VisualAge for Java, Version 2.0

IDE Basics

Contents

Legal Notices

Notices

Programming Interface Information Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

IDE Basics

Overview of the VisualAge for Java IDE

VisualAge for Java is an integrated, visual environment that supports the complete cycle of Java program development. You can create Java applets, which run in web browsers, and standalone Java applications.

With VisualAge for Java, you can do the following tasks:

- build Java programs interactively
- run Java programs
- run fragments of Java code before you include them in classes
- debug Java programs, changing them as you run the code
- manage multiple editions of Java code
- import Java source and binary code from the file system
- export Java source and binary code to the file system
- build, modify, and use beans

Development Without Files

As a programmer, you are probably familiar with developing programs in a file-based environment. You write code in source files and compile and link them into executable binary files. As you build the executable code, you must manage files and file dependencies. This is an error-prone process that takes time away from your primary task of programming.

By providing a development environment without files, VisualAge for Java performs the tasks of managing and compiling code. Browsers let you view and edit classes and methods individually. VisualAge for Java compiles Java source code for you when you save it.

From within the IDE, you can import code from the file system into VisualAge for Java. You can also export code to the file system if you want to work outside VisualAge for Java.

Editing is Integrated
The Workbench and browsers reflect an object model rather than a file-based model. The object hierarchy of program elements (projects, packages, classes, interfaces, and methods) provides a structure for the code. The Workbench and browsers have a source pane with full editing capabilities that let you modify and save source code.

Compilation Occurs when You Save Source Code
After you modify the source for a class or method, you save it, and VisualAge for Java compiles the new code for you. VisualAge for Java keeps track of both the source and its corresponding bytecodes. However, you never see .java or .class files in the IDE.

VisualAge for Java compiles source code incrementally. That is, it compiles only those parts of the source code that you change (and other code that is directly dependent on it) and therefore significantly reduces the overall compilation time.

Incremental Compilation

The VisualAge for Java IDE automatically compiles Java source code into Java bytecode. When source code is imported into the workspace (from .java files) or added from the repository, it is compiled and analyzed with respect to the existing contents of the workspace. Any errors are flagged and listed in the Problems page of program element browsers and the Workbench.

When you import Java bytecode classes (.class files), or add them from the repository, they are similarly analyzed with respect to the existing contents of the workspace, and errors are similarly flagged. However, you cannot edit bytecode classes whose source is not available in the workspace.

Other changes, such as deleting, moving, copying, or renaming program elements, also initiate a compilation of affected program elements, to flag any new problems.

When you make a change to the source code for a method, field, or class, the change and all affected code is compiled when you save the changes. If you introduce an error, the IDE will warn you and give you the option of fixing the problem immediately, or of adding the problem to the Problems page and fixing later. If you choose to fix it immediately, the IDE's code clues tool will suggest possible solutions, if it can determine them.

Compiled code is stored in the workspace, but not in the repository (except for those classes that were imported from bytecode files rather than source code files). If you delete a class from the workspace, it deletes the bytecode, while the source code is still stored in the repository. If you add it back to the workspace, it will be recompiled before you can work with it again.

Unresolved Problems

VisualAge for Java compiles source code incrementally. If you save source code that has a compiler error (also called an unresolved problem), VisualAge for Java displays a warning message. As a result, you see errors immediately.

You can cancel the save operation. In some cases, you can also save the source with the error. However, you cannot save the source in the following cases:

- a class or an interface has a Java syntax error
- a method has a Java syntax error outside the method body

When you fix an unresolved problem, VisualAge for Java fixes the problem throughout the workspace. This means that many unresolved problems disappear for you as you write code.

Even though a class contains unresolved problems, you may be able to run it. The debugger may open and suspend the program during execution.

Unresolved problems occur for many reasons, for example:

- the source code includes a Java syntax error
- the source code refers to a field, method, class, interface or package that is not declared (this could be as a result of deleting or renaming a method, field, class, interface, or package)
- the source code refers to a class or an interface that is not visible

- you import code and do not have all the packages on which the code depends

If you save a class that contains an unresolved problem, the class is marked with .

If you save a method that has an unresolved problem, the method is marked with . If the class of the incorrect method has no unresolved problem of its own, it is marked with .

VisualAge for Java maintains a list of all the classes and methods that have unresolved problems and updates it when you save new or modified code. You can view the list from the **Unresolved Problems** page of the Workbench and fix the problems there. As well, anywhere where you can select the class or method and see its source (except in a view of the repository) you can modify the source to fix the problem.

Workspace

All activity in VisualAge for Java is organized around a workspace, which contains the Java programs that you are developing. The workspace also contains all the packages, classes, and interfaces that are found in the standard Java class libraries, and other libraries that your classes may need.

[ENTERPRISE] In the team development environment, each VisualAge for Java client has its own workspace.

The workspace differs from the repository in the following ways:

- Program elements must be added to the workspace before they can be modified. Program elements that are in the repository can only be browsed.
- The workspace contains bytecode. The repository contains source code and VCE information.
- You can have only one edition of any program element in the workspace at any time. For performance reasons, your workspace should only contain the program elements that you are currently working on. By contrast, the repository contains every edition of every program element that you have ever developed, unless you have compacted the repository.
- You use the Workbench window to view, manipulate, create, modify, and manage program elements that are in the workspace. You use the Repository Explorer window to view program elements that are in the repository, add them to the workspace, and purge them from the repository.
- Changes to the workspace are not saved until you select **Save Workspace** from the **File** pull-down menu, or until you exit the IDE. Changes to the source repository are saved immediately, every time you save changes to a method, class, or interface.

When you start the IDE, the workspace is connected to the repository. The first time that you connect, VisualAge for Java builds pointers to the source code in the repository for every program element that exists in the workspace.

[ENTERPRISE] When you change repositories, this set of pointers is recached. You can not browse program elements that do not reside in the repository to which you are currently connected. You can always run code that is in the workspace, but if the connection to the repository is broken - for example by a server failure - then you can not browse source code or save changes.

You can add program elements from the repository to the workspace, replace the edition that is in the workspace with a different edition from the repository, or delete program elements from the workspace. You can maintain different versions of the workspace, customized for different projects or releases. See the list of topics below, for links to related information.

Repository

In the VisualAge for Java IDE, the repository is a source control mechanism that allows you to track changes made to program elements. When you start the IDE, it connects to a repository. As you create and modify program elements in the workspace, your changes are automatically stored in the repository. You can undo changes by retrieving previous editions from the repository.

Unlike the workspace, the repository contains all editions of all program elements. When you remove program elements from the workspace, they remain in the repository. Over time, the repository will grow. You should periodically purge program elements that are no longer required, and then compact the repository to reduce its size.

[ENTERPRISE] In the team environment of VisualAge for Java, Enterprise Edition, all team members' editions are stored in a shared repository on a server.

The Repository Explorer is the visual interface to the repository. Here are some examples of tasks that you can perform from the Repository Explorer window:

- Browse editions of projects, packages, classes, interfaces, and methods
- Compare different editions of program elements
- Add program elements to the workspace
- Change to another repository

Although the VisualAge for Java repository manages your code, it does not manage resource files, such as images and sound clips. Resource files are stored in the file system, and you are responsible for managing them. For more information, see the list of related topics at the end of this file.

[ENTERPRISE] In the team development environment, developers can use a *shared resource directory*, rather than keeping individual copies of resource files. This is unrelated to the repository.

Projects and Other Program Elements

The starting point for development work in the VisualAge for Java IDE is a *project*. Projects are units of organization used to group packages. They can be used, for example, to group packages from a certain provider, to group packages related to one application or customer, or to group frequently used classes that provide interrelated function. You can use them as best suits your development situation.

Packages and classes have the same meaning as in other implementations of the Java language. They are Java constructs. Projects contain packages, packages contain classes and interfaces, and classes contain methods. We refer to these constructs collectively as 'program elements'.

In the IDE, the following symbols are used to represent the different program elements:

- Projects
  - Packages
    - Classes or interfaces
      - Methods or constructors

The Workbench organizes all the program elements that are in the workspace. From the Workbench, you can view, create, modify, and manage program elements. You can also open browsers and other windows that help you perform specialized tasks on program elements.

The Repository Explorer organizes all the program elements that are in the repository. With the Repository Explorer, you can view all editions of all projects, packages, classes, interfaces, and methods that are in the repository.

The Standard Projects
The following projects are loaded into the workspace by default when you first install the IDE:

- IBM Java Implementation
- Java class libraries
- JFC class libraries
- Sun class libraries

Others that are shipped with VisualAge for Java are stored in the repository, but not initially loaded into the workspace. You can add them as you need them.

[ENTERPRISE] Every project, package, class, or interface has an owner who is responsible for the quality of that program element and is authorized to release it. Each edition of a class or interface also has a developer, who is the only person who can version that program element.

Importing Files from the File System

To import Java source code files, bytecode files, and resource files from the file system to the Workspace:

1. Start the Import SmartGuide by selecting **Import** from the **File** menu.
2. Select the **Directory** option (for regular files) or the **Jar file** option (for files stored in Jar or zip files).
3. Follow the instructions in the remaining SmartGuide pages to select the files and the target project.

Note: Because the Import SmartGuide can look into Jar and zip files, you do not need to unzip them before importing files that are in them.

Imported Java code is compiled and unresolved problems that are introduced are added to the All Problems page. Open editions of the classes and interfaces are created in the VisualAge for Java repository, and added to your workspace. If you select to automatically version program elements, then they will be versioned after importing.

Resource files (any file imported that is not a .java or .class file) are copied into the local resources directory for the project.

If you import only bytecode (compiled Java code in .class files), and not source code, for a class or interface, then you will not be able to edit the source in the workspace. These classes are indicated in the Workbench and other browser lists by the bytecode file symbol .

[ENTERPRISE] If the containing package or project in your VisualAge for Java workspace has been versioned, then a scratch edition of the package or project will be created. To prevent this, create an open edition of each project or package before importing.

Including Resource Files in a Project

When developing a program in Java, you sometimes need to use external resources that are not part of the language. For example, it is quite common to use images and audio clips in Java applets. VisualAge for Java does not store these external resources in the repository or workspace along with your Java source and byte codes. Instead you must create these resource files explicitly outside of the development environment and store them on the standard file system.

VisualAge for Java makes certain assumptions about where resource files are located. For a project called ProjectX, it assumes that the resource files will be found in a directory called `IBMVJava\Ide\project_resources\ProjectX` on the client machine (where IBMVJava is the install directory for the product). All projects in the workspace have a subdirectory of the `project_resources` directory, even if no resources have been created for the project.

Adding Resource Files to a Project
When you create a resource file, import it with the Import SmartGuide. This will copy the resource file into a project's resources directory. From there, it will be accessible to classes in that project.

Importing Packages the Have Resource Files
If you import a group of files from the file system, any non-.java, non-.class files can be imported into the project resources directory. In the Import SmartGuide, while importing files from the files system, enable the **Resources** option and click **Details** to select which resource files to import.

Exporting Resources
When you export a project to a JAR file, resource files are exported to the JAR file along with the classes. Also, when you select the Publish export option, resource files can be exported along with the bytecode classes to the target directory.

Running Applets and Applications that Use Resources
When running an applet or application from within the IDE, the default CLASSPATH contains the project resources directory so that it can find any resource files that the program uses.

When running an applet from within the IDE, the code base for the applet is specified as the name of the resource directory. The URL of this directory will be returned by the `java.applet.Applet.getCodeBase()` method.

[ENTERPRISE] Shared Resource Files
You can create a shared resources directory *in addition to the local one*, so that all team members use the same copy of the resource file.

When you run an applet in the IDE, it looks for resource files first in the default location on the local file system, and second in the shared resource directory, if one is specified. When you export a resource files with classes, the local resource files are exported by default. For example, if the project has a resource file called `picture.gif`that exists in both local and shared resource directories, then the local one is exported, regardless of the timestamps on the files.

Java Methods that Get Resources
Use the following methods, or others like them, to get resource files into your Java applets or applications:

- java.lang.ClassLoader.getResource(String) - Program can specify where to look for resource file
- java.lang.Class.getResource(String) - Looks for resource file on the class path (then code base, if in an applet)
- java.lang.ClassLoader.getSystemResource(String) - Looks for resource file on the class path
- java.applet.Applet.getAudioClip(URL) - Looks for audio file in the code base (applets only)
- java.applet.Applet.getImage(URL) - Looks for audio file in the code base (applets only)

See the related reference links below for API documentation for these classes.

Example
Applet `COM.ibm.ivj.examples.awttests.AwtBlueGreenGem` in project IBM Java Examples uses the file `terre.gif` and accesses it using `getImage('terre.gif')`. The default class path that VisualAge for Java uses to look for the file includes the `project_resources\IBM Java Examples` directory.

The default value for an applet's code base is the `project_resources` directory that corresponds to the applet's project.

The default value for a program's class path is the `project_resources` directory that corresponds to the program's project. As well, VisualAge for Java includes `program/lib` and `program/lib/CLASSES.ZIP` in the class path by default.

Internationalization in VisualAge

VisualAge supports two means of text separation by locale: list bundles and property files. A list bundle is a persistent form of *java.util.ListResourceBundle*. A property file is a persistent form of *java.util.PropertyResourceBundle*.

Both types of resource bundle contain key-value pairs. *ListResourceBundle.getContents( )* returns an array of key-value pairs. The key-value pairs stored as static strings in a property file are used to initialize the corresponding bean when it is loaded. Each resource bundle contains values for one (or a default) locale. The name of the bundle can be keyed by locale so that the virtual machine loads the appropriate resources for the current locale setting.

VisualAge supports the creation, editing, and use of resource bundles for all text found in a class. You can separate String property values as you set them from the Visual Composition Editor, or you can separate all text at once from the Workbench.

You can use your own resource bundles, or you can create them using VisualAge. You can edit existing resource bundles by hand or from within VisualAge. Multiple resource sources can be referenced within a single bean. VisualAge generates the appropriate code the next time you save the bean.

VisualAge does not separate text located in user code fields. To take advantage of programmatic string separation, move the user code into a separate method and call the method from within the user code block.

Separating Strings for Translation

Before doing this task, please read the conceptual information listed at the end of this topic.

From the Workbench, you can separate all String values from the class at once. From the Visual Composition Editor, you can separate String property values as you set them in the property sheet.

To separate String values from an entire class at once, follow these steps:

1. From the Projects page of the Workbench, select the class.
2. Select **Selected** and then **Externalize Strings**. Alternatively, click mouse button 2 and select **Externalize Strings** from the pop-up menu that appears.

   The Externalizing: Package.Class window appears, bearing a list of hardcoded strings found in the class.
3. Specify the type of resource bundle by selecting one of the following radio buttons:
   - **List resource bundle**
   - **Property resource file**
4. Specify the name of the resource bundle.
   - To choose an existing resource bundle, select the **Browse** button, pick a bundle from the standard dialog box, and select **OK**.
   - To create a new bundle, select **New**. Enter values as prompted, depending on the type of resource bundle; select **OK**.
5. If necessary, mark for exclusion those strings listed under **Strings to be separated** that should be left as is. To mark an item, select the graphic listed to the left of the column, as follows:
   - To separate the item, do nothing. ☑ **Translate** is already displayed.
   - If the item must never be separated, select ☑ once to display ☒ **Never translate**.
   - To leave the item hardcoded for now, select ☑ twice to display ? **Skip**.

   If you are not sure of an item, review it in the **Context** field.
6. Select **OK** to proceed with separation.

VisualAge marks each item marked ☒ with a special comment. To make a string previously marked ☒ appear in the externalization list once again, find the string inthe code and delete the comment at the end of the line: `//$NON-NLS-1$`. Then perform this task a second time.

Separating Strings through Property Sheets

To separate String property values as you set them, follow these steps:

1. Open the property sheet for each embedded bean that contains a text setting.
2. Select the value field to the right of the property name. A small button ... appears to the right.
3. Select the small button. The String Externalization Editor window appears. At the top of the window is a set of radio buttons that enable you to specify how you want VisualAge to handle the text. The current property setting, if any, appears in the **Value** field.
4. Select the appropriate radio button:
   - **Do not externalize string**
   - **Externalize string**
5. If you selected **Do not externalize string**, you are finished. Just select **OK** to close the window.
6. If you selected **Externalize string**, specify the type of resource bundle by selecting one of the following radio buttons:
   - **List resource bundle**
   - **Property resource file**
7. Specify the name of the resource bundle.
   - To choose an existing resource bundle, select the **Browse** button, pick a bundle from the standard dialog box, and select **OK**.
   - To create a new bundle, select **New**. Enter values as prompted, depending on the type of resource bundle; select **OK**.

   The name of the bundle appears in the **Bundle** list. If you selected an existing class, a currently defined key-value pair appears in fields below the bundle name.
8. To define a resource, type its name in the **Key** field. If the resource already exists, the corresponding value for the key appears in the **Value** field underneath; otherwise, the field is empty. To edit the resource, type a new value.
9. Select **OK** to close the window.

The next time you save the class, VisualAge modifies the generated get methods for the beans whose properties you just set as bundles.

Bean Interfaces and BeanInfo

The bean interface defines the property, event, and method features of your bean. These features can be used in visual composition when your bean is added to another bean. A BeanInfo class describes the bean and features that you add to the bean. Other features are inherited from the superclass of your bean unless you choose not to inherit features.

You can define a bean interface in the following ways:

- In the Workbench window, create a new bean class based on a class with features you need. By default, the new bean inherits the features of the class it extends. You can control feature inheritance by setting the **Inherit BeanInfo of**

bean superclass option in the **Design Time** pane of the Options window. Open the Options window from the **Window** menu of the Workbench.

- In the BeanInfo page, add new features to a bean. You can add features to extend the inherited feature set, to override inherited features, or both. When you add a feature to a bean, VisualAge generates code that describes the feature in the BeanInfo class for the bean.
- In the Visual Composition Editor, promote features of embedded beans to the interface of a composite bean. When you promote a feature of an embedded bean, VisualAge generates code that describes the promoted feature in the BeanInfo class for the composite bean.

When you create a new bean, it does not initially have a BeanInfo class. VisualAge creates a BeanInfo class when you add or promote the first feature that is not inherited, or when you explicitly direct VisualAge to create a BeanInfo class. You can create a BeanInfo class in the BeanInfo page.

Editions and Versioning

In VisualAge for Java, whenever you work with any project, package, or class, you are actually working with a specific *edition* of that program element. At any time, you can only have one edition of each program element in the workspace. To see which editions are in the workspace, click the **Show Edition Names** button.

[ENTERPRISE] You will usually work with open and versioned editions; occasionally, you may also create scratch editions of program elements to experiment with. You will periodically release editions of classes and packages that you have been working on, to provide a baseline for the team and to make your changes easily available to them. Editions, releasing, and ownership are all fundamental to managing application changes in the team environment. Editions are discussed below; releasing and ownership are discussed as separate topics.

Open Editions
Open editions are works in progress. Before you can make changes to an existing project, package, or class, you must create an open edition of it. You can have multiple open editions of the same program element, with each one implemented differently. For example, if you are adding features to an application that you have customized for different industries, you might have multiple open editions of a package with the same name.

Open editions appear in VisualAge for Java windows with a timestamp, in parentheses, showing when they were created. Here is an example:

```
PackageA (3/28/98 4:21:15 PM)
```

Versioned Editions
Versioned editions are editions that can not be changed. You version your open editions for the following reasons:

- To keep a copy of a program element at some meaningful point, so you can return to it at a later date. In the case of packages and projects, versioning freezes a specific configuration of the contained program elements, which must also be versioned.
- [ENTERPRISE] To make your changed classes available to other team members who are browsing the repository.

- [ENTERPRISE] To release a class into its containing package, thereby updating the team baseline. Classes must be versioned before they can be released.

Versioned editions appear in VisualAge for Java windows with version names, as opposed to the timestamps that identify open editions. When you version an open edition of a program element, VisualAge for Java can automatically assign a name for you, or you can specify your own name. Here are some examples of versioned editions:

```
PackageA 1.6.1
PackageB VersionBRel2
PackageC JS - Fixed print problems for CustomerX
```

Versioning does not prevent you from ever changing a program element again. To make changes, create a new open edition of the program element. To revert to an earlier version, replace the edition in the workspace with a different edition from the repository, and create an open edition based on that.

You will probably version your classes frequently, whereas you may leave packages and projects open for extended periods of time.

[ENTERPRISE] In the team development environment, version control is achieved by means of releasing editions into a team baseline. Only program element owners can release. See the list of related topics at the end of this document for links to more information on ownership, baselines, or releasing.

[ENTERPRISE] Scratch Editions

Scratch editions are editions that no other users of the shared repository can see. Scratch editions appear in VisualAge for Java windows with < > around the edition name:

```
PackageA <1.0>
```

Scratch editions are discussed separately.

[ENTERPRISE] Undefined Editions

You may see a class or interface whose edition name is 'undefined edition':

```
PackageA undefined edition
```

This means that someone has created a class or interface, but has never versioned or released it. VisualAge for Java has reserved the new program element's name in the shared repository. Such editions are also marked with the undefined symbol.

[ENTERPRISE] Tools for Managing Your Editions

VisualAge for Java provides two tools for working with editions in a team development environment:

- The Managing page of the Workbench window consolidates information about all the editions that are in the workspace, and is a convenient place to perform activities such as versioning and releasing.
- The Management Query tool helps you search for program elements in the workspace by edition status. Open it by selecting **Management Query** from the **Workspace** menu.

You can also view edition details, such as status and ownership, by selecting **Properties** from a program element's pop-up menu.

The Scrapbook

The Scrapbook is a window that helps you organize, develop, and test ideas for your Java programs. In the Scrapbook, you can experiment with Java code fragments without specifying a containing class. The Scrapbook can have several pages. Contents of the pages may be saved to files, but are not saved in the repository.

The Compilation Context
The *compilation context* is the class you choose to contain the Java code fragment when you compile and run it. When you evaluate code in the Scrapbook, it is treated as though it were part of the compilation context, so the code inherits any imports from the selected class. It can also refer to protected or private fields and methods in the class and nonpublic classes in the same package.

Each Scrapbook page has its own compilation context, and does not interfere with other pages, so you can do things such as test the client and server parts of a program. The default compilation context is java.lang.Object.

Other Uses for the Scrapbook
The Scrapbook can open, read, and save to text-based files, including .java files, from the file system. It supports a variety of formats and origins, including NT, OS/2, UNIX-based, Solaris, and Macintosh. You can run Java code that is in any of these types of files.

You can import Java code from these files into a class, interface, or method in the workspace.

The Scrapbook is also useful as a simple text editor for viewing and editing text-based files within the file system, and for making notes to yourself.

Organizing Your Ideas in the Scrapbook

The Scrapbook window is a flexible text editor and test environment within the IDE. It lets you open and work with text-based files from the file system. It also lets you create your own notes to yourself, and it provides a test area for Java code fragments.

A Scrapbook page can contain any text you want. You can open files, copy and paste from other locations inside and outside the IDE, and write Java code or free-form text. For this reason, it is a good place to keep a to-do list and reminders to yourself, along with ideas for code. Contents of the pages are saved to files, not to the repository.

Opening the Scrapbook
To open the Scrapbook, select **Scrapbook** from the **Window** menu of any IDE window. Each time you start the Scrapbook, it will have only one blank page, regardless of what files were open when you closed the Scrapbook; you must open files you need again. For this reason, lock the Scrapbook open during a development session so that you do not inadvertently close it.

Adding and Removing Pages
Using multiple pages in the Scrapbook is a good way to organize your ideas and to provide separate testing contexts for different fragments.

To add an empty page, click the New Pagebutton on the Scrapbook toolbar.

To remove a page, select its tab, and then click the Delete Pagebutton . You cannot delete a page if it is the only one left.

Opening a File
To open a text-based file in the file system, select **Open** from the **File**menu, or use the shortcut key Ctrl+O. This creates a new Scrapbook page that contains the specified file. The title of the page is the name of the file.

You can edit this file and save your changes back to the file system by selecting **Save** from the **File** menu. When you are finished with the file, delete its page from the Scrapbook. Removing the page that contains a file does not delete the file from the file system; it removes it from the workspace.

Experimenting with Code Fragments

There may be times you have a fragment of Java code that you want to try out before adding it to a class or project in the workspace. Along with being a good place to keep notes and lists of ideas, the Scrapbook also provides a flexible environment for testing and experimenting with any piece of Java code. Contents of Scrapbook pages are saved to files, not to the repository.

To open the Scrapbook, select **Scrapbook** from the **Window** menu of any IDE window.

Running Code Fragments in the Scrapbook
To run a code fragment in the Scrapbook:

1. On a page in the Scrapbook, type in Java expressions and statements, or copy and paste them from another source. Alternatively, open a file that contains Java code.
2. Select all or part of the code on the page by highlighting it with the cursor. Only the highlighted code will be run.
3. Select the **Run** button from the toolbar.

While the page is running the code fragment, its page symbol changes to indicate that it is busy. No other code can be run from that page while it is busy.

Resetting a Busy Page
To stop a running code fragment:

1. Click the tab for a Scrapbook page that is running a code fragment.
2. Select **Restart Page** from the **Page** menu.

When a page is reset, all threads started from the page are terminated, and all classes in the code are uninitialized. The compilation context and the text on the page do not change.

Changing the Compilation Context
When the code fragment runs, the IDE assumes it belongs to a particular class and package that exists within the workspace. This class provides a *compilation context* for the fragment, which determines what other program elements are inherited by and accessible to the fragment. Each Scrapbook page has one compilation context, which is by default `java.lang.Object`.

To select another compilation context:

1. Click the tab for the Scrapbook page for which you want to change the context.
2. Select **Run In** from the **Page** menu in the Scrapbook.
3. Select a package and a class or interface from the lists.
4. Click **OK**.

The information line at the bottom of the Scrapbook page indicates the compilation context for the page. All code fragments run from the page will use this compilation context.

Inspecting and Debugging Code Fragments
Just as you can inspect variables for code in the workspace, you can also inspect variable values in code fragments in the Scrapbook. To inspect a code fragment in the scrapbook, select the code that you want to inspect, and then click the **Inspect**button .

If the selected code returns a result that can be inspected, an Inspector window is launched. If the selected code does not return a result that can be inspected, the following message is displayed on the page:

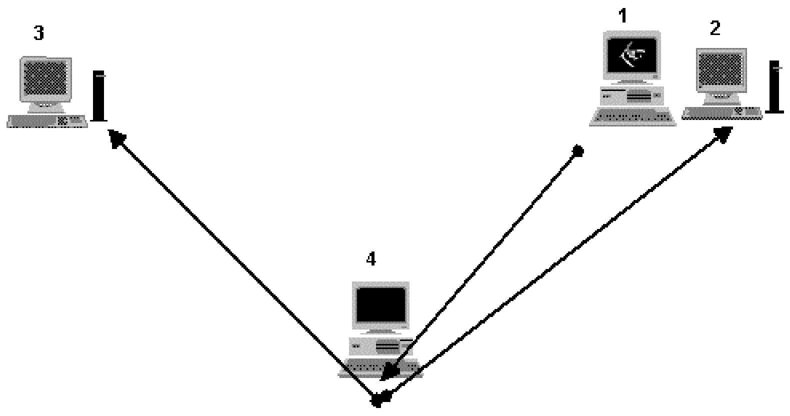
```
No explicit return value.
```

Similarly, you can debug the code fragment by using the integrated debugger. To launch the debugger, select the code you want to debug, and then click the **Debug** button .

VisualAge for Java for the Network Station

VisualAge for Java provides a productive environment for developing applications for the IBM Network Station. The VisualAge for Java IDE lets you develop, debug and prototype your code before deploying it to a Network Station. Deploying the code to the Network Station is as easy as exporting the compiled files to a shared directory on the Network Station Manager and configuring the application.

The following figure shows the suggested configuration for application developers, testers and users of the system.



The following are the key points of this application development scenario:

1. Developers have application development machines that support VisualAge for Java.
2. Developers have Network Stations to test their developed code in the Network Station environment.
3. Each tester or user of the system has a Network Station.
4. There is a single Network Station Manager that manages all of the network stations and has a shared directory that is exported through its web server.

The Integrated Debugger

Use the integrated debugger to debug applets and applications running in the IDE. In the debugger, you can view running threads, suspend them, and inspect their visible variable values.

You can open the debugger manually or you can have it open automatically by setting breakpoints or specifying caught exceptions to notice. It will also open automatically if an uncaught exception is thrown.

When a program is running, the debugger lists running threads. Suspend a thread to view a list of methods that represent the current stack state. Select one of these methods to inspect the visible variables in that stack frame.

While a thread is suspended you can do a number of things:

- View and modify visible variable values.
- Edit most methods' source code.
- Step into, over, or to the return statement of a method.
- Drop to a selected stack frame.
- Replace most methods with another edition.
- Evaluate expressions.
- Resume running the thread.
- Terminate the thread.

The debugger can work with multiple running programs concurrently, which can be useful, for example, when you are debugging client and server portions of an application. In the Debugger window, threads are grouped by the program that started them, for easy manipulation.

The integrated debugger lets you set regular or conditional breakpoints, and it supports debugging inner classes. Also, it can optionally generate a class loading and initialization trace.

Debugging during the Development Cycle with the Integrated Debugger

The integrated debugger assists in debugging applets and applications running in the IDE. It is a live debugger: it always shows the exact current state of running programs.

The Integrated Debugger browser displays the following information:

- On the Debug page:
  - All currently running threads, grouped by program.

  - When a running thread has been suspended:
    - The methods in the thread.
    - The visible variables and their values for the methods.
    - The source code for the methods.
- On the Breakpoints page:
  - All methods in the workspace that have breakpoints set in them.
  - The source code for the methods.

From the Debugger, you can launch Inspectors to look at and modify variable values for suspended threads.

You can open the debugger manually while a program is running to inspect threads and variables. As well, the debugger will automatically open, with the current thread suspended, for any of several reasons:
- A breakpoint in the code is encountered.
- A conditional breakpoint that evaluates to true is encountered.
- An exception is thrown and not caught.
- An exception selected in the Caught Exceptions dialog is thrown.
- A breakpoint in an external class is encountered.

Once the debugger is open and a thread is suspended you can work with the program in the following ways:
- Inspect visible variable values.
- Modify most variable values.
- Step through methods.
- Modify source code for methods in the workspace.
- Replace methods with other editions from the repository.
- Modify, clear or disable breakpoints.
- Evaluate expressions in the source pane.

Using the debugger, you can optionally generate and view the class loading and initialization trace.

Opening the Integrated Debugger Manually

To open the Integrated Debugger browser, select **Debug > Debugger** from any Windowmenu or click the Debugtoolbar button . This will open the Debugger browser to the **Debug** page, which lists running threads, their states, and their methods’ visible variables.

If you click the **Breakpoints** tab, or if you selected **Debug > Breakpoints** from the **Window** menu, then the Debugger browser opens to the Breakpoints page, which summarizes all methods in the workspace that have breakpoints set.

If one or more program are running when you open the Debugger, then the running threads are listed in the All Programs/Threads pane of the Debug page. If you suspend the thread, you can look at the methods on the thread, and inspect the methods’ visible variables.

If you leave the Debugger open and start a program, the new program's threads will be added to the All Programs/Threads pane.

Suspending, Resuming, and Terminating Threads

When one or more programs are running, the Debug page in the Debugger browser shows all running threads, grouped by program. You can suspend, resume, and terminate the threads as needed.

Suspending a Running Thread
To examine a thread at any point while it is running, you must suspend it manually. Then you can modify or step through its methods and inspect its variables.

To suspend a thread manually:

1. Open the Debugger browser.
2. Select the thread in the All Programs/Threads pane in the Debug page.
3. Click the **Suspend** button in the toolbar.

Threads halted because of a breakpoint or uncaught exception are suspended automatically.

Once a thread is suspended, expand it in the All Programs/Threads pane to view its method stack. Select a method to inspect its visible variables or work with its code. In the source pane, you can see how the current value of a variable is being calculated by holding the mouse pointer (I-bar) over the variable in the source. After about a second, a pop-up dialog will appear, showing the information.

Resuming Running a Thread
When a thread has been suspended, either by the manual method above, or automatically when the Debugger opens for a break point or uncaught exception, you can resume running the suspended thread. It will run until suspended manually or automatically, or until the thread terminates.

To resume running a thread:

1. Select the suspended thread.
2. Click the **Resume** button on the toolbar.

The running is resumed at the point it left off, unless one of the following things has been done while the thread was suspended:

- You replaced a method, in which case it starts again at the start of the method, maintaining all changes to the state of the object that have been made.
- You stepped through code, in which case it starts again after the last step taken.

The program will continue running until it is suspended again or until it terminates.

Note: You can not resume running after an uncaught exception.

Terminating a Thread
When you terminate a thread, it is removed from the Debugger browser and cannot be suspended or resumed. Often, terminating a thread will stop the running of the program. To terminate a thread:

1. Select the thread in the All Programs/Threads pane of the Debug page.

2. Click the **Terminate** button on the toolbar.

The thread will terminate. To restart the thread, you must restart the program from the beginning.

Alternative Methods of Suspending, Resuming, and Terminating
Besides using the toolbar buttons described above, you can select **Suspend**, **Resume**, and **Terminate** menu options from the **Selected** menu or the pop-up menu for a thread. Also, pressing F8 resumes a suspended thread.

Setting Breakpoints in Source Code

A breakpoint is a signal to the IDE's Integrated Debugger to suspend a program thread that is running the code that contains the breakpoint. When the IDE is running a program and encounters a breakpoint, it suspends the thread (the program temporarily stops running), and the Debugger browser opens so that you can see the run-time stack for the thread.

Breakpoints can be set on any instruction in source code in the workspace. They can be set at any time, including while code is being debugged; that is, you can add a breakpoint to a method in a suspended thread's stack without being dropped to the top of the method.

To set a breakpoint in source code in the IDE:

1. Go to the Workbench or any browser that shows the source code for the place in the program where you want to suspend the thread.
2. Place the cursor in the line.
3. From the **Edit** menu, select **Breakpoint**

A breakpoint symbol is placed in the margin of the Source pane next to the line in which you placed the cursor. If you try to set a breakpoint at an invalid location (for example, a comment line), the breakpoing will be set at the closest valid location. If you try to set a breakpoint in a method in which breakpoints cannot be used, a dialog will inform you that there are no valid locations in the method to set a breakpoint.

Setting and Clearing Breakpoints by Double-Clicking
Now that you can see where the breakpoint symbol is placed (in the margin of the Source pane next to the code), you can learn the short-cut to setting and clearing breakpoints.

If you set a breakpoint by using the **Breakpoint** menu option in the **Edit**menu, clear it by double-clicking on the symbol. You can likewise set a breakpoint by double-clicking on the margin next to the desired line.

The Breakpoints Page in the Debugger Browser
The Breakpoints page in the Debugger browser shows a summary of all breakpoints that have been set in the workspace. To open this page, select **Debug > Breakpoints** from the **Window** menu. The Methods pane lists all the methods in the workspace that contain breakpoints, and the source pane shows the source code for the method selected in the Methods pane. From this page, you can set breakpoints in methods that already have a breakpoint, and you can remove breakpoints. You can also disable breakpoints (which means that the breakpoints remain set, but the IDE ignores them when running programs).

Other Types of Breakpoints
Breakpoints can be modified so that they only suspend the thread under certain circumstances; these are called *conditional breakpoints*. You can configure the breakpoint to perform an action or evaluation, which, if results in a true value, will suspend the thread. Also, you can set breakpoints on methods in external classes (classes that reside on the file system and that are called by programs running in the IDE that have the external classes in their class path). See the related tasks below for information on setting these breakpoints.

Configuring and Setting Conditions on Breakpoints

Conditional breakpoints are breakpoints that suspend code and open the debugger *only when certain conditions are met*. For example, you can set a breakpoint to suspend code only if a variable’s value falls in a particular range of values.

To set conditions on a breakpoint, click mouse button 2 on the breakpoint symbol , and select **Modify** from the pop-up menu.

In the dialog that appears, you can select a condition from the drop-down list, or you can type in your own condition. (The drop-down list contains up to ten conditions you have previously set on breakpoints). *If the condition is evaluated to a boolean value of true, then the breakpoint suspends the code and opens the Debugger browser.*

Configuring the Breakpoint to Do Something
Similarly, you can configure a breakpoint to run a Java statement and *then* return true or false. For example, when the IDE encounters the breakpoint, you can have it output a message and then evaluate to false, thereby not suspending the code.

The text entry field in the dialog has code assist support; if you type in the start of a package or class name, press Ctrl+Spacebar to get a pop-up list of available classes or methods. Select the desired one by continuing to type or by using the arrow keys, and press Enter.

Setting Breakpoints in External Classes

The VisualAge for Java IDE can run programs that dynamically load and run external classes. *External classes* are classes that have not been imported into the workspace, but rather reside in a .class file, Zip file, or Jar file on the file system. The path to the file must be part of the class path for the program.

If you want to debug such a program, you have the option of setting breakpoints on methods in the external classes.

To set a breakpoint on a method in an external class:

1. Select **Debug > External .class file breakpoints** from the **Window** menu, or click the **External Breakpoints** toolbar button in the Debugger browser.
2. The External Method Breakpoints dialog shows a list of methods available for setting breakpoints. Add methods to the list by clicking **Add**.
3. The Add External Methods dialog looks into .class and archive files and lets you select methods within those files to add to the list of methods available for setting breakpoints. To access methods in a .class file:
   a. Click **Directory**.

   b. Browse through the file system to the directory that contains the .class files in which you want to set breakpoints.

   To access methods in a .class file that has been archived:

   a. Click **Archive**.
   b. Select **Zip Files (*.zip)** or **Jar Files (*.jar)** in the **Files of Type** drop-down list.
   c. Browse to the archive file that contains the .class files in which you want to set breakpoints.

   Now, the dialog lists all the .class files in the selected directory or archive. Select one to see the list of methods available for setting breakpoints.

4. If you want to add one of the listed methods to the list of methods available for setting breakpoints, enable its check-box.
5. When you have selected all the methods you want, click **OK**. The list of methods in the External Method Breakpoints list now shows the methods you selected.
6. To set a breakpoint on one of these methods, enable its check-box. The breakpoint is not enabled simply by the method being listed in this box; its check-box must be enabled for the breakpoint to be set.
7. Click OK to exit the dialog.

Once the breakpoint is set, any thread that calls it will be suspended when the method is entered. External breakpoints cannot be conditional and do not display the breakpoint symbol in the source pane margin.

Putting Source Code on the Debug Source Path
If the source code for the method is available on the file system, and if the path to the source code is included either in the class path or in the debug source path (set in the Debugger Options), you will be able to step into the method code. The debugger looks for the source code first in the class path, and if it cannot find it there, then in the debug source path.

If the source is not available (not on either the class path or the debug source path), you will only be able to step over the method.

Clearing Breakpoints on External Methods
To clear a breakpoint from an external method, clear its check-box in the External Method Breakpoints dialog. You may leave the method on the list so that it is easily accessible if you want to set the breakpoint again later. If you want to remove it from the list, however, select it and click **Remove**.

Selecting Exceptions for the Debugger to Catch

Usually, if an exception is thrown while a program is running, and the program does not catch it, the IDE Debugger opens and the offending thread is suspended. However, if the program does catch it, the debugger will not open, and the program will continue. Even if the program outputs the stack trace when it catches the exception, you might not be able to determine its origin.

To make debugging easier, the IDE debugger lets you effectively set breakpoints on exceptions, so that any time an exception of a certain type is thrown, the debugger suspends the thread that threw it and opens the Debugger browser. You can then see where the exception is happening.

To select a type of exception to be caught by the debugger:

1. Select **Debug > Caught Exceptions** from the **Window** menu, or click the **Caught Exceptions** toolbar button in the Debugger browser.
2. From the list of available exception types, select the types of exceptions you want to set breakpoints on by enabling their check-boxes. **Note:** You can select how the exception types are listed: by exception type name, or by package name.
3. Click **OK**.

Now when you run a program that throws an exception (of the type you selected), the thread is suspended and the Debugger browser opens, regardless of whether the program catches the exception.

Clearing and Disabling Breakpoints

Once a breakpoint is set, you can remove it at any time, including while you are debugging the code it is in. If you remove a breakpoint from a method while the thread it is in is suspended, the debugger does not drop to the top of the method.

To see a summary of all breakpoints in the workspace, go to the Breakpoints page of the Debugger browser by selecting **Debug > Breakpoints** from any **Window**menu. Select a method in the list to see its source code and the breakpoints.

Clearing Breakpoints
To clear a breakpoint in source code, double-click on its symbol in the margin of the Source pane. You can remove breakpoints from any Source pane (not just the one in the Breakpoints page in the Debugger browser).

However, if you are in the Breakpoints page, you can use the following toolbar buttons to clear breakpoints:

	Clears all breakpoints in the currently selected method; removes the method from the Breakpoints page.
	Clears all breakpoints in the workspace.

Disabling Breakpoints
Suppose you want to run a program that has breakpoints set throughout its code, but you do not want the debugger to open during the running. You can disable the breakpoints by clicking the **Enable Breakpoints** toolbar button so that is is in the 'up' position, as shown: . The IDE will ignore all the breakpoints it encounters. (The debugger may still launch if an exception is thrown and not caught.) All debugger symbols in the margin of Source panes will change colors from blue to gray.

To re-enable all the breakpoints in the workspace, click the **Enable Breakpoints**button so that it is in the 'down' position, as shown: .

Clearing Breakpoints in External Classes and Caught Exceptions
To clear a breakpoint on a method in an external class:

1. Select **Debug > External .class file breakpoints** or **Caught Exceptions** from the **Window** menu.
2. Remove a breakpoint on a method or exception by disabling its checkbox. In the case of external methods, the method will remain in the list, in case you want to place a breakpoint on it again. To remove the method from the list, select it and click **Remove**.
3. To remove breakpoints from all external methods or caught exceptions, click **Clear All**.
4. Click **OK**.

Inspecting and Modifying Variable Values

When a thread has been suspended, the Debugger browser displays all running methods and the variables visible within them.

Select a method in the All Programs/Threads pane. The Visible Variables pane shows the variables in use. You can change which variables are shown by changing the selections in the **Inspector** menu. Variables that themselves contain fields can be expanded to show the fields by clicking the plus symbol ⊞ in the tree.

When you select a variable in the Visible Variables pane, its current value (at the exact point in the program where it was suspended) is shown in the Value pane. If you select multiple variables, the values for each are shown in the Value pane. To select multiple variables:

- Select one; hold down the Shift key; select another: the two variables, plus all variables between them in the list are selected; or
- Select one; hold down the Ctrl key, select another: the two variables are selected; their values are listed in the order you select them. Continue holding down the Ctrl key and selecting.

Opening an Inspector
To closely look at one variable that contains fields, select it in the Visible Variables pane and click the Inspect button in the toolbar. An Inspector window will open, showing the variable’s fields, and their values. This information is the same as the information in the Debugger browser Visible Variables and Values panes, and you can select, view, and modify the contents in the same ways.

Modifying Variable Values While the Program is Running
The values of variables can be modified while the thread is suspended. To modify a variable’s value in the Value pane :

1. Select the variable in an Inspector window or in the Visible Variables pane .
2. Edit the value shown in the Value pane.
3. Select **Save** from the Value pane’s pop-up menu.

Alternatively, you can modify the value right in the source pane. For example, if you have an integer variable called depth that has a current value of 4, and you want to change its value to 6, do the following steps:

1. Anywhere in the source pane, type in `depth=6`.
2. Highlight `depth=6`.
3. Select **Run** from the **Edit** menu. This evaluates the expression and changes the value of the variable.

4. To see that the value has changed, find an occurrence of depth in the Source pane and hold the mouse pointer over it for about a second. A pop-up lable will appear with the label 'depth=(int) 6'.
5. Delete the fragment, `depth=6`.

The change in variable value is immediately available to the running program. When you resume running, the *new* value is used.

Stepping through Methods

When a running thread is suspended at a certain place in the program, the Source pane in the Debug page indicates the point at which the execution stopped by highlighting the corresponding code. You can move forward through the code, step by step, in a variety of ways.

Toolbar Icon	**Selected Menu Option**	**Description**
↓	Step Into	Steps into the current statement, and if the statement calls a method, it adds the method to the stack and stops execution on the first line of the method. If the method is in an external class, but if source is available on the Debugger class path, then this works as though the method were in the workspace; if the source is not available, the external method will be stepped over. Each time you click Step Into, the debugger steps into each method called, adding and removing each to and from the stack as they are stepped through. If you step into a statement that does not call a method, the effect is the same as stepping over the statement.

⟳	Step Over	Runs the current statement, including all methods called within the statement. Stops before the next statement. If you step over a method that takes a significant amount of time to run, the string '`/* Thread is currently stepping*/`' will be inserted into the Source pane. You may wait till it returns or click the **Resume** button ▶ to stop debugging it.
⟳	Run To Return	Runs the current method up to the return statement, and stops before returning to the statement that called the current method.
	Run To Cursor	Resumes running up to the statement where you have placed the cursor in the Source pane.
	Drop to Selected Frame	Resets thread execution to the start of the selected method.
▶	Resume	Runs to next breakpoint, until you manually suspend thread, or to the end of the program.

Note that execution may stop earlier than indicated above, if the debugger encounters a breakpoint or an exception.

Modifying Code While Debugging

When a thread has been suspended, most of the methods on the stack can be edited or replaced by another edition of the method (methods required by the system or those in external classes may not be modified).

To edit a method on the stack:

1. Select it in the All Programs/Threads pane.
2. Edit its source code in the Source pane as required. The Source pane has code-assist; type Ctrl+Spacebar to get help with method and field names.
3. Select **Save** from the Source pane’s pop-up menu.

To replace a method with another edition:

1. Select it in the All Programs/Thread pane.
2. Select **Replace With > Previous Edition** or **Replace With > Another Edition** from the method’s pop-up menu.
3. If selecting with another edition, select the desired edition.

In either case, when you resume running the program, execution will drop to the beginning of the method; any side effects of running the method before are not undone.

Evaluating Expressions in the Integrated Debugger

When a thread is suspended in the debugger, you can view the source code for all methods in the thread, and you can evaluate any expression in the source code to see what its value is, given the current values of visible variables in the program.

Using an Inspector Window
To evaluate an expression in the debugger and display the results in an Inspector window:

1. In the Debug page of the Debugger browser, suspend a thread in a running program.
2. In the All Programs/Threads pane, select a method in the suspended thread. Its source code will be shown in the Source pane of the browser.
3. Select an expression that will evaluate to some value.
4. Click the **Inspect** button , or select **Inspect** from the selected code’s pop-up menu.

An Inspector window will open to show the value of the expression, given the current values of the program’s variables.

Displaying the Value In-Line
To evaluate an expression and display the result in the text of the Source pane:

1. In the Debug page of the Debugger browser, suspend a thread in a running program.
2. In the All Programs/Threads pane, select a method in the suspended thread. Its source code will be shown in the Source pane of the browser.
3. Select an expression that will evaluate to some value.
4. Select **Display** from the selected text’s pop-up menu.

The value of the expression, along with the value’s type, will be output as selected text in the Source pane. Press the Delete key to remove the highlighted text.

Note: This technique can be used to evaluate expressions in the Scrapbook as well. In the Scrapbook, the toolbar button displays the resulting value in-line.

Evaluating expressions is useful, for example, for debugging if-statement and loop conditions that are producing unexpected results.

Generating the Class Trace

The debugger will generate a trace of class loading and initialization, if you enable the Class Trace option. The class trace is useful for determining which classes your program uses, and can help in debugging.

To turn the trace on:

1. Select **Options** from the **Window** menu.
2. Select the Debugging page.

3. Enable the **Trace class initialization for running programs** checkbox.
4. Click **OK**.

Note: When this option is enabled, some processing time is required to compute and store the trace. As a result, the program may run significantly more slowly when this option is enabled.

To see the trace:
1. Open the Debugger browser by selecting **Debug > Debugger** from the **Window** menu.
2. In the Workbench or a browser, start a program running.
3. While the program is running, select the program (not a thread) in the All Programs/Threads pane.
4. The trace will be shown in the Source pane. Copy the contents to the clipboard to save them.

As soon as the program terminates, the trace will disappear from the Source pane. If your program runs quickly, you may want to add a breakpoint near the end of the program's code so that the program will stay in the debugger's list of active programs long enough for you to view the trace.

External SCM Tools (Windows)

The VisualAge for Java IDE offers an interface for checking .java source files in and out of an external software configuration management (SCM) system. This interface is a complementary feature that you can select when you install VisualAge for Java. It supports the following SCM tools:
- ClearCase 3.2 for Windows NT, from Rational Software Corporation
- PVCS Version Manager 6.0, from INTERSOLV, Inc.
- VisualAge TeamConnection Version 3.0, from IBM Corporation

The interface from VisualAge for Java to external SCM tools uses Microsoft's Source Code Control (SCC) API. It is supported for Windows NT and Windows 95 clients. It may work with other SCC-compliant SCM tools, but IBM has only tested the products and releases listed above.

If you selected the interface to external SCM tools when you installed VisualAge for Java, the Workbench window provides a menu for adding classes to source control, checking classes in and out of the SCM repository, and importing the most recently checked-in version of a class from the SCM repository. Prior to VisualAge for Java, Version 2.0, if you wanted to check a class into your SCM tool's repository, you had to do the following steps:
1. Export the classes from the VisualAge for Java repository to the file system
2. Launch your SCM client program from outside the IDE
3. Use your SCM client program to check in the files that you created when you exported from VisualAge for Java

To check out a class, you would reverse the process. By contrast, with the interface to external SCM tools, the intermediate import and export steps are automated so you only need to select **Checkin** or **Checkout** from a menu in the IDE.

The interface from VisualAge for Java to external SCM tools does not provide any automatic synchronization of version names between the VisualAge for Java repository and the external SCM tool. VisualAge for Java does not prevent you from changing a program element in your workspace if you have omitted to check it out in the external SCM tool. The External SCM menu provides a convenient way for you to use an existing SCM tool without leaving the VisualAge for Java IDE, but you will need to correlate the functions of the two systems.

See the table below for a comparison of terms used by different SCM programs.

[ENTERPRISE] External SCM Tools and VisualAge for Java Team Development
VisualAge for Java, Enterprise Edition, provides a team development environment that uses a shared, object-based source code repository. This is VisualAge for Java's implementation of SCM; it provides software configuration support for development projects where multiple programmers work on the same code at the same time, and where they may need to support multiple versions. This shared repository implementation, sometimes called ENVY, is used by VisualAge Smalltalk and has become a de facto SCM standard for team development in Smalltalk environments.

Version control and repository management are integrated into VisualAge for Java, Enterprise Edition. The shared repository offers excellent support for day-to-day team programming activities. Even so, you may wish to install external SCM support as a complementary feature for one of the following reasons:

- You already use another SCM tool as your standard for application development.
- You have established practices for archiving applications on a particular enterprise server, for example for disaster recovery purposes.
- The repository in VisualAge for Java manages Java objects only; you may wish to manage all of your development artifacts with a single tool, or to integrate multiple programming languages across your environment.
- VisualAge for Java, Enterprise Edition, allows programmers to work concurrently on the same class. (Each programmer works in a separate, unique edition of the class, and class owners must approve changes by releasing them.) This approach encourages programmers to think in terms of objects rather than files, and it fosters team communication. Nonetheless, you may be more comfortable with a traditional file checkin/checkout approach that enforces serial development of classes.

Comparison of SCM Terms
As you use the VisualAge for Java interface to external SCM tools, the following table may help you to correlate the terms that you encounter.

VisualAge for Java's interface to external SCM tools	**PVCS**	**ClearCase**	**VisualAge TeamConnection**
SCM repository	archives	data repository	repository
project	project	combination of VOB + view	combination of family + release + component + work area; sometimes known as *version context*
check in	check in	check in	check in part
check out	check out	check out	check out part
undo checkout	unlock revision	undo checkout	unlock part

add to source control	create archive	add to source control	create part
get latest	check out the tip (latest version) with no read or write lock	not applicable	extract part
comments	change description	comments	remarks

Exporting Code

Once you have finished developing, testing, and debugging your project within the IDE, export it to the file system by using the Export SmartGuide. The following options are available to you when you export:

- Export code to a directory in the file system.
- Export code to a Jar file.
- [ENTERPRISE] Export versioned packages and projects to another repository.

Exporting Code to a Directory

To export code to a directory in the file system:

1. Select the project, package, or type from which you want to export.
2. Select **Export** from the **File** menu.
3. In the Export SmartGuide, select the **Directory** radio button. Click **Next**.
4. Provide a target directory for the export. Exported projects will create subdirectories of the target directory.
5. Select the types of code you want to export (bytecode, source code, or resource files), and click **Details** to select the specific classes, interfaces, or resource files.
6. If you are exporting an applet and you want to generate an HTML file to launch it, enable the **.html** option and click Details to select the applets for which you want the HTML launch file.
7. Select other options, if needed, and click **Finish**.

Exporting Code to a Jar File

To export code to a Jar file:

1. Select the project, package, or type from which you want to export.
2. Select **Export** from the **File** menu.
3. In the Export SmartGuide, select the **Jar file** radio button. Click **Next**.
4. Provide a target Jar file name for the export. If you specify one that already exists, it will be overwritten.
5. Select the types of code you want to export (bytecode, source code, resource files, or beans), and click **Details** to select the specific classes, interfaces, resource files, or beans.
6. If you are exporting an applet and you want to generate an HTML file to launch it, enable the **.html** option and click Details to select the applets for which you want the HTML launch file.
7. Select other options, if needed, and click **Finish**.