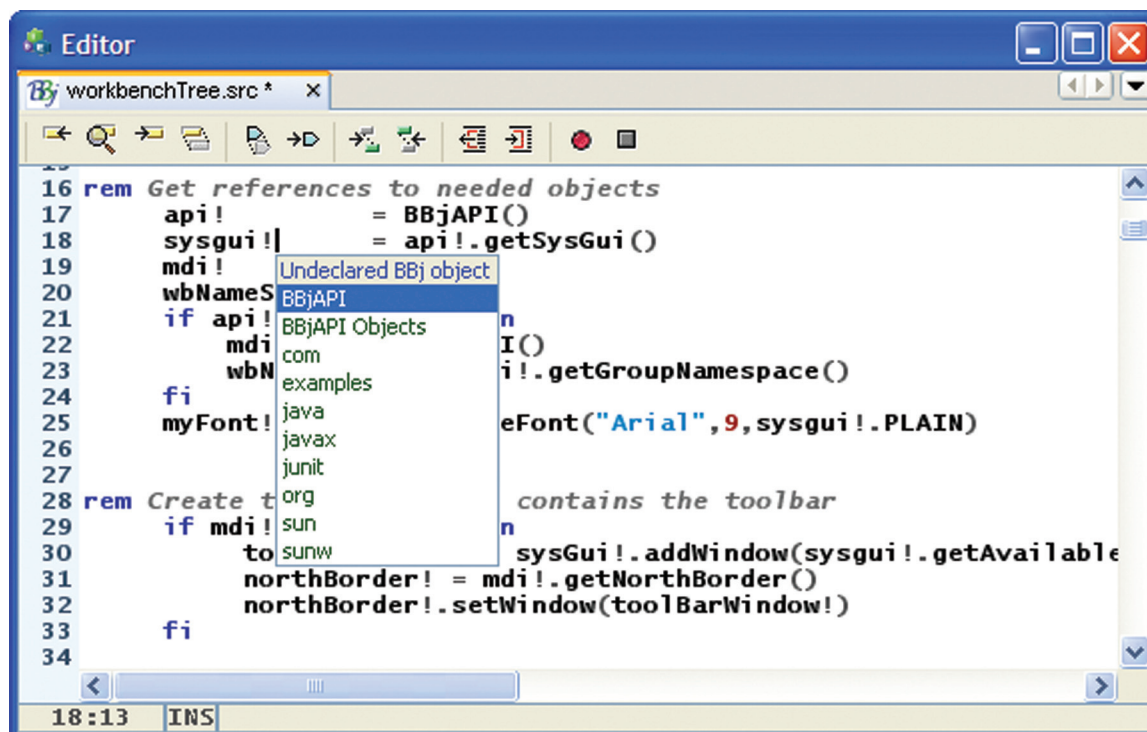


Figure 1. The Source Editor



Nick Decker
Engineering
Supervisor

Each project created in the IDE has its own **BBjObjectTypes.txt** file. This file is located in the **ide/defaultuser/system/Projects/<project name>/system/ParserDB** directory and is read whenever a file is loaded into an editor and written whenever a file is saved from an editor. As shown in **Figure 2**, this simple text file lists the BBj object variable names used in each file belonging to the project. Alternatively, a developer can declare all the intended BBj object variables before even starting an editing session by opening and modifying this file in any text editor.

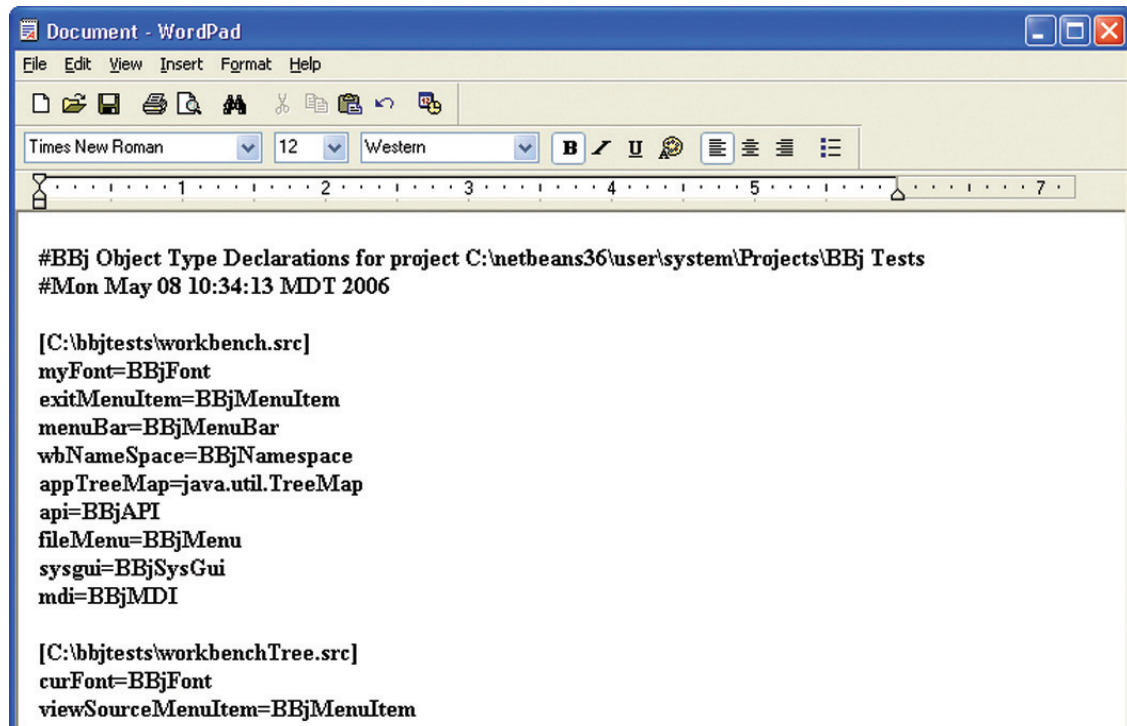


Figure 2. Sample **BBjObjectTypes.txt** file

BBj object variable declarations, once made, are not set in concrete. Selecting the **Edit/Import BBj Object Declarations** item from the Tools menu opens the edit/import dialog shown in **Figure 3**. Use the **Edit Declarations** tab of this dialog to add, edit, or delete variable declarations, then press [Save Variable Declarations] to apply the changes in the editor. The **Import Declarations** tab allows the import of variable declarations from other files listed in the **BBjObjectTypes.txt** file into the current file, which saves time by reusing work already done elsewhere.

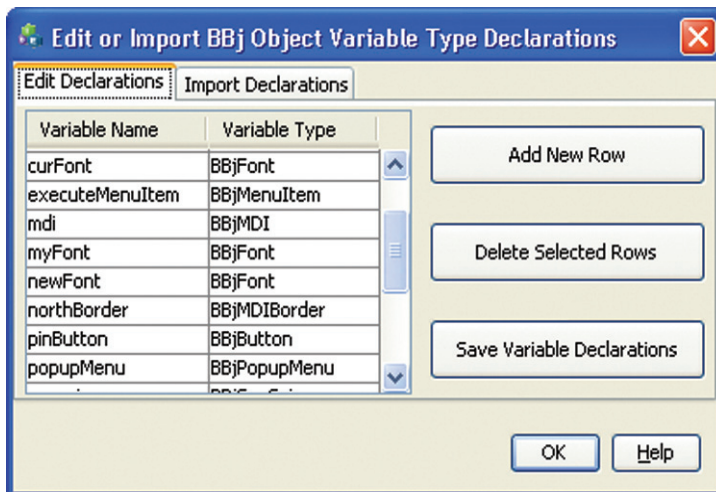


Figure 3. Edit/Import dialog

Code completion works well in the BBj 5.0 IDE after defining the BBj object variable, but all the extra effort does make it less convenient to use. The lack of a declaration syntax means that variable type declarations are not “built in” to the applicable source files; they have to be stored separately in the **BBjObjectTypes.txt** file. This means that other developers on your team who work with the same files also need your **BBjObjectTypes.txt** file or perhaps you need a copy of theirs. Sharing or merging these files becomes yet another time-consuming coordination issue.

continued...

New BBj 6.0 Custom Object Syntax and BBj Language Parser

BBj 6.0 makes a quantum leap in the direction of true object-oriented programming. Developers can now write their own custom object classes in Business BASIC and use them just as BBj API or Java objects used in previous versions. Along with custom objects comes a new syntax for declaring variable types and a new language parsing capability, both of which have a huge impact on code completion in the IDE.

The IDE Source Editor and Debugger both access the BBj language parser to handle custom objects. The parser analyzes the editor's contents and stores information it finds about BBj object variables and custom object classes. In turn, the editors have access to everything the parser learns about the structure of the file a developer is editing, as well as any files with classes mentioned in it. This is the key BBj 6.0 improvement that enables code completion to work with custom object classes.

The most critical new syntax, as far as code completion is concerned, involves the **USE** and **DECLARE** statements. **Figure 4** shows a set of USE statements from the **Workbench.src** file used in a demo at TechCon2006. USE statements tell the system which classes the BBj object variables will use inside the file currently being edited (**Workbench.src** in this case). Without a corresponding USE statement, a custom object class definition cannot be found and read by the BBj language parser, which in turn means the code completion feature has no information about it.

```
rem Use statements
use :.../Workbench/AppMgr.src::AppMgr
use :.../Workbench/WorkbenchMDI.src::WorkbenchMDI
use :.../Workbench/WorkbenchApp.src::BBjWorkbenchAppInfo
use :.../Workbench/WorkbenchApp.src::BBj5031WorkbenchAppInfo
use :.../Workbench/WorkbenchApp.src::ThirdPartyWorkbenchAppInfo
```

Figure 4. Sample USE statements

Figure 5 shows some examples of the DECLARE statement from **Workbench.src**. DECLARE statements assign a particular BBj API, Java, or user-written class as the type of a specific BBj object variable. For example, the definition of the variable **workbenchMDI!** is an instance of the **WorkbenchMDI** custom class (referenced in the statement **use :.../Workbench/WorkbenchMDI.src::WorkbenchMDI**) and the variable **appMgr!** is an instance of **AppMgr**.

```
rem Declarations
declare WorkbenchMDI workbenchMDI!
declare AppMgr appMgr!
```

Figure 5. Sample DECLARE statement

DECLARE is the magic word that eliminates manual variable declaration and the need for storing type declarations in the **BBjObjectTypes.txt** file. The BBj language parser reads the DECLARE statements directly from the source file, which makes external storage of that information unnecessary. Configuring or sharing **BBjObjectTypes.txt** files between developers is no longer necessary. Simply write a DECLARE statement for the desired variable and avoid the nagging "Undeclared BBj object" popup window. All of the manual variable type declaration mechanisms remain in place in the BBj 6.0 IDE, but are not necessary when using the USE and DECLARE syntax. Variables in DECLARE statements are not written to the **BBjObjectTypes.txt** file and are not visible in the Edit/Import dialog. Furthermore, variable type declarations made with DECLARE statements automatically supersede any previously defined type declarations created using the manual system.

Using Code Completion

The most effective way to explore code completion is to approach it from an instructional perspective. Before getting started, note that BBj relies on the directory prefix list in the **config.bbx** file. If your custom object classes are located in directories not included in the prefix list, the BBj language parser will not find them and code completion will not work, regardless of the USE statements that reference them.

A couple of editor options affect code completion behavior. Open the Options window and expand **Editing | Editor Settings | BBj Source Editor/Debugger** to expose the Auto Popup Completion Window and Delay of Completion Window Auto Popup properties (see **Figure 6**). The Auto Popup Completion Window property is the code completion "on/off" switch. Uncheck this property to turn off code completion if that is your preference. The Delay of Completion Window Auto Popup property controls the amount of time, in milliseconds, that must pass before the completion popup window shows itself. Lengthen or shorten this delay time to suit your own preference and typing speed.

continued...

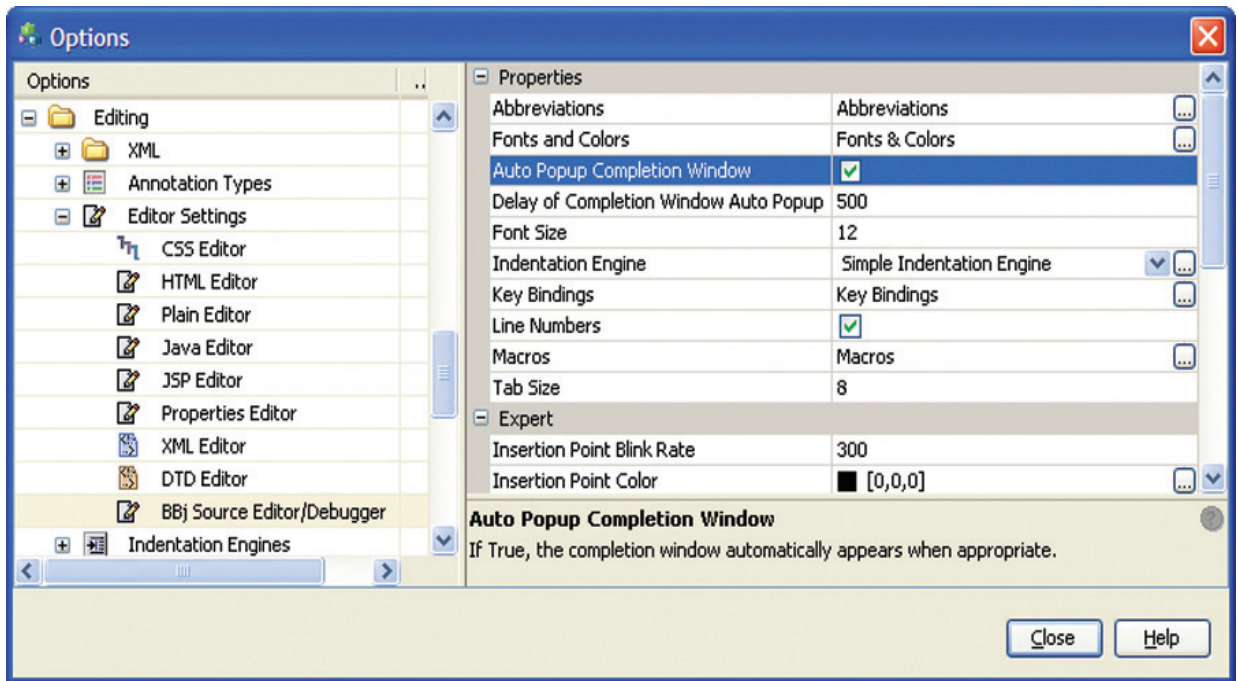


Figure 6. Options window showing the BBj Source Editor/Debugger options

The IDE begins initialization of the code completion system the instant a file opens in the Source Editor or Debugger. Clicking the mouse in the editor window or typing a character does several things:

- Sends the content of the opened file to the BBj language parser
- Causes the parser to
 - analyze the file to learn about its DECLARED variables and custom object classes
 - scan the USE statements found in the file and then analyze each of those files in turn for their variable and class information
 - Invokes the editor to ask the parser for information about all of the variables and classes it found, then stores this information in an online database for later retrieval.

This process occurs quickly when first editing a blank file or a file that is relatively simple. When opening a large and complex file, especially one that has many USE statements, the initial parsing process may require a few seconds to complete (because of a keystroke buffer, text will appear in the editor as soon as the parsing finishes). Once a file referenced in the USE statement is parsed, it is never reparsed unless you edit the referenced file.

To activate the code completion feature, type such “trigger” characters as a ‘.’ immediately after the name of a BBj object variable or with certain keywords like **new**, **declare**, **extends**, or **imports** followed by a space character. When the editor detects one of these characters or character sequences, it asks the parser to analyze the file again, updates the code completion database with any changes made since the last time it parsed the file, and displays the completion popup window with the list of applicable choices. Continue typing to narrow the list of choices, or select an item with the [ENTER] key or a mouse double click to paste it into the editor at the cursor position, shown in the screenshots in Figure 7.

continued...

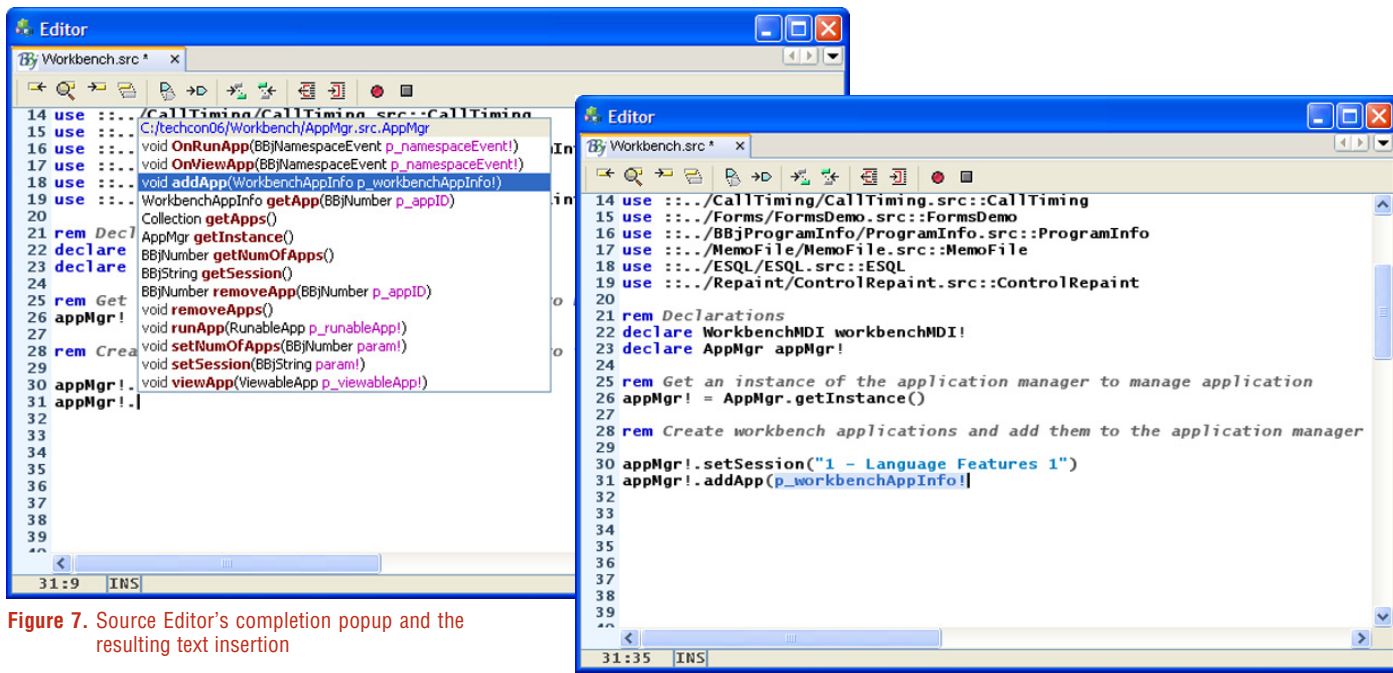



Figure 7. Source Editor's completion popup and the resulting text insertion

Summary and Future Possible Enhancements

Connecting the IDE to the BBj language parser opened a new world of possibilities for extending and improving the code completion feature. In future releases, BASIS is considering these enhancements:

- *File navigation from the Explorer* Like the Source Editor, the file Explorer would learn about the structure of a program file from the BBj language parser. Similar to Java nodes in the Explorer, BBj file nodes could be expandable and display the custom object classes they contain, as well as each method and variable. Double clicking one of these “child” nodes would reposition the editor to the location in the file of the class declaration.
- *Error highlighting in the Source Editor* The parsing process would uncover syntax and type-check errors. The BBj language parser could forward a list of these errors back to the Source Editor to highlight the affected lines. The programmer would then see errors in the editor without needing to compile the code first.
- *Javadoc-style help* In addition to the code completion window that lists the available classes and methods, create a matching text window with short comments explaining what each choice can do and how to use it.

Code completion exists for one reason only: To make developers more productive. Give it a try in the new BBj 6.0 IDE. 



For more information, see [A Primer for Using Custom Objects](#).