SNiFF+™

Version 3.2 for Unix and Windows

Java Tutorial

TakeFive software
An Integrated Systems Company

TakeFive Software, Inc.
Cupertino, CA
E-mail: info@takefive.com

TakeFive Software GmbH
5020 Salzburg, Austria
E-mail: info@takefive.co.at

Credits

The first version of Sniff was developed at the Informatics Laboratory of the Union Bank of Switzerland. Its development was considerably facilitated by the public domain application framework ET++.

Authors of the first version:

Walter Bischofberger (Sniff)

Erich Gamma (Sniffgdb)

Erich Gamma and André Weinand (ET++)

Table of Contents

Part I
Guidelines

About this Manual 1

What this manual is

This manual is part of the SNiFF+ documentation set, which consists of:

- User's Guide
- Reference Guide
- C++ Tutorial
- C Tutorial
- Java Tutorial
- Fortran Tutorial
- Quick Reference Guide
- Release Notes, Installation Guide and Application Papers
- Online documentation of the above in HTML, PostScript and PDF formats

Conventions

One basic term

- **Symbol** — any programming language construct such as a class, method, etc.

Two conventions: menu references

For clarity and to avoid undue verbosity, the phrase:
"Choose the MenuCommand from the MenuName" is presented as follows:

- Choose **MenuName > MenuCommand**.

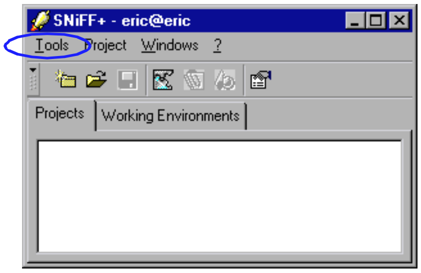
A context menu that appears when you click the right mouse button is referred to as:
Context menu, and consequently:
"Choose a menu command from the context menu that appears when you click the right mouse button" is presented as follows:

- Choose **Context menu > MenuCommand**

A note on Unix/Windows

The screenshots in this manual are all done on Windows NT. If you are working on Unix, what you see on your screen may look slightly different.

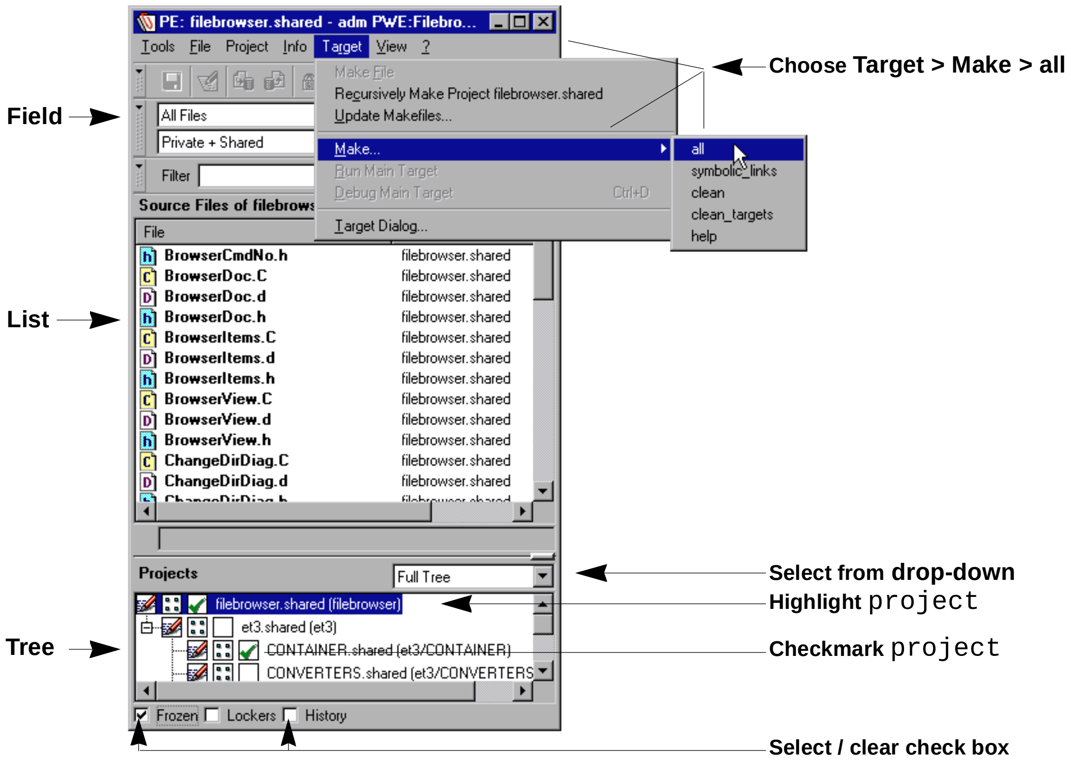
When you start SNiFF+, the first tool that appears is the Launch Pad. In this and other SNiFF+ tools, the first item in the menu bar is for launching tools.



- On **Windows,** it is called **Tools**.
- On **Unix**, it is depicted by an **Icon**.

  When we refer to this menu in order to launch a tool from the Launch Pad, or any other open SNiFF+ tool, we will use the notation: Choose **Tools > ToolName**.
- On Unix a “check box” looks like a “button” (Motif Look), and a “drop-down” looks like a “pop-up”.

Tool elements



BrowserCmdNo.h	filebrowser.shared
BrowserDoc.C	filebrowser.shared
BrowserDoc.d	filebrowser.shared
BrowserDoc.h	filebrowser.shared
BrowserItems.C	filebrowser.shared
BrowserItems.d	filebrowser.shared
BrowserItems.h	filebrowser.shared
BrowserView.C	filebrowser.shared
BrowserView.d	filebrowser.shared
BrowserView.h	filebrowser.shared
ChangeDirDiag.C	filebrowser.shared
ChangeDirDiag.d	filebrowser.shared

Typography

Capitalized Words	Names of tools, windows, dialogs and menus start with capital letters. Examples: Symbol Browser, **Tools** menu, File dialog.
Italics	Names of manuals and newly introduced terms are in italics. Examples: *User's Guide*, the *workspace* concept.
Boldface and ***Bold italics***	Menu, field and button names and menu entries are printed in boldface. Placeholders for symbols, selections or other strings in menus are in bold italics. Example: From the menu, choose **Show > Symbol(s) *selection*...**
`Monospace`	Code examples and symbol, file and directory names, as well as user entries are printed in monospace type. Examples: `.login`, `$PATH`, class `VObject`. Type `abc`.
`<Keys>`	Special keys are printed in monospace type with enclosing '`< >`'. Examples: `<CTRL>`, `<Return>`, `<Meta>`.

Feedback and useful links

Your feedback is always very welcome. Please send feedback to one of our support e-mail addresses.

Europe:

sniff-support@takefive.co.at

USA:

sniff-support@takefive.com

Useful links

SNiFF+ web pages:

- SNiFF+ Users Mailing List

  http://www.takefive.com/support/sniff-list.html
- SNiFF+ Users Mailing List Archive

  http://www.takefive.com/sniff-list
- Frequently Asked Questions

  http://www.takefive.com/support/faq.html
- Customer Newsletter

  http://www.takefive.com/news/customer_newsletter.html

Road Map 2

Introduction

This manual is a handbook for getting to know the SNiFF+ solution for Java source code engineering and is centered around three tutorials.

Each of the tutorials focuses on different SNiFF+ tools, tasks and concepts.

The Java example code used in all three tutorials is based on a multi-threaded client/server simulation, based on an idea by Kai-Uwe Maetzel at Ubilab, Union Bank of Switzerland, and adapted by TakeFive.

Technical Reference

The Technical Reference — page 91 summarizes Java-specific compilation, debugging, and other aspects of SNiFF+ for Java.

Please refer also to the *User's Guide* and/or the *Reference Guide* for more in-depth information relating to SNiFF+ in general.

What this manual is not

This manual is not an exhaustive guide to SNiFF+, nor will it teach you Java.

The SNiFF+J Java Tutorial

Note

Please note that a Log Window, displaying SNiFF+ error and control messages, may appear at several stages throughout this tutorial.

The SNiFF+ Java Tutorial consists of the following parts

I Project Setup

This part is for you if

- you want to quickly set up a single user project
- you want to add a browsing only project to an existing project

II Browsing

This part is for you if

- you are a new SNiFF+ user
- you want to quickly learn how to use SNiFF+ for browsing Java code

III Edit/Compile/Debug

This part is for you if

- you want to learn about compiling Java targets
- you want an introduction to the tools used in the Java edit/compile/debug cycle
- you need SNiFF+ and RCS (included in the SNiFF+ package) for configuration management and version control (CMVC)
- you are responsible for setting up and maintaining projects and working environments in a multi-user/multi-platform work situation (*Working Environments Administrator*)

  Note that this tutorial introduces concepts and tools used in developing, irrespective of whether you are working alone or as part of a team.

IV Technical Reference

This section deals only with the Java-specific aspects of SNiFF+. For all other aspects of SNiFF+, please see the *User's Guide* and the *Reference Guide*.

Part II
Project Setup

Setting up a Java Project 1

In this chapter we are setting up a Java project for a single user. Under Adding new team members — page 87, we will show you how to add new team members.

This chapter is about

- settings for SNiFF+ Java projects
- using the Project Setup Wizard to set up a SNiFF+ version controlled project.

Preparing the Environment

Before starting this tutorial —

- Make sure that your PATH environment variable points to

  `<your_jdk_installation_dir>/bin`

  so that SNiFF+ can find the JDK javac compiler.
- If you didn't select the "Other Packages" option (for Java) during the SNiFF+ installation process, start the SNiFF+ installation again and select only this option.
- **If** you are using **JDK version 1.1.x**, set the following system environment variable

  `CLASSPATH=<your_jdk_1.1x_dir>/lib/classes.zip`

  The **JDK 1.1.x** compiler needs this to find the JDK class library.

  On Windows NT

  Problems can be caused by a `CLASSPATH` environment variable that does not conform to the upper case conventions (e.g., an application may set an environment variable as `ClassPath`). If this is the case, set a single environment variable `CLASSPATH` to point to all the paths in an existing `CLASSPATH` as well as any variation such as `Class-Path`.

- **If** you are using **JDK version 1.2**, make sure you do **not** have a `CLASSPATH` system environment variable pointing to **older** JDK class libraries.

  You needn't set any `CLASSPATH` environment variable at all in order for the **JDK 1.2** compiler to find the JDK class library. However, system environment variables will override the compiler's defaults, so caution is advised.
- You can download the latest JDK from

  http://java.sun.com

Copying the example

Copy the directory, including subdirectories,

`<your_sniff_installation_dir>/example/java/sniff_java`

to somewhere, where you have write permissions. We will refer to the full path of this directory as `<sniff_java>` in the rest of this tutorial.
The `<sniff_java>` directory contains

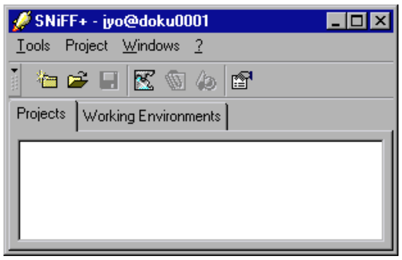
- two directories (`pwe_1` and `pwe_2`) which will hold your *Private Working Environments.*

  `pwe_1` already contains the `OfficeApp` directory with the example source code files (`<sniff_java>/pwe_1/OfficeApp`).
  `pwe_2` (at present empty) will be used later on for team work.
- a *Repository* (`<sniff_java>/repository`), which will be used for version control (at present empty).
- A configuration directory (`<sniff_java>/working_envs_config`), where SNiFF+ will maintain working environment configuration information (at present empty).

Setting your Preferences

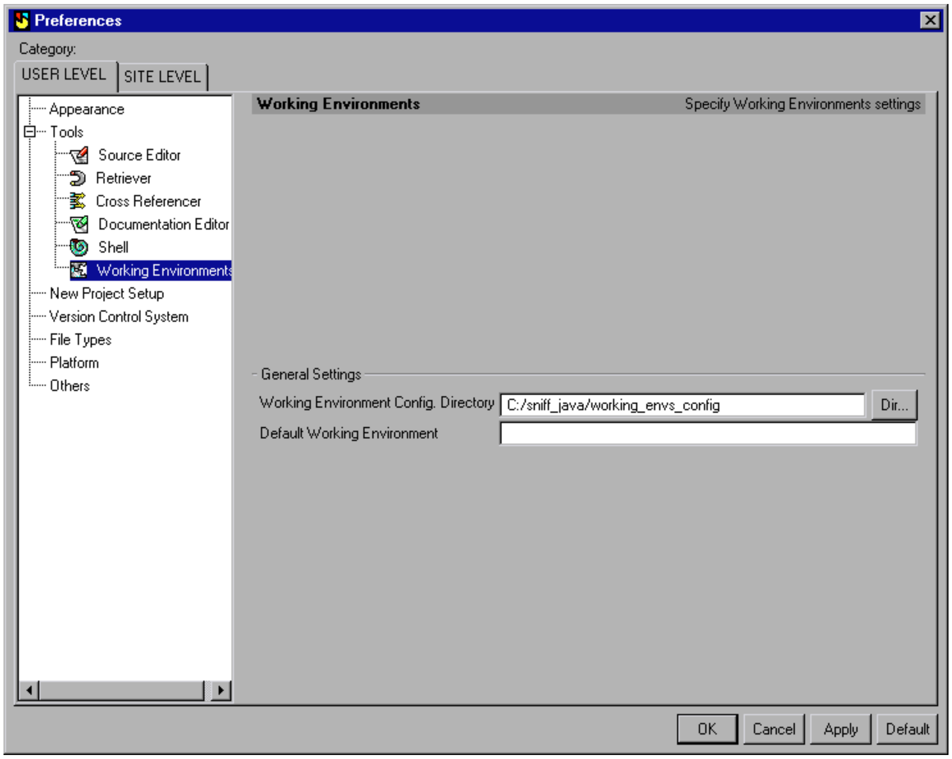
- Start SNiFF+.

  The Launch Pad appears.



- From the menu, choose **Tools > Preferences**, (on Unix, the **Tools** menu is represented by an icon).

In the Preferences — Working Environment Settings



Working Environment Config. Directory: C:/sniff_java/working_envs_config

Default Working Environment:

1. Under the Tools node, select Working Environments.
2. Press the **Dir...** button.
3. In the Directory dialog, navigate to `<sniff_java>/working_envs_config` and open it.
4. Press the **Select** button.
5. Press OK to close the Preferences and apply the setting.

   All Working Environment information will now be maintained in this directory by SNiFF+.

Project Setup

We assume you have completed all the necessary preparations described in the previous section of this chapter.

- In the Launch Pad, choose **Project > New Project... > with Wizard...** to start the Project Setup Wizard.

In the Project Setup Wizard

The Wizard starts by asking you to select how you intend to use the new SNiFF+ project.

- Accept the default selection, **Standard Setup,** and press **Next**.

  The “Select development task” page appears.

  In the remaining steps, we will refer to the names of Wizard pages. You can find a page’s name in the title bar of the Wizard.

In the “Select development task” page

- Select **Create a new SNiFF+ Project from scratch** and press **Next**.

In the “Your development organization” page

This tutorial starts off with a single-user development situation using RCS for configuration management and version control, we will later add a new team member to quickly make it a multi-user development situation so:

1. For the radio buttons, accept the defaults (**No/No**).
2. From the drop-down, choose **RCS**.

Note

RCS must be available or installed on your computer. If RCS isn’t already installed, install RCS from the SNiFF+ package.

3. Press **Next**.

In the “Select file types” page

- Select **Java** and press **Next**.

  HTML and **Visaj_Project** are automatically added when you select **Java**. Note that, after project setup, you can always add more file types. How to do so is described in the *User’s Guide*.

Note - Multi Language Projects

If you are using SNiFF+ Make Support, a directory can only contain Java files. It is not possible to build a project correctly if source files of other languages are located in the same directory. To avoid this problem, store these files in separate directories and create separate projects for them.

In the “Specify Repository” page

You are asked to specify your *Repository* (RWE) root directory. We will use the repository root directory, `<sniff_java>/repository`.

1. Press **Browse...**.
2. In the Directory dialog, navigate to `<sniff_java>/repository`, open it and press **Select**.
3. In the **RWE name** field, enter a name for the RWE, e.g., `repository`.
4. Press **Next**.

In the “Specify Private Working Environment” page

You are asked to specify your *Private Working Environment* (PWE) root directory. We will use the PWE_1 root directory, `<sniff_java>/pwe_1.`

1. Press **Browse**.
2. In the Directory dialog, navigate to `<sniff_java>/pwe_1`, open it and press **Select**.
3. In the **PWE name** field, enter a name for the PWE, e.g., `pwe_1`.

   Notice that your user name is entered next to the selected **Owner** check box. SNiFF+ needs your user name to correctly handle permissions. Being the owner of the PWE means that you are the only one who is allowed to modify its attributes.
4. Press **Next**.

In the “Create New SNiFF+ Project” page

SNiFF+ automatically enters the root of your Private Working Environment (PWE) in the **Project root directory** field.

1. Modify the entry in the **Project root directory** field to specify the root directory of the new project. This is:

   `<sniff_java>/pwe_1/OfficeApp`

   Notice that the new project’s name has changed to `OfficeApp`.
2. Accept the default project name.

> **Note**
>
> When you are running SNiFF+ in "personal mode", you must accept the default project name to be able to open the example project once you’ve added JDK as a subproject (we will do so later).

3. The `OfficeApp` directory, which does not itself contain any source code files, is the root directory where your source code packages start. Remember that, according to the Java language specification, the **class path to the package root** always ends one directory level **higher** than the highest-level directory containing named package code.

   So leave the **Source Package Root** field blank. When the project is generated, SNiFF+ will correctly use `OfficeApp` as the source code package root directory by default.
4. In the **Byte-Code Package Root** field, enter a name, e.g. `Classes`, for the generated byte-code root. When you compile, SNiFF+ will create a directory called `Classes` at the same level as the project root directory (`OfficeApp`). The byte-code will then be generated into the `Classes` directory (the javac compiler’s `-d` option).
5. By default, the **Create Subprojects** check box is selected, which is correct.
6. Select the **Use SNiFF+’s Makefiles** checkbox.
7. Press **Next**.

In the “Specify Java Make Attributes” page

You are asked to enter Classpath(s) to external packages. You needn’t enter anything here.

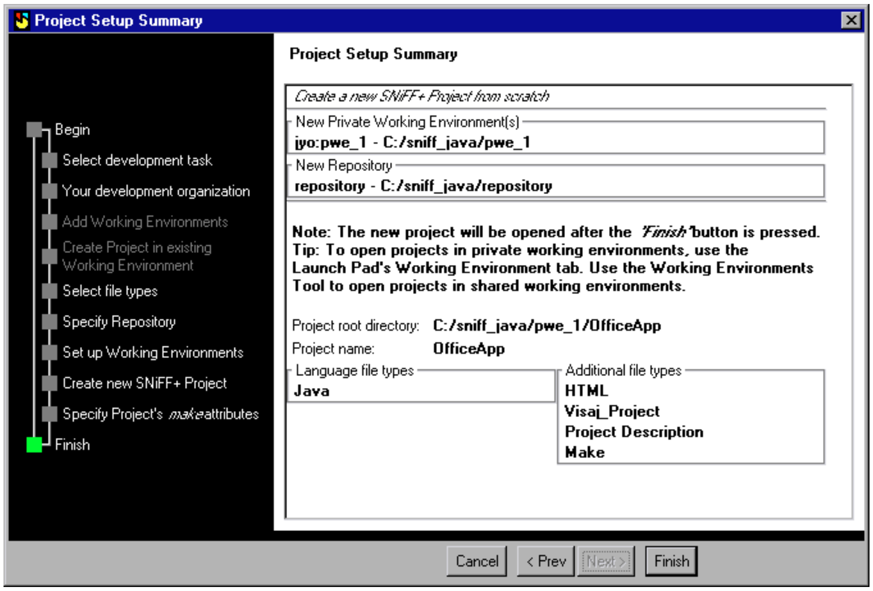
If you are using **JDK 1.1.x**, we assume you have already set a system environment variable for the JDK class library (under Preparing the Environment — page 11), so this need not be re-entered here. The SNiFF+ Classpath setting is generally used for libraries, where no source code is available. We do not use any such libraries in the example, so

- Press **Next**.

In the “Project Setup Summary” page

This page summarizes your specifications for the new SNiFF+ Java project and required Working Environments.

- Make sure that your Project Setup Summary page is similar the following. If it isn’t, please go back to the beginning of the Wizard and start again.


Project Setup Summary
Begin
Select development task
Your development organization
Add Working Environments
Create Project in existing Working Environment
Select file types
Specify Repository
Set up Working Environments
Create new SNiFF+ Project
Specify Project's make attributes
Finish
Cancel
< Prev
Next >
Finish

Project Setup Summary

Create a new SNiFF+ Project from scratch

New Private Working Environment(s)
jyo:pwe_1 - C:/sniff_java/pwe_1

New Repository
repository - C:/sniff_java/repository

Note: The new project will be opened after the *'Finish'* button is pressed.
Tip: To open projects in private working environments, use the Launch Pad's Working Environment tab. Use the Working Environments Tool to open projects in shared working environments.

Project root directory: C:/sniff_java/pwe_1/OfficeApp
Project name: OfficeApp

Language file types
Java

Additional file types
HTML
Visaj_Project
Project Description
Make

- Press **Finish**.

  SNiFF+ will now create the new `OfficeApp` project and all its subprojects.

- In the dialog that appears asking if you want to generate cross reference information, press **No**.

  Cross Reference information will be automatically generated when you open the Cross Referencer later on.
  When SNiFF+ is finished, the new project is opened in the Project Editor.

Review

In this chapter you

- made the necessary preparations for your environment
- set the directory where SNiFF+ maintains working environments information
- set up a version controlled single-user project using the Project Setup Wizard

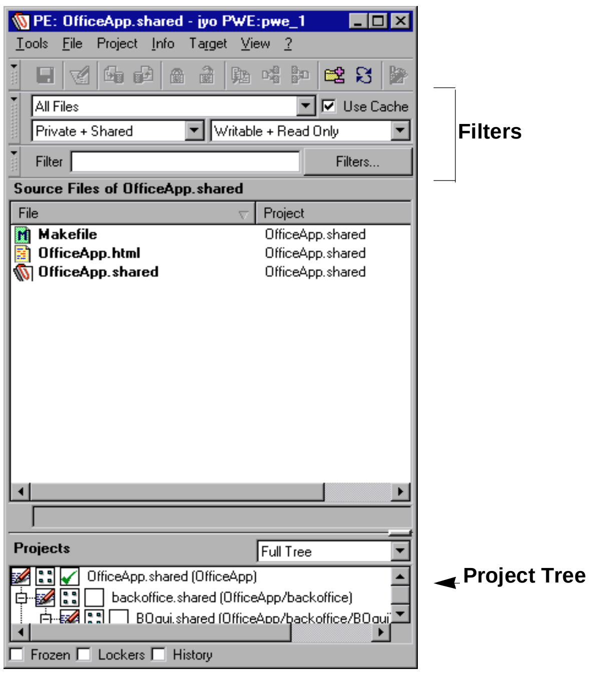
The Project Editor 2

This chapter is about

- project filtering in the Project Editor
- adding a subproject (the JDK API classes) to an existing project (`Office-App.shared`) so that you can properly follow all the references in your source code

Opening the Project Editor

- The Project Editor is opened automatically when you create a new project.
- To open the Project Editor from any tool, you would choose **Tools > Project Editor**; make sure the project you want to open is highlighted in the Launch Pad.



PE: OfficeApp.shared - iyo PWE:pwe_1

All Files ☑ Use Cache

Private + Shared | Writable + Read Only

Filter

Source Files of OfficeApp.shared

File	Project
Makefile	OfficeApp.shared
OfficeApp.html	OfficeApp.shared
OfficeApp.shared	OfficeApp.shared

Projects Full Tree

OfficeApp.shared (OfficeApp)
backoffice.shared (OfficeApp/backoffice)
BOgui.shared (OfficeApp/backoffice/BOgui

☐ Frozen ☐ Lockers ☐ History

The Project Tree

Checkmarking Projects

In the Project Tree, click into the checkboxes (left of the project names) to show/hide files in the File List. In SNiFF+, a project that has a checkmark in its checkbox is called a *check-marked project*.

1. In the Project Tree, click into the checkbox next to `backoffice.shared` and notice what happens in the File List.

   All the files in the `backoffice.shared` project are now shown in the File List.

2. Click into the checkbox next to `backoffice.shared` again to clear it.

   The files in this project are no longer shown.

Selecting from a tree of projects

Very often, when the project structure gets more complex and contains many subprojects, you will want to view and manipulate a tree of projects like a single project.

1. Click on the node of `backoffice.shared` to collapse it.
2. Try alternately checkmarking and clearing the checkbox next to the collapsed node.

   When the project is checkmarked, all the files in `backoffice.shared` and its tree of subprojects are listed. Conversely, when the project is not checkmarked, neither its own files, nor any of those in its subprojects, are shown.

Creating a project for the JDK sources

To properly follow references into the JDK sources, you need to create a project for the JDK API sources (in `<your_jdk_directory>/src/` directory) and then add it to the existing project as a subproject.

Note

You will need at least JDK 1.1.x for the example project. If you don't have an up-to-date JDK, you can download one from:

http://java.sun.com.

A project for the JDK API sources is, like other libraries, a typical browsing-only situation.

In the Launch Pad

- To set up a SNiFF+ project for the JDK sources, choose the menu command **Project > New Project... > with Wizard...**

In the Project Setup Wizard

The Wizard starts by asking you to select how you intend to use the new SNiFF+ Project.

- Select **Browsing-only Setup**, and press **Next**.

  The “Select file types” page appears.

In the “Select file types” page

- Select **Java** and press **Next**.

In the “Specify project location and name” page

1. Press the **Browse** button next to the **Source code root directory** field and navigate to `<your_jdk_directory>/src/`
2. Open the `<your_jdk_directory>/src` directory and press **Select**.
3. Enter `jdk_src_java` as a name for the project in the **Project Name** field.

   In this tutorial, we will use `jdk_src_java` to refer to the project.
4. Press **Next**.

   If you do not have write permissions for the source code root directory, the **Specify writable location for SNiFF+ Generated Files** page appears. In the **Generate Directory root** field, press **Browse...** and specify a root directory for which you have write permission.

In the “Specify Java Make Attributes” page

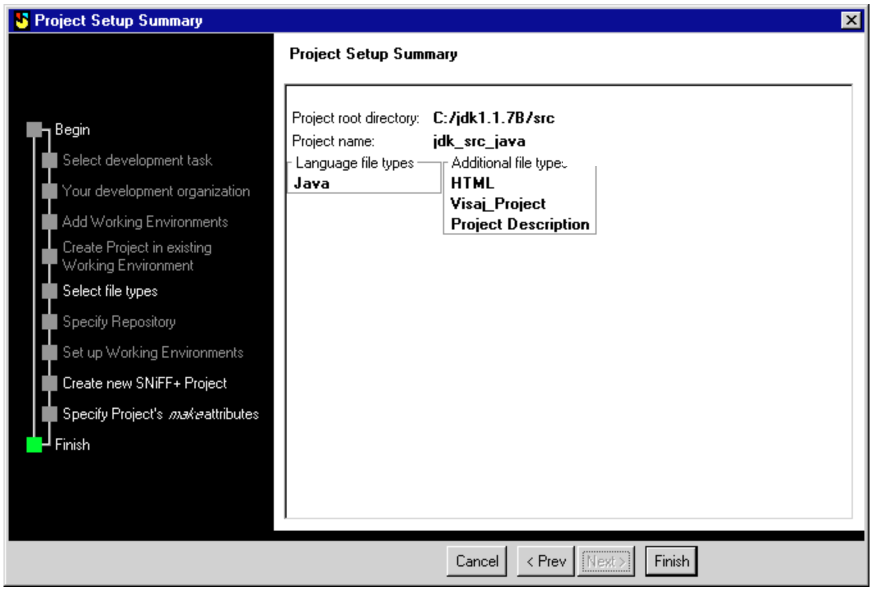
Leave this field blank.

- Press **Next**.

In the “Project Setup Summary” page

This page summarizes your specifications for the new SNiFF+ Java Project.

1. Make sure that your Project Setup Summary page is similar to the following. If it isn’t, please go back to the beginning of the Wizard and start again.


Project Setup Summary
Begin
Select development task
Your development organization
Add Working Environments
Create Project in existing Working Environment
Select file types
Specify Repository
Set up Working Environments
Create new SNiFF+ Project
Specify Project's make attributes
Finish
Cancel
< Prev
Next >
Finish

Project Setup Summary

Project root directory: C:/jdk1.1.7B/src
Project name: jdk_src_java

Language file types	Additional file types
Java	HTML
	Visaj_Project
	Project Description

2. Press **Finish**.

If you are running Personal SNiFF+

If you are not running Personal SNiFF+, ignore the step below and continue with If you have a SNiFF+ License other than one for Personal SNiFF+ — page 23.

- A dialog appears warning you that you can load a maximum of 200 files. The example projects are an exception so you can safely ignore this warning and press **Ok**. This project has been created and then closed. You will add the `jdk_src_java` project to the `OfficeApp` project later on.

If you have a SNiFF+ License other than one for Personal SNiFF+

If you are running Personal SNiFF+, continue with Adding the JDK projects to your project — page 23.

1. In the dialog that appears asking if you want to generate cross reference information, press **No**.

   Cross Reference information will be automatically generated when you open the Cross Referencer later on.

   When the SNiFF+ Project Setup Wizard has finished creating the project for you, a Project Editor is automatically opened, and the project structure of the new project is displayed.

2. You won't need the jdk_src_java project open in SNiFF+ so in the Launch Pad, select the `jdk_src_java.proj` and choose **Project > Close Project jdk_src_java.proj** to close the project.

   You will add the `jdk_src_java` project to the `OfficeApp` project later on.

Adding the JDK projects to your project

In the Project Editor

To add the `jdk_src_java` project to your `OfficeApp.shared`:

1. In the Project Tree, highlight `OfficeApp.shared` by clicking on its name.
2. Choose **Project > Add Subproject to OfficeApp.shared**.
3. In the Subproject File dialog that appears, navigate to

   `<your_jdk_1.1.x_directory>/src`

   or to the root directory where you've stored your project description file.
4. Select `jdk_src_java.proj` and press **Open**.
5. In the dialog that appears asking you if you want to synchronize the "Byte-Code Package Root Directory", select the **Repeat** checkbox and press **No**.

   The Project you set up with the Wizard was for browsing only, and no Makefiles were generated. Hence when you compile, the projects you are now adding will not be compiled. Also, even if this were the case, you would only choose this option if you are sure that you will not be adding the source code library projects to any other projects.

   The `jdk_src_java.proj` will now be added as a subproject of `OfficeApp.shared`. You can verify that this has been done by taking a look at the Project Tree. Notice also that the icon next to `OfficeApp.shared` has changed to warn you that the project has been modified and not yet saved.

- To save the project, choose **Project > Save OfficeApp.shared**.

Saving a Project Tree view

Although you added the JDK API sources so as to properly follow references, you will generally be more interested in your own projects than in libraries. Rather than always resetting the Project Tree, you can save a view of the Project Tree to reuse later.

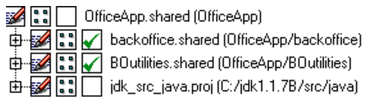
Preparing the Project Tree

- Collapse the nodes of all projects and checkmark only

  `backoffice.shared`

  `BOutilities.shared.`

  The root project, `OfficeApp.shared`, doesn't contain any source files, so clear the checkbox next to it.
  Your Project Tree should now look like this:



```
OfficeApp.shared (OfficeApp)
backoffice.shared (OfficeApp/backoffice)
BOutilities.shared (OfficeApp/BOutilities)
jdk_src_java.proj (C:/jdk1.1.7B/src/java)
```

Saving the Project Set

1. From the menu, choose **View > Save Project Set**.
2. In the dialog that appears, enter a name for the view of the Project Tree as it appears now, e.g., `MySources` and press **Ok**.

   We will use this name to refer to the project set when we next use it.
   Note that you can save and reuse project sets using the **View** menu in any tool that has a Project Tree.

Review

In this chapter you

- worked in the Project Tree
- created a browsing-only project for a source code library
- added a subproject to an existing project
- saved a view of a project set

Checking in files 3

Checking in project files for the first time is the first step in version-controlling your SNiFF+ projects. Once files have been checked in, you can see the history and version tree of selected files.

This chapter is about:

- checking in files into the repository

Checking in the project files

To check the project in, complete the following steps.

In the Project Editor

Remember that, in the last section Saving a Project Tree view — page 24, you saved a project set which we called MySources. Use this set again and since we will be version controlling OfficeApp.shared:

- in the Project Tree, checkmark OfficeApp.shared as well.

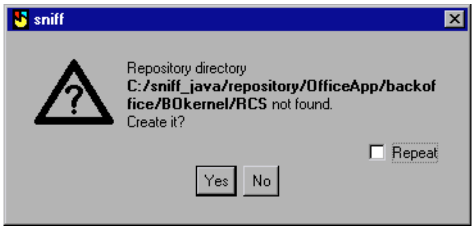
We won't checkmark the `jdk_src_java.proj` project because there is no need to version control this project since we won't be making any changes to it.

In the Project Editor

We will now check in all the projects' files to the Repository.

1. From the menu, choose **File > Select All**.
2. From the menu, choose **File > Check In...**.

   SNiFF+J informs you that it cannot find the directories of the project in the RWE root directory (they haven't been created yet). You will now have SNiFF+ initialize your RWE by copying the PWE project directory structure into the RWE. This dialog will reappear for each new Repository directory, unless you enable **Repeat**.


sniff

Repository directory
C:/sniff_java/repository/OfficeApp/backoffice/BOkernel/RCS not found.
Create it?

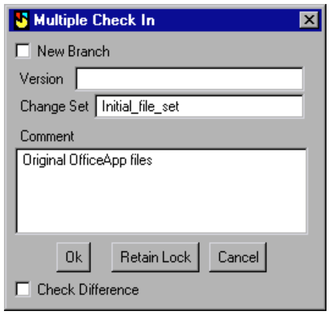
3. Select the **Repeat** check box and press **Yes** to create the necessary Repository directories for the project.

   When SNiFF+ has finished initializing your RWE, the Check In dialog appears.

In the Multiple Check In dialog

You can use this dialog to check in versions of single or multiple files. When you have made changes to multiple files, you can check in all the files at the same time and associate them with a *change set*. By doing so, you can perform a variety of version-control operations on all the files in a change set at the same time.
At this point, although we haven't made any changes, we will make use of the **Change Set** field to reflect the fact that we are checking in the initial versions of all the files in the project.

1. Leave the **Version** field blank. SNiFF+ will automatically assign a version number (1.1) and later increment it automatically.
2. In the **Change Set** field, enter a description, e.g., `Initial_file_set`.
3. In the **Comment** field, enter a descriptive text, e.g., `Original OfficeApp files`.
4. Press **Ok**.



Initial_file_set
Original OfficeApp files

In the Project Editor

When the check in process is over, take a look at your Project Editor. The following will have changed:

- The files in the File List are no longer in bold typeface. This means they are now read-only.
- The icons in the Project Tree have also changed to indicate that the projects, too, are read-only.

  To get an overview of what icons and typefaces signalize in a particular tool, choose **Help > Quick Ref**.

Review

In this chapter you checked in all the files into the repository.
The next part of this tutorial introduces you to the SNiFF+ browsing tools.

Part III
Browsing

Browsing Symbols 1

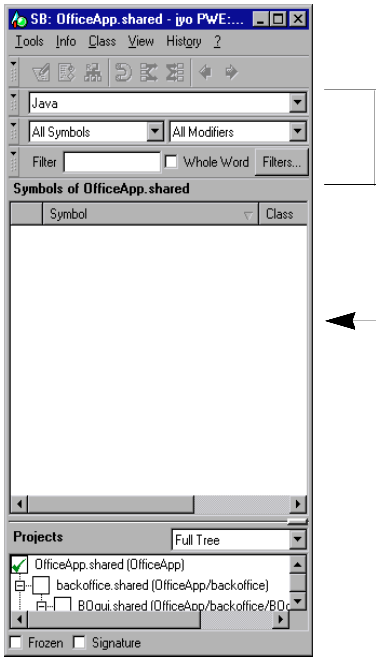
In SNiFF+, a *symbol* is any programming language construct such as a class, method, field, etc.

This chapter is about

- further user interface features which are common to many SNiFF+ tools
- using the Symbol Browser to filter symbol information

Opening the Symbol Browser

- To open the Symbol Browser from any tool, choose **Tools > Symbol Browser**.


SB: OfficeApp.shared - jyo PWE:...
Tools Info Class View History ?
Java
All Symbols
All Modifiers
Filter
Whole Word
Filters...
Symbols of OfficeApp.shared
Symbol
Class
Projects
Full Tree
OfficeApp.shared (OfficeApp)
backoffice.shared (OfficeApp/backoffice)
BOgui.shared (OfficeApp/backoffice/BO
Frozen
Signature

Filters

Symbol List (now blank): The symbols that are shown depend on the settings in the filters and the Project Tree

The Symbol List is empty now because there are no symbols in the default selection in the Project Tree. You will now selectively populate the Symbol List with symbols from your source code.

The Symbol List

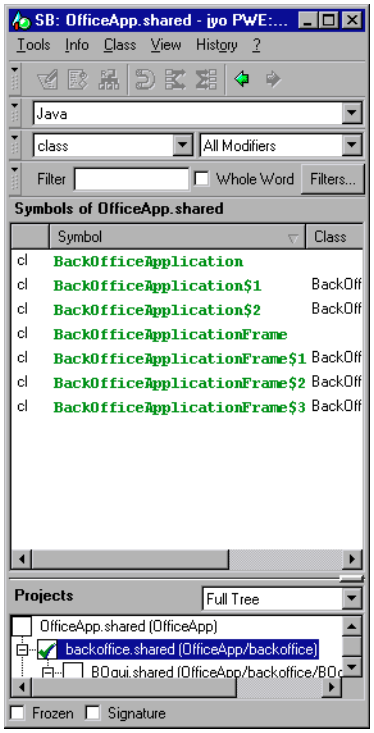
To look at only those symbols defined in a given project, e.g. `backoffice.shared`

1. In the Project Tree, highlight `backoffice.shared` by clicking on its name.
2. Right-click anywhere in the Project Tree, and choose

   Context menu > Select From backoffice.shared Only.
3. From the Symbols drop-down in the tool (the Symbols filter), select **class** to see only the classes in `backoffice.shared.`

   As you can see, there are seven classes defined in `backoffice.shared`. Five of these are anonymous classes.

   Naming of anonymous classes follows the SUN convention: the anonymous class name consists of the outer class name, followed by the `'$'` character and a consecutive number for each new anonymous class within the outer class.


SB: OfficeApp.shared - iyo PWE:...
Tools Info Class View History ?
Filters...

Java

class | All Modifiers

Filter [ ] Whole Word

Symbols of OfficeApp.shared

	Symbol	Class
cl	BackOfficeApplication	
cl	BackOfficeApplication$1	BackOff
cl	BackOfficeApplication$2	BackOff
cl	BackOfficeApplicationFrame	
cl	BackOfficeApplicationFrame$1	BackOff
cl	BackOfficeApplicationFrame$2	BackOff
cl	BackOfficeApplicationFrame$3	BackOff

Projects: Full Tree

- [ ] OfficeApp.shared (OfficeApp)
- [✓] backoffice.shared (OfficeApp/backoffice)
  - [ ] BOgui.shared (OfficeApp/backoffice/BO

[ ] Frozen [ ] Signature

The Signature check box

You can find out more about these symbols by selecting the **Signature** check box at the bottom of the tool.

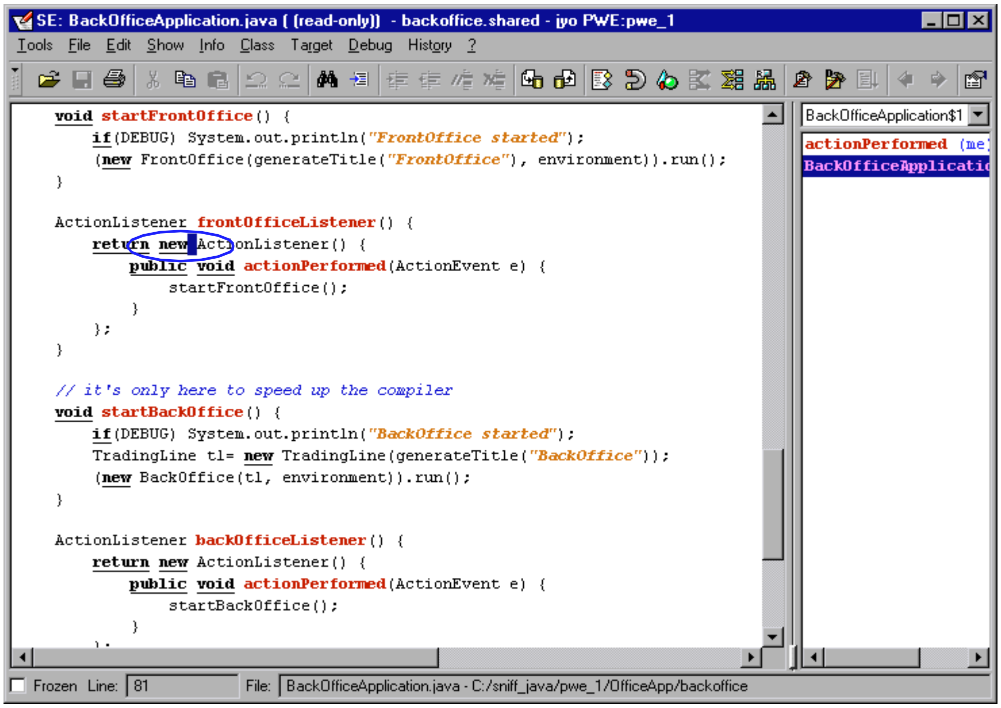
- Select the **Signature** check box at the bottom of the tool, and then make sure you see all the columns in the Symbol List.
- Clear the **Signature** check box to get a less cluttered view.

Looking at an anonymous class

To take a look at the anonymous class `BackOfficeApplication$1`:

- In the Symbol List, double-click on `BackOfficeApplication$1`.

  A Source Editor is now opened, the source file `BackOfficeApplication.java` is loaded and the cursor is positioned to the line where the symbol is defined.
  Here, because the class is anonymous, the space immediately after `new` is highlighted. This is how SNiFF+ shows anonymous classes in the Source Editor.


SE: BackOfficeApplication.java ((read-only)) - backoffice.shared - jyo PWE:pwe_1
Tools File Edit Show Info Class Target Debug History ?
BackOfficeApplication$1
actionPerformed (me
BackOfficeApplicati
Frozen Line: 81
File: BackOfficeApplication.java - C:/sniff_java/pwe_1/OfficeApp/backoffice

```
    void startFrontOffice() {
        if(DEBUG) System.out.println("FrontOffice started");
        (new FrontOffice(generateTitle("FrontOffice"), environment)).run();
    }

    ActionListener frontOfficeListener() {
        return new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                startFrontOffice();
            }
        };
    }

    // it's only here to speed up the compiler
    void startBackOffice() {
        if(DEBUG) System.out.println("BackOffice started");
        TradingLine tl= new TradingLine(generateTitle("BackOffice"));
        (new BackOffice(tl, environment)).run();
    }

    ActionListener backOfficeListener() {
        return new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                startBackOffice();
            }
```

- To avoid cluttering your screen, close the Source Editor.

Using filters

Besides classes, you can filter for all other Java types and also modifiers. Because the example project contains only Java source files, only Java symbol types can be filtered, and only **Java** can be selected in the **Language** drop-down.

You can filter for any combination of Java symbol types and modifiers.

1. In the Project Tree, choose **Context menu > Select From All Projects**.
2. Press the **Filters...** button.
3. In the Filters dialog, select different items on the various tabs and press **Apply**.

   Your selections are applied and the Filter dialog remains open, ready for new selections. You might want to play around with different selections. Go ahead.
4. To close the Filter dialog, press **Ok**.

   Your selections in the dialog are applied and the dialog is closed.

   Notice that, if multiple filters were selected when you closed the dialog, the selected entry in the corresponding drop-downs has changed to **Filtered...**. You can now either choose other list entries, or the **Filtered...** entry itself. If you do so, the Filters dialog opens again and you can check to see the combination selected.

Keyboard navigation in lists

In each list of any SNiFF+ tool, you can quickly navigate to entries by clicking into the list, then typing the name of the entry you wish to find. Each consecutive keystroke immediately causes the list to position to the next matched entry.

Situation: You know the name of a method, but you don’t know where it is implemented. For example, you will need to know where `main` is implemented in order to compile later on.

The quickest way to get the information you need is:

1. First, restrict the search to the actual source code projects. Remember you saved a view to these projects under Saving a Project Tree view — page 24.

   So, choose **View > Select Project Set > MySources**
2. In the Filter drop-downs, make sure **All Symbols** and **All Modifiers** are selected.
3. Click into the Symbol List.
4. Press the `<m>` key.

   The list is positioned to the first entry that starts with `’m’`, which is already the entry you need.
5. For more information, select the **Signature** check box at the bottom of the tool.

   The main method is implemented in the class `BackofficeApplication`.

   In the next chapter we’ll use this class as a starting point for a structured “top down” overview of the classes in whole project structure. So leave the Symbol Browser open and make sure that `main` in `BackofficeApplication` is selected.

Review

In this chapter you

- worked with filters and the Filters dialog, which is available also in other SNiFF+ tools
- learned about keyboard navigation in lists. This useful feature is available throughout SNiFF+.
- found the file and project location of symbol
- used a previously saved view of a project set

Understanding Class Hierarchies 2

To understand the interrelationships between symbols, you need to add a new dimension to the “flat” view the Symbol Browser provides. This new dimension is provided by the Hierarchy Browser.

This chapter is about

- using the Hierarchy Browser to
- get a hierarchically structured “top down” view of interface and class inheritances in the Java software project

Opening the Hierarchy Browser

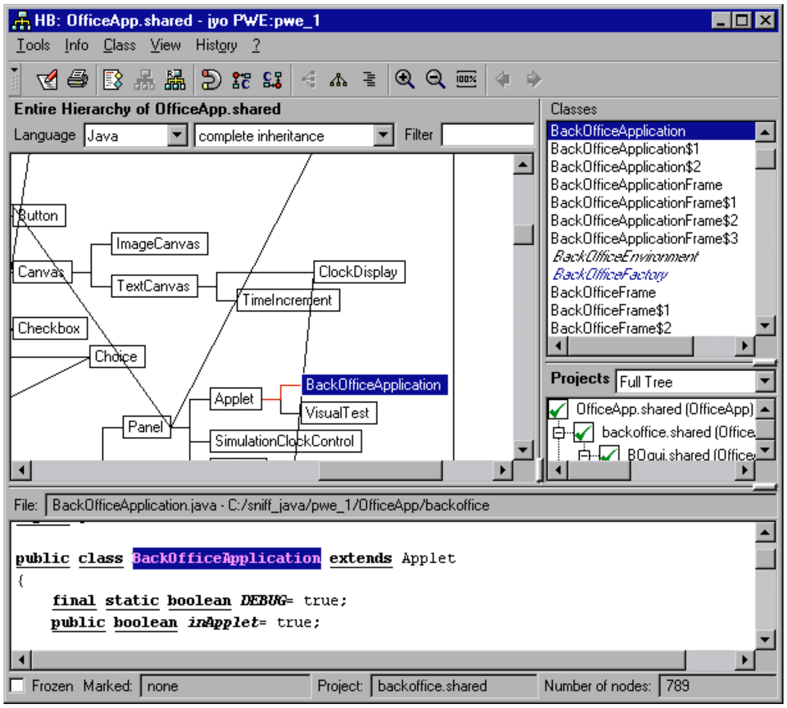
- Make sure `main` in `BackofficeApplication` is still selected in the Symbol Browser, then right-click anywhere in the Symbol List and choose

  Context menu > Show BackofficeApplication in Entire Hierarchy.

  The Hierarchy Browser opens.
- You might want to close the Symbol Browser to get a less cluttered view.

You can open the Hierarchy Browser from any tool where you can select a class by clicking on the class you are interested in, and then choosing the menu commands
Class > Show className in Entire Hierarchy, or **Class > Show className Relatives.**
You can also open the Hierarchy Browser, like any other SNiFF+ tool, using the **Tools** menu.

The Hierarchy Browser opens to show the class BackOfficeApplication in the Entire Hierarchy.


HB: OfficeApp.shared - iyo PWE:pwe_1
Tools Info Class View History ?
Entire Hierarchy of OfficeApp.shared
Language Java
complete inheritance
Filter
Button
ImageCanvas
Canvas
TextCanvas
ClockDisplay
TimeIncrement
Checkbox
Choice
BackOfficeApplication
Applet
VisualTest
Panel
SimulationClockControl
Classes
BackOfficeApplication
BackOfficeApplication$1
BackOfficeApplication$2
BackOfficeApplicationFrame
BackOfficeApplicationFrame$1
BackOfficeApplicationFrame$2
BackOfficeApplicationFrame$3
BackOfficeEnvironment
BackOfficeFactory
BackOfficeFrame
BackOfficeFrame$1
BackOfficeFrame$2
Projects Full Tree
OfficeApp.shared (OfficeApp)
backoffice.shared (Office
BOgui.shared (Office
File: BackOfficeApplication.java - C:/sniff_java/pwe_1/OfficeApp/backoffice
Frozen Marked: none Project: backoffice.shared Number of nodes: 789

```
public class BackOfficeApplication extends Applet
{
    final static boolean DEBUG= true;
    public boolean inApplet= true;
```

Colors, typeface and frames

Different colors, typefaces and frames around the symbol names provide visual feedback about the kind of class you are looking at. For a description of these elements, choose

Help(?) > Quick Ref

Many SNiFF+ tools use different typefaces, icons etc. to provide different kinds of visual feedback. Choosing the **Help(?) > Quick Ref** menu command opens online documentation at the section describing these features.

Class inheritance

The Hierarchy Browser opens with all projects in the Project Tree checkmarked. You have a graphical representation of all the inheritance relationships among all the classes and all the interfaces, including the JDK classes. In other words, far too much information — scroll around and take a look.

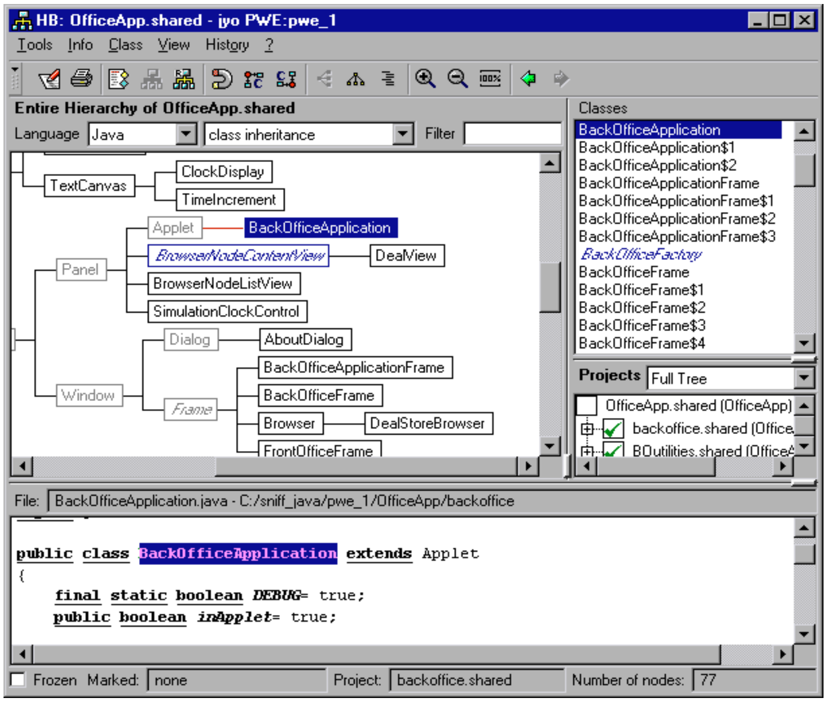
Remember that, in the section Saving a Project Tree view — page 24, you saved a project set which we called `MySources`. You can now use this set again.

1. From the menu, choose **View > Select Project Set > MySources**

   The classes and interfaces of all the projects in the Project Tree, except for those in `jdk_src_java.proj` that are not used in the example project, are displayed. SNiFF+ traces only those classes and interfaces needed to display a complete tree (grayed); all others are hidden.

2. Now, let's take a look at the classes used in the project (without interfaces).

   From the **Inheritance** drop-down (above the Hierarchy view), choose **class inheritance**. You now have an overview of the class hierarchy in your project structure.



Language: Java
class inheritance
Filter:
Projects: Full Tree

File: BackOfficeApplication.java - C:/sniff_java/pwe_1/OfficeApp/backoffice

```
public class BackOfficeApplication extends Applet
{
    final static boolean DEBUG= true;
    public boolean inApplet= true;
```

Frozen Marked: none Project: backoffice.shared Number of nodes: 77

Review

In this chapter you

- moved from a flat view of the symbols to a hierarchically structured view of the classes and interfaces.
- used a previously saved view of a project set.
- got an overview of the class hierarchy in the example project.

Browsing class members 3

With the Class Browser you can find out about

- member methods and fields of a class
- visibility of members
- modifiers (explicit and implicit)
- overloaded, overriding and overridden methods

Opening the Class Browser

You should still have the Hierarchy Browser open. After looking at the overall class hierarchy, let's now browse the members (the internal structure) of an individual class of a project. We will use the class named `Browser` as a test case. To open the Class Browser with the member information for the class named `Browser`:

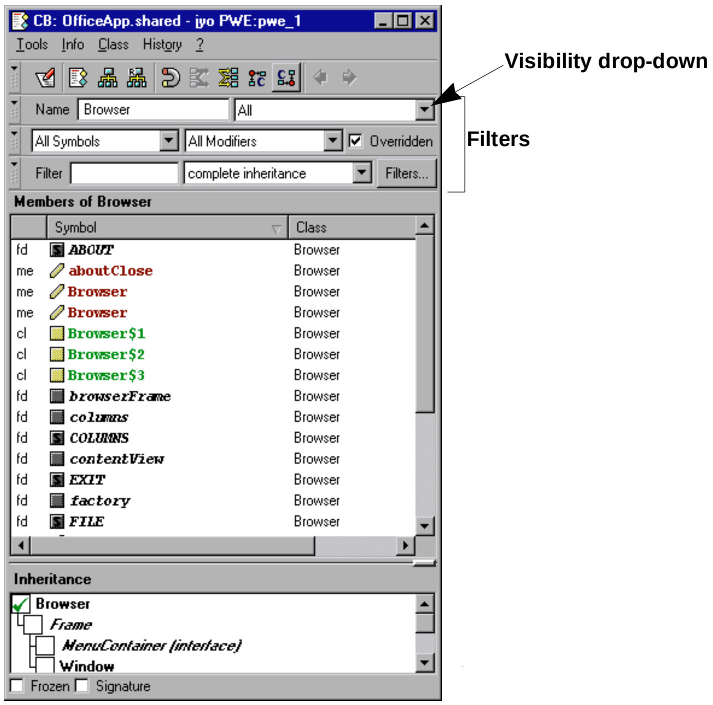
1. Click into the Class List and type `br`

   The focus is set to `Browser`.

2. Choose **Context menu > Browse Browser**.

   Note that you can open the Class Browser from any tool where you can select a class by clicking on the class and then choosing the **Browse** *ClassName* command either from the right-click **Context menu** or from the **Class** menu.

The Class Browser opens to show you the members of `Browser`.



CB: OfficeApp.shared - jyo PWE:pwe_1

Name: Browser — All

All Symbols — All Modifiers — [✓ Overridden]

Filter: — complete inheritance — Filters...

Members of Browser

	Symbol	Class
fd	ABOUT	Browser
me	aboutClose	Browser
me	Browser	Browser
me	Browser	Browser
cl	Browser$1	Browser
cl	Browser$2	Browser
cl	Browser$3	Browser
fd	browserFrame	Browser
fd	columns	Browser
fd	COLUMNS	Browser
fd	contentView	Browser
fd	EXIT	Browser
fd	factory	Browser
fd	FILE	Browser

Inheritance

- [✓] Browser
  - Frame
    - MenuContainer (interface)
    - Window

Frozen Signature

The Class Browser is very similar to the Symbol Browser in layout. But, instead of a Project Tree, there is an Inheritance Tree at the bottom of the tool. The essential handling concepts of Lists, Trees and Filters, however, stay the same in all the tools.

The Member List in the Class Browser also has (often a combination of) icons; the color tells you about member visibility:

- public — yellow
- protected — blue
- private — dark gray

For a description of the icons, choose **Help(?) > Quick Ref**

Filtering

As in the Symbol Browser and the Project Editor, there are also a number of Filter drop-downs. Pressing the **Filters...** button opens a Filters dialog where you can choose any combination of filters.

Visibility

When the tool opens, you see all the members of `Browser`, this is because the default selection in the **Visibility drop-down** (above the **Overridden** check box) is **All**.
You might be interested in all symbols that `Browser` has access to:

1. Right-click in the Inheritance Tree at the bottom of the tool, and choose

   Context menu > Select From All Classes.
   The Member List shows a list of all the members of all the classes in the inheritance tree as seen by `Browser`. That is, all the members of `Browser` itself, as well as the public and protected members of `Browser`'s base classes. Private members in base classes are grayed because these are not visible from `Browser`. To hide these completely:
2. Press the **Filters...** button.
3. In the Filters dialog, select the Visibility tab.
4. In the Visibility tab, clear the **invisible private** check box.
5. Press **Apply**.

   You now see only those symbols that are accessible from `Browser`.

Overloading

You might be interested in all overloaded methods in the hierarchy.

1. In the Modifiers drop-down, which is set to **All Modifiers** by default, select **overloaded**.

   Assuming that all the classes are still checkmarked, you see all overloaded methods in the Members List of `Browser`.
2. Enable the **Signature** check box in the status line.

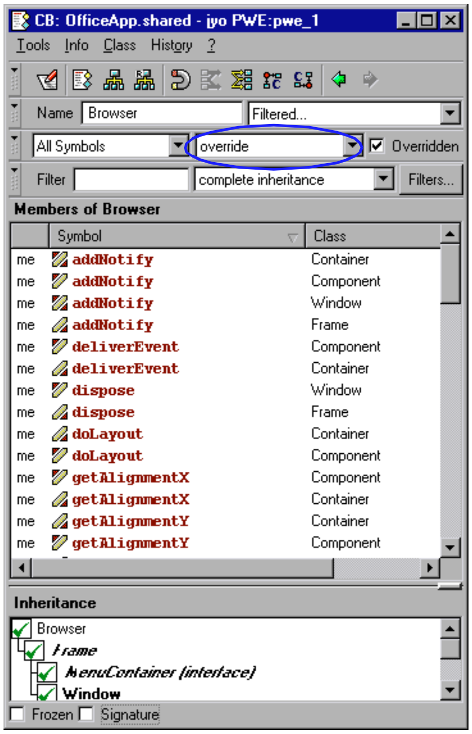
   The signature of each member is also listed, and you can see the parameter lists.

Overriding

You may need to know which methods override and/or are overridden by other methods. For this kind of information:

1. From the **Modifier** drop-down, choose **override**.
2. Make sure the **Overridden check box** is selected.
3. Clear the **Signature** check box in the status line.

   You now see all the members that override methods defined in a base class, as well as all the methods that are overridden in derived classes.
   Look at the `addNotify` method at the top of the Member List, you can see that it is overridden 3 times, giving a total of 4 implementations.



Name: Browser — Filtered...
All Symbols — override — [✓] Overridden
Filter: — complete inheritance

Members of Browser

	Symbol	Class
me	addNotify	Container
me	addNotify	Component
me	addNotify	Window
me	addNotify	Frame
me	deliverEvent	Component
me	deliverEvent	Container
me	dispose	Window
me	dispose	Frame
me	doLayout	Container
me	doLayout	Component
me	getAlignmentX	Component
me	getAlignmentX	Container
me	getAlignmentY	Container
me	getAlignmentY	Component

Inheritance
✓ Browser
✓ Frame
✓ MenuContainer (interface)
✓ Window

☐ Frozen ☐ Signature

Marking classes where methods are overridden

You can see by the icon in the Class Browser's Member List that the initial implementation of `addNotify` is in `Component`, so this is your starting point.
To visualize the relationships:

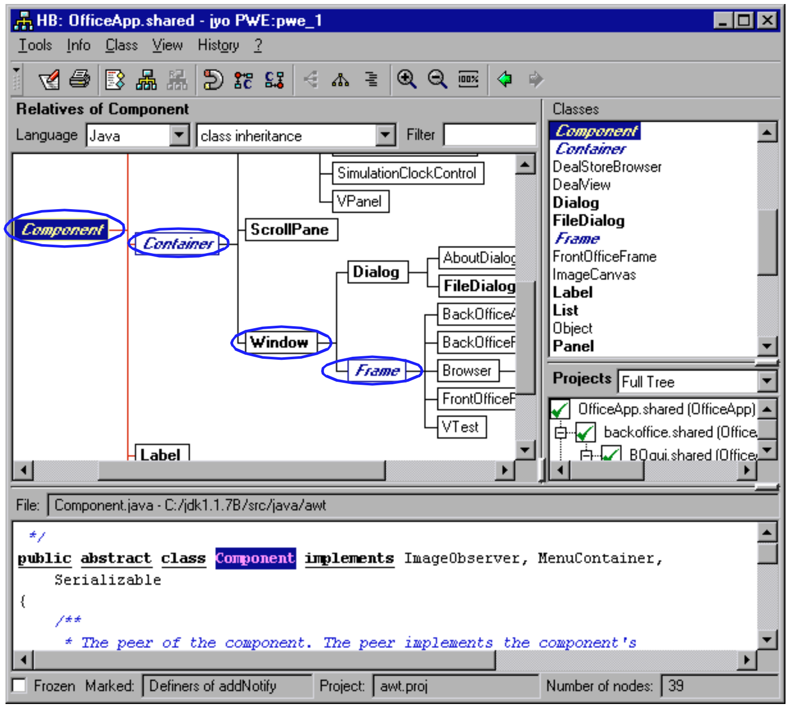
1. In the Class Browser's Member List, highlight the `addNotify` method implemented in the class `Component`.
2. Choose **Class > Mark Relatives Defining addNotify**.

   The Hierarchy Browser opens and is focussed on `Component`.

In the Hierarchy Browser

- From the **Inheritance** drop-down, choose **class inheritance** to see only class inheritance.

  The classes implementing `addNotify` are indicated in **bold** typeface. We have ringed the classes where `addNotify` is defined/overridden in `Browser`'s chain of inheritance (you know this from the Class Browser).
  You can also see clearly which classes override which implementation of `addNotify`.



Language: Java — class inheritance — Filter:

Projects: Full Tree

File: Component.java - C:/jdk1.1.7B/src/java/awt

```
 */
public abstract class Component implements ImageObserver, MenuContainer,
    Serializable
{
    /**
     * The peer of the component. The peer implements the component's
```

Frozen Marked: Definers of addNotify Project: awt.proj Number of nodes: 39

Review

In this chapter you

- used visibility filters.
- looked at overloaded methods.
- visualized overriding/overridden methods in a class hierarchy.

4 Code Dependencies and Impact Analysis

This chapter is about using the Cross Referencer to find out about:

- symbol types used as components of a given symbol
- all the symbols that a given symbol refers to (“Refers-To”)
- all the symbols that refer to a given symbol (“Referred-By”)

Opening the Cross Referencer

You should still have the Hierarchy Browser and the Class Browser open from the previous chapter.

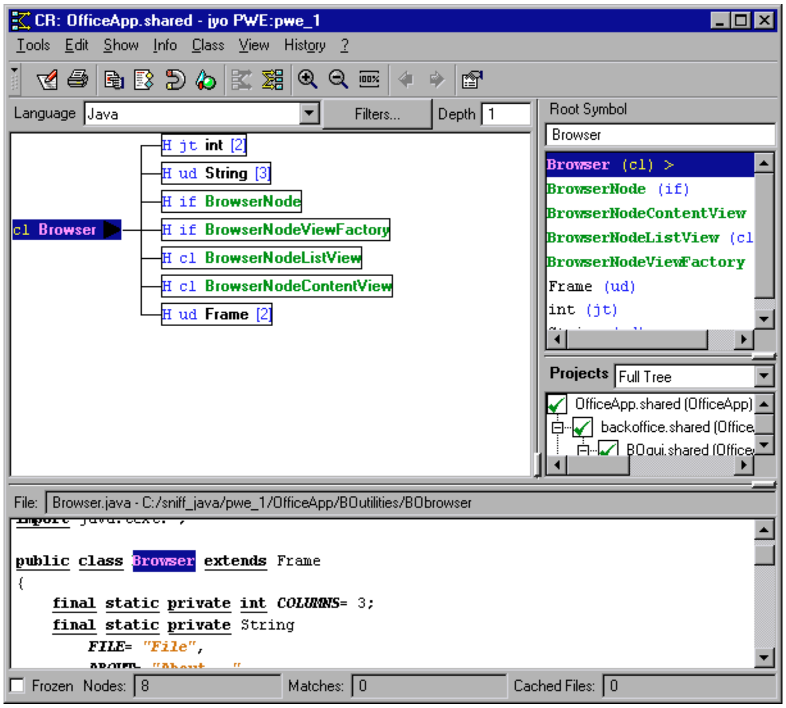
1. In the Hierarchy Browser highlight the class `Browser` (click into the Class List and type `br`).
2. To take a look at the components of the class, choose

   Context menu > Class Browser Refers To Components.
   The Cross Referencer appears and shows the components of `Browser`.
   You can also open the Cross Referencer from any tool where you can select a symbol by clicking on the symbol you are interested in, then choosing
   Info > ***symbolName*** **Refers-To,** or whichever one of the next 3 items in the **Info** menu is appropriate to your needs.

- To get a less cluttered screen, close the Class Browser and the Hierarchy Browser.

Component browsing – Has-A relationships

You opened the Cross Referencer by choosing **Info > Browser Refers-To Components**. The Cross Referencer therefore opens to display all the components of `Browser`.


CR: OfficeApp.shared - jyo PWE:pwe_1

Language: Java   Depth: 1

Root Symbol: Browser

Projects: Full Tree

File: Browser.java - C:/sniff_java/pwe_1/OfficeApp/BOutilities/BObrowser

```
public class Browser extends Frame
{
    final static private int COLUMNS= 3;
    final static private String
        FILE= "File",
```

Frozen   Nodes: 8   Matches: 0   Cached Files: 0

In your Reference Tree you should see the following:

cl `Browser` > H jt `int [2]`

This means that: the **cl**ass `Browser` **H**as **j**ava data **t**ype `int` components `[2` of them`]`.

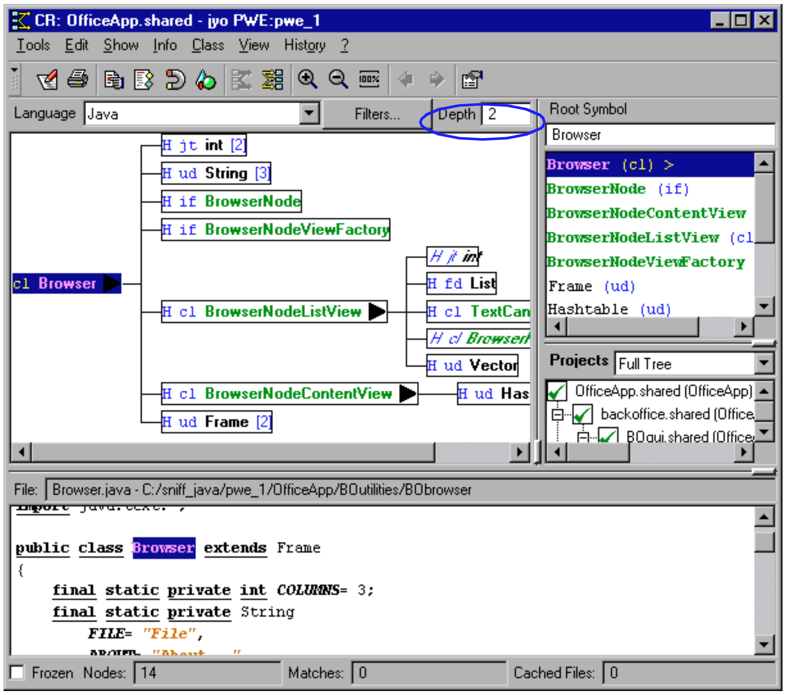
The only component types shown in this view are the **cl**ass type(cl), the **j**ava data **t**ype(jt) and the **i**nter**f**ace type(if). The number of times each component type occurs is only supplied when it is used more than once, e.g., `[2]`.

The Depth Field

Now, to follow the references to the next deeper level:

- In the **Depth** Field enter 2 and hit `<Return>`.

  All the components that have more than one component reference level are shown to two levels.



Language: Java
Depth: 2
Root Symbol: Browser
Projects: Full Tree

File: Browser.java - C:/sniff_java/pwe_1/OfficeApp/BOutilities/BObrowser

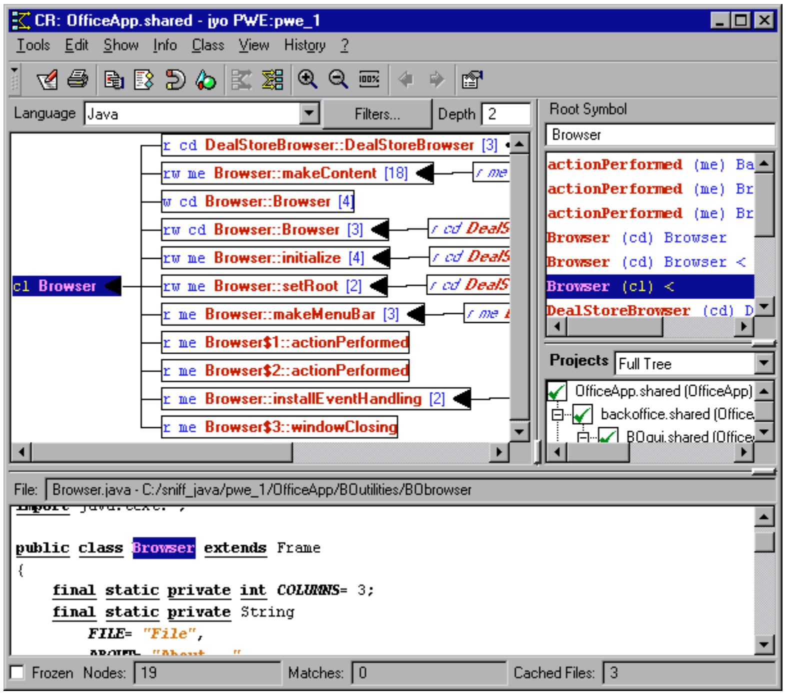
```
public class Browser extends Frame
{
    final static private int COLUMNS= 3;
    final static private String
        FILE= "File",
```

Frozen Nodes: 14 Matches: 0 Cached Files: 0

Impact Analysis

Let’s change perspective and see who `Browser` is referenced by.

- Choose **Context menu > Browser Referred-By**


CR: OfficeApp.shared - jyo PWE:pwe_1
Tools Edit Show Info Class View History ?
Language
Filters...
Depth
Root Symbol
Projects
File:
Frozen

Language: Java   Depth: 2   Root Symbol: Browser   Projects: Full Tree

```
cl Browser
    r cd DealStoreBrowser::DealStoreBrowser [3]
    rw me Browser::makeContent [18]          r me
    w cd Browser::Browser [4]
    rw cd Browser::Browser [3]               r cd DealS
    rw me Browser::initialize [4]            r cd DealS
    rw me Browser::setRoot [2]               r cd DealS
    r me Browser::makeMenuBar [3]            r me
    r me Browser$1::actionPerformed
    r me Browser$2::actionPerformed
    r me Browser::installEventHandling [2]
    r me Browser$3::windowClosing
```

```
actionPerformed (me) Ba
actionPerformed (me) Br
actionPerformed (me) Br
Browser (cd) Browser
Browser (cd) Browser <
Browser (cl) <
DealStoreBrowser (cd) D
```

```
OfficeApp.shared (OfficeApp)
  backoffice.shared (Office
    BOgui.shared (Office
```

File: Browser.java - C:/sniff_java/pwe_1/OfficeApp/BOutilities/BObrowser

```
import java.text.*;

public class Browser extends Frame
{
    final static private int COLUMNS= 3;
    final static private String
        FILE= "File",
        ABOUT= "About...",
```

Nodes: 19   Matches: 0   Cached Files: 3

Navigating through the source code references

To look at the references in the source code:

1. In the Reference View, **<SHIFT>double-click** on the first reference in one of the levels.

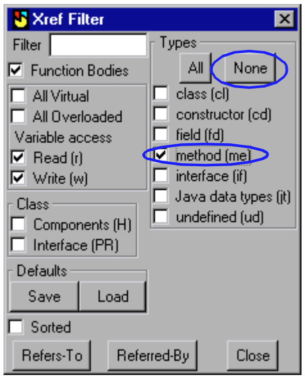
   The Source Editor opens at the reference.

2. Position the Source Editor and the Cross Referencer so that you can see both.
3. In the Source Editor, you can follow the references by choosing the menu command **Show > Next Match**.
4. When you are ready, close the Source Editor.

Call graphing

To see the call graph of, for example, `makeMenubar`

1. Highlight the `makeMenubar` method either in the Reference Graph or the Symbol List.
2. Press the **Filters...** button.



- [✓] Function Bodies
- [ ] All Virtual
- [ ] All Overloaded
- [✓] Read (r)
- [✓] Write (w)
- [ ] Components (H)
- [ ] Interface (PR)
- [ ] Sorted
- [ ] class (cl)
- [ ] constructor (cd)
- [ ] field (fd)
- [✓] method (me)
- [ ] interface (if)
- [ ] Java data types (jt)
- [ ] undefined (ud)

3. In the Xref Filter dialog, press the **None** button to deselect all Symbol Types.
4. Select **method (me)**.
5. Press the **Refers-To** button.

   You now see the call graph of `makeMenubar` to two levels.

- Close the Cross Referencer.

Review

In this chapter you used the Cross Referencer for

- component analysis
- impact analysis
- navigating through source code references
- call graphing

Textual search with the Retriever 5

This chapter is about using the Retriever to:

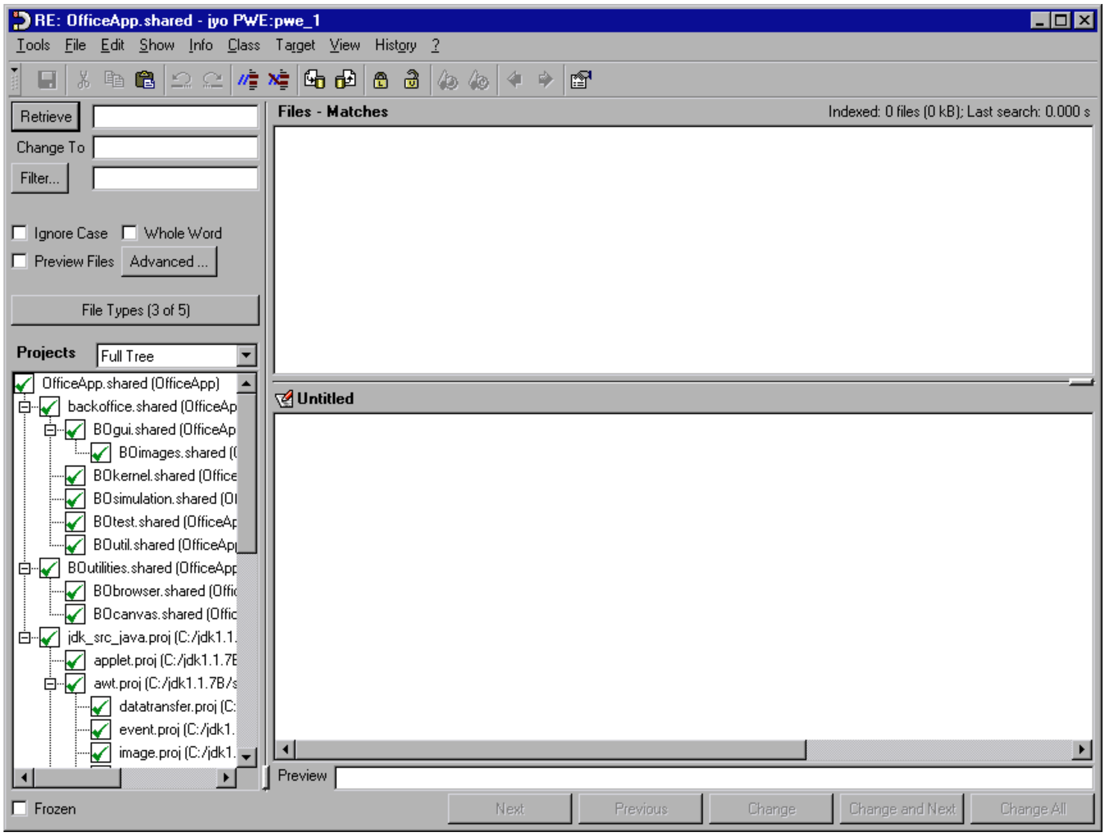
- find every line in all your source files containing a given set of characters
- re-filter search results to conform more closely to your needs
- effectively edit text in combination with the Source Editor

Opening the Retriever

- Open the Retriever by choosing **Tools > Retriever**.

You could also open the Retriever from any tool by highlighting the string you are interested in, then choosing one of the **Info > Retrieve *string*** menu commands.

We chose the **Tools** menu to avoid retrieving from all the JDK projects as well.



OfficeApp.shared (OfficeApp)
backoffice.shared (OfficeAp
BOgui.shared (OfficeAp
BOimages.shared (
BOkernel.shared (Office
BOsimulation.shared (O
BOtest.shared (OfficeAp
BOutil.shared (OfficeAp
BOutilities.shared (OfficeApp
BObrowser.shared (Offic
BOcanvas.shared (Offic
jdk_src_java.proj (C:/jdk1.1.
applet.proj (C:/jdk1.1.7B
awt.proj (C:/jdk1.1.7B/s
datatransfer.proj (C:
event.proj (C:/jdk1.
image.proj (C:/jdk1.

Global Find and Replace

The Retriever can do a lot more than “only” retrieve strings.

- You can apply second stage regular expression filters to narrow down your query to conform more closely to your needs. For information on Regular Expressions, see *Reference Guide — Regular Expressions in SNiFF+*.
- You can directly edit code in the integrated Source Editor
- You can globally find and replace strings in code lines.

Setting the Project Tree

Especially for global editing you wouldn’t want to include the JDK files. Also, you save time by restricting the scope of your queries.

Remember that, in the section Saving a Project Tree view — page 24, you saved a project set which we called `MySources`. You can now use this project set again.

- From the menu, choose **View > Select Project Set > MySources**

Entering a string

Earlier on you looked at `Browser` in various contexts, so now we will retrieve all occurrences of `Browser` from the source files.

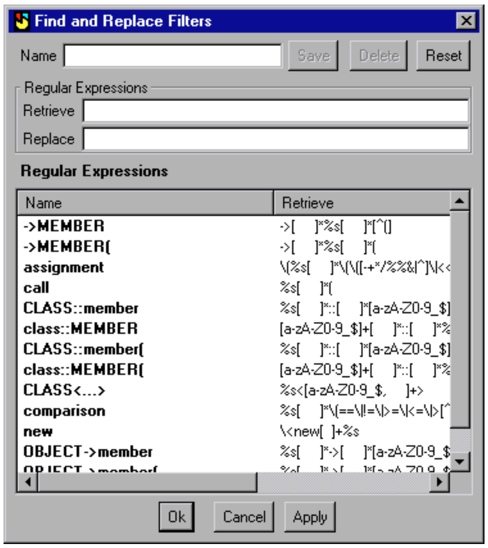
1. In the **Retrieve** field (top-left edit field), enter `Browser`.
2. Press the **Retrieve** button.

   All occurrences of the (sub-)string `Browser` are displayed.

Filtering

For some reason, you might want to see all the method calls that contain the string `Browser`.

- To open the Find and Replace dialog, press the **Filter...** button.



Name	Retrieve
->MEMBER	->[ ]*%s[ ]*[^(]
->MEMBER(	->[ ]*%s[ ]*(
assignment	\(%s[ ]*\(\([-+*/%%&|^]\\|<<
call	%s[ ]*(
CLASS::member	%s[ ]*::[ ]*[a-zA-Z0-9_$]
class::MEMBER	[a-zA-Z0-9_$]+[ ]*::[ ]*%
CLASS::member(	%s[ ]*::[ ]*[a-zA-Z0-9_$]
class::MEMBER(	[a-zA-Z0-9_$]+[ ]*::[ ]*%
CLASS<...>	%s<[a-zA-Z0-9_$,]+>
comparison	%s[ ]*\(==\\|!=\\|>=\\|<=\\|>[^
new	\<new[ ]+%s
OBJECT->member	%s[ ]*->[ ]*[a-zA-Z0-9_$
OBJECT->member(	%s[ ]*->[ ]*[a-zA-Z0-9_$

In the Find and Replace Filter dialog

1. From the dialog’s Regular Expressions List, select **call**.
2. Press the **Apply** Button.

   The filter is applied as a second stage filter to the results of the original query. Note that you can write and save your own regular expressions in this dialog.
3. Press **Ok** or **Cancel** to close the dialog.

The Filter field

The **Filter** field is used for regular expression filters. You have just filled the filter field using the Find and Replace Filter dialog. For simple filters, it is easiest to enter them directly in the field. For example, if you want to know where the string `Browser` is used in a comment:

1. In the edit field next to the **Filter...** button (above the Project Tree), delete the **call** regular expression.
2. Press the **Retrieve** button again to requery.
3. In the edit field next to the **Filter...** button, enter // and hit `<Return>`.

Global Editing

Now, to change `Browser` to something else in all comments:

1. In the **Change To** edit field, enter, e.g., `MyBrowser`.

   Take a look at the **Preview** field below the integrated Source Editor, the code line (highlighted in the Files — Matches List) is shown as it would appear after modification. You can use the **Next** button at the bottom of the tool to look at each line as it would appear after being changed.
2. To change all commented occurrences of `Browser` to `MyBrowser`, press the **Change All** button.
3. In the dialog that appears, press **Yes**.

Remember we checked in all files in Checking in files — page 25, now we have to check out the files that we want to change - i.e., make them writable.

4. In the Locking Status dialog that appears, press the **Select All** button below the **Read Only files** list and press **Check Out**.
5. In the Multiple Check Out dialog that appears, press **Exclusive Lock**.

   You'll notice that the Read Only files list in the Locking Status dialog is now empty.
6. In the Locking Status dialog, press **OK**.

   All occurrences of `Browser` are changed to `MyBrowser`. You can verify this by pressing the **Retrieve** button again to requery.

Undoing global changes

Although the changes you made above are harmless, knowing how to undo global changes is not a bad idea. There are two possibilities to undo the changes you have just made:

- From the menu choose **Edit > Undo Change All**
- Use the Diff/Merge tool to track the changes and merge the differences

  Although the first possibility is easier, we will choose the second possibility to undo changes and to introduce the Diff/Merge tool in the next chapter.

Review

In this chapter you:

- looked for lines in your source files containing a given set of characters
- globally edited text in the Retriever

What’s next

- Close all tools except for the Launch Pad.
- In the Launch Pad, choose **Tools > Project Editor**.

  The Project Editor appears. You will be using the Project Editor in the next part, so leave it open.

Differences between versions of files 6

This chapter is about using the Diff/Merge tool to:

- track changes between files
- to merge differences between versions of files

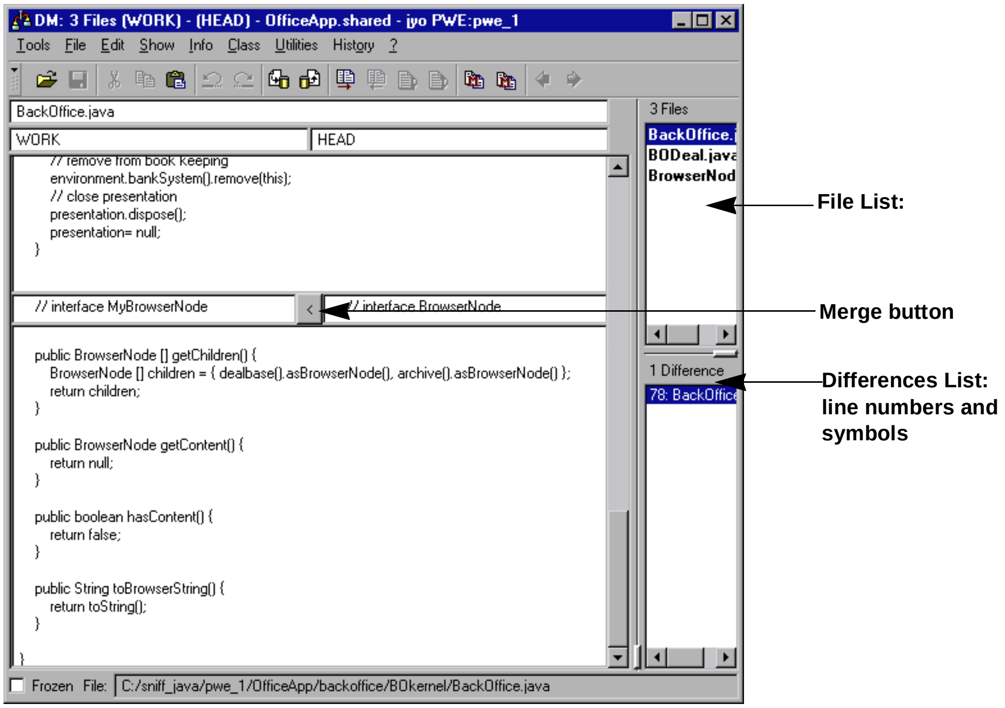
Opening the Diff/Merge tool

- From the Visibility drop-down in the Project Editor, select **Writable**.

  Only the Writable files are displayed.
- In the Project List, `<CTRL>click` on `BackOffice.java, BODeal.java and BrowserNode.java`.

  These files are now highlighted.
- Choose **Context menu > Show Differences...**
- In the dialog that appears, press **Ok**.

The Diff/Merge tool appears.


DM: 3 Files (WORK) - (HEAD) - OfficeApp.shared - jyo PWE:pwe_1
Tools File Edit Show Info Class Utilities History ?
BackOffice.java
WORK
HEAD
3 Files
BackOffice.j
BODeal.java
BrowserNod
1 Difference
78: BackOffice
Frozen File:
File List:
Merge button
Differences List: line numbers and symbols

```
        // remove from book keeping
        environment.bankSystem().remove(this);
        // close presentation
        presentation.dispose();
        presentation= null;
    }
```

```
    // interface MyBrowserNode   <   // interface BrowserNode
```

```
    public BrowserNode [] getChildren() {
        BrowserNode [] children = { dealbase().asBrowserNode(), archive().asBrowserNode() };
        return children;
    }

    public BrowserNode getContent() {
        return null;
    }

    public boolean hasContent() {
        return false;
    }

    public String toBrowserString() {
        return toString();
    }

}
```

C:/sniff_java/pwe_1/OfficeApp/backoffice/BOkernel/BackOffice.java

Remember the files in which you changed all occurrences of Browser to MyBrowser (Global Editing — page 54), these files are now displayed in the File List.

Differences between two file versions

In the above illustration, you can see the differences between the WORK and the HEAD version (the latest version of the file in the Repository) of `BackOffice.java`. The difference in the comment in the two file versions is shown to the left and the right of the Merge button.

1. Press the Merge button (**<**) to merge the differences between the two file versions.
2. Choose **File > Save** to save the changes.

   The Differences List is now empty.
3. Drag the layout handle to the left to increase the width of the File List.
4. Highlight `BODeal.java` in the File List.

   You can see the difference between the two file versions.
5. Follow steps 1 and 2 above to merge the differences and now do the same for `BrowserNode.java`.

Review

This was the last of the SNiFF+ browsing tools to be introduced in this tutorial.
In this part of the SNiFF+J Java Tutorial you were introduced to the following browsing tools:

- The Symbol Browser
- The Hierarchy Browser
- The Class Browser
- The Cross Referencer
- The Retriever
- The Diff/Merge tool

What's next

- Close all tools except for the Launch Pad and the Project Editor.
- From the Visibility drop-down in the Project Editor, select **Writable + Read Only**.

The next part introduces you to:

- Setting up the SNiFF+ build system
- Tools used for editing, compiling and debugging Java code with SNiFF+.

Part IV
Edit/Compile/Debug

SNiFF+ Java Build System 1

This chapter is about

Assumptions

We assume you have completed the steps as outlined under

- Preparing the Environment — page 11
- Setting your Preferences — page 12
- Project Setup — page 14

The following are not necessary for successful compilation, but we also assume that you have completed the steps as outlined under

- Creating a project for the JDK sources — page 20
- Adding the JDK projects to your project — page 23
- Saving a Project Tree view — page 24

What SNiFF+ needs to know

- The root directory where source code packages start. You have already set this in the Project Setup Wizard (page 16) by accepting the default.
- The name and project location of the class implementing the `main` method.

Setting Java Make

The name of the application class is `BackOfficeApplication`. This class is in the `backoffice.shared` project.

In the Project Editor

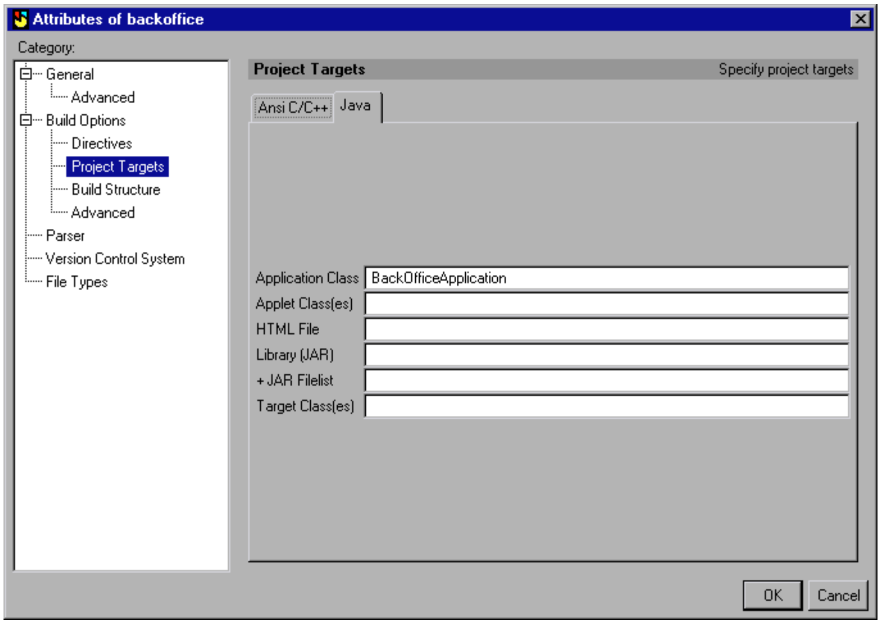
In order to set the Java make attributes, we must first check out `backoffice.shared`.

1. In the Project Tree, checkmark the checkbox next to `backoffice.shared.`
2. In the File List, select `backoffice.shared`.
3. Choose **File > Check Out...**
4. In the Check Out dialog that appears, press **Exclusive Lock**.
5. In the Reload Project Structure dialog, press **Yes**.
6. In the Project Tree, double-click on `backoffice.shared` (the name, not the check-box).

   The Project Attributes dialog opens.

In the Project Attributes dialog

1. Under the **Build Options** node, select **Project Targets**.
2. In the Java tab, enter `BackOfficeApplication` in the **Application Class** field.



Application Class: BackOfficeApplication
Applet Class(es):
HTML File:
Library (JAR):
+ JAR Filelist:
Target Class(es):

3. Press **Ok** to close and apply the Project Attributes.

4. Press **Yes** in the Update Makefiles dialog that appears.

In the Project Editor

You have now set all the necessary Make attribute, and it might be a good idea to save the Project.

- In the Project Tree, the icon warns you that the `backoffice.shared` project has changed. Make sure `backoffice.shared` is highlighted, then choose

  Project > Save backoffice.shared.

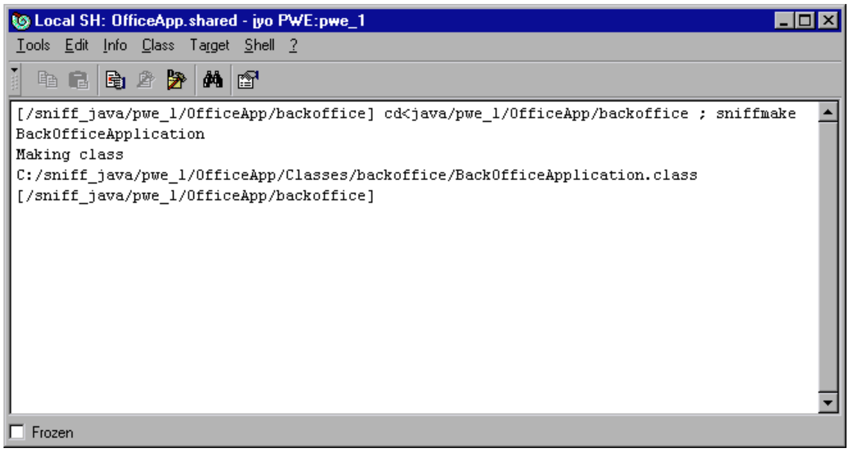
Compiling the application

In the Project Editor

Note that SNiFF+ needs to know where to start Make execution. You tell SNiFF+ this by highlighting the appropriate project. In the example project, Make execution starts in
`backoffice.shared`
where you specified the Java target. So:

1. In the Project Tree, make sure `backoffice.shared` is highlighted.
2. Choose **Target > Make > BackOfficeAppliation** to compile the application.

   A Shell Tool appears, after compilation it should look similar to the following:


Local SH: OfficeApp.shared - jyo PWE:pwe_1
Tools Edit Info Class Target Shell ?
Frozen

```
[/sniff_java/pwe_1/OfficeApp/backoffice] cd<java/pwe_1/OfficeApp/backoffice ; sniffmake
BackOfficeApplication
Making class
C:/sniff_java/pwe_1/OfficeApp/Classes/backoffice/BackOfficeApplication.class
[/sniff_java/pwe_1/OfficeApp/backoffice]
```

On successful compilation, the byte-code `.class` files are generated to
`<sniff_java>/pwe_1/OfficeApp/Classes/`
Remember you specified this attribute during project setup (In the “Create New SNiFF+ Project” page — page 16).

Troubleshooting

- If compiler errors are reported in the shell, something may have gone wrong with the setup of the projects Java Make attributes. Try going through the steps in this tutorial again, carefully check them, compare screenshots, and re-compile.
- Make sure your `PATH` environment variable points to

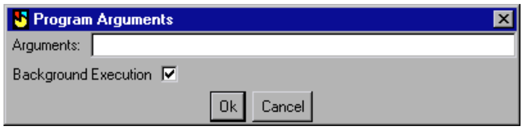
  `<your_jdk_installation_dir>/bin`
- **On Windows NT**: Problems can be caused by a `CLASSPATH` environment variable that does not conform to the upper case conventions (e.g., an application may set an environment variable as `ClassPath`). If this is the case, set a single environment variable, `CLASSPATH`, to point to all the paths in an existing `CLASSPATH` as well as any variations thereof, such as `ClassPath.`

Running the application

In the Project Editor

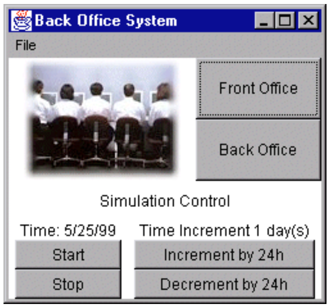
1. Make sure that `backoffice.shared` is highlighted (the project containing the application class) in the Project Tree.
2. Choose **Target > Run BackOfficeApplication.sh**.

   The compilation of a Java Program does not result in an executable. SNiFF+ hides this fact by generating a start-up shell script. This is why the targets have the ending "`.sh`". The Program Arguments dialog appears:



3. Press **Ok**.

   The application is started and appears on your screen:



4. Close the application by choosing **File > Exit** in the application window.
5. Close the Shell Tool.

Review

In this chapter you

- set the Java Make attributes in the Project Attributes dialog
- compiled the application
- tested the application

Editing and Compiling 2

This chapter is about

Checking out and opening a file

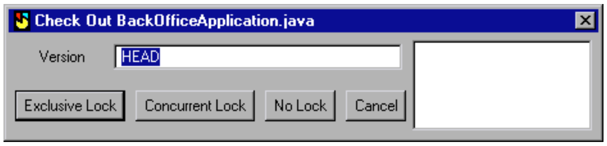
Remember, you are now working with a version controlled project. You checked in all your source files, so now they are read-only. To be able to edit a file, it first has to be checked out.

In the Project Editor

We will check out and edit the file `BackOfficeApplication.java`

1. In the Project Tree, make sure that `backoffice.shared` is checkmarked.
2. In the File List, highlight `BackOfficeApplication.java` and right-click to choose

   Context menu > Check Out BackOfficeApplication.java...
   The Check Out dialog appears.



   By default, the HEAD version is suggested, which at this time is still the same as the INIT version. With repeated editing, the content will obviously change over time.

3. To get a writable version of the file, press **Exclusive Lock** (the other two lock buttons are relevant only in team projects).

   Notice that, in the Project Editor's File List, the file name now appears in bold typeface. This indicates that it is writable.

4. To open `BackOfficeApplication.java` double-click on it in the Project Editor's File List.

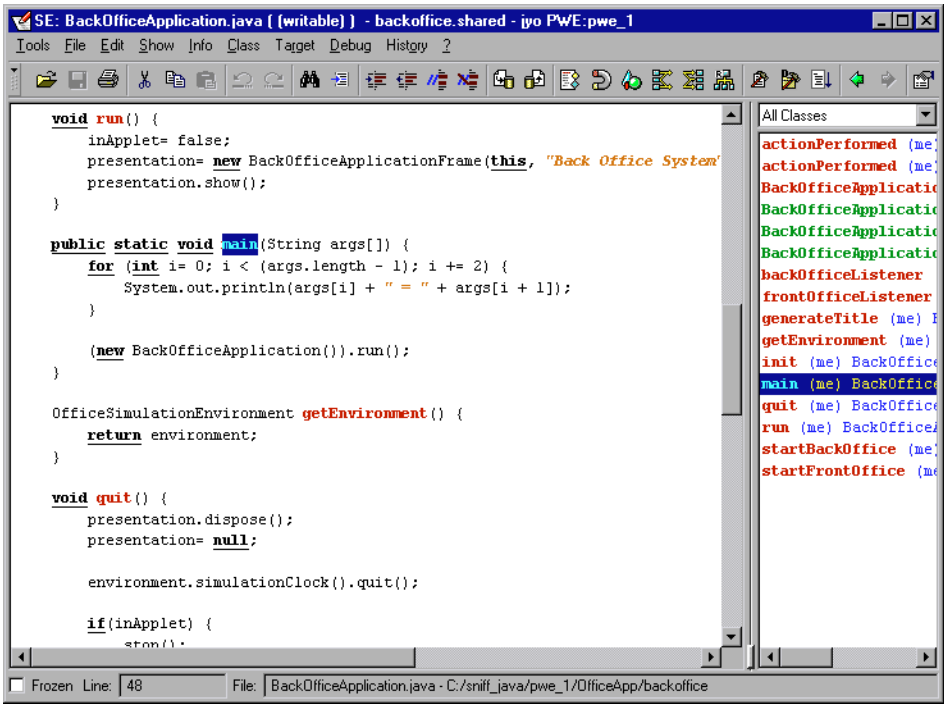
   The file is opened in the Source Editor.

Compilation errors

Now, we’ll edit the file so as to induce a compilation error. After attempting to compile, the error will be reported in the Shell tool. From the error message, you can go straight to the point in the source code where the error was found.

1. In the Source Editor’s Symbol List (on the right), click on `main (me)` — the `(me)` stands for ’method’.

   The editor positions to `main`.


SE: BackOfficeApplication.java ((writable)) - backoffice.shared - jyo PWE:pwe_1
Tools File Edit Show Info Class Target Debug History ?
All Classes

```
void run() {
    inApplet= false;
    presentation= new BackOfficeApplicationFrame(this, "Back Office System"
    presentation.show();
}

public static void main(String args[]) {
    for (int i= 0; i < (args.length - 1); i += 2) {
        System.out.println(args[i] + " = " + args[i + 1]);
    }

    (new BackOfficeApplication()).run();
}

OfficeSimulationEnvironment getEnvironment() {
    return environment;
}

void quit() {
    presentation.dispose();
    presentation= null;

    environment.simulationClock().quit();

    if(inApplet) {
        stop();
```

```
actionPerformed (me)
actionPerformed (me)
BackOfficeApplicatio
BackOfficeApplicatio
BackOfficeApplicatio
BackOfficeApplicatio
backOfficeListener
frontOfficeListener
generateTitle (me) I
getEnvironment (me)
init (me) BackOffice
main (me) BackOffice
quit (me) BackOffice
run (me) BackOfficeA
startBackOffice (me)
startFrontOffice (me
```

Frozen Line: 48 File: BackOfficeApplication.java - C:/sniff_java/pwe_1/OfficeApp/backoffice

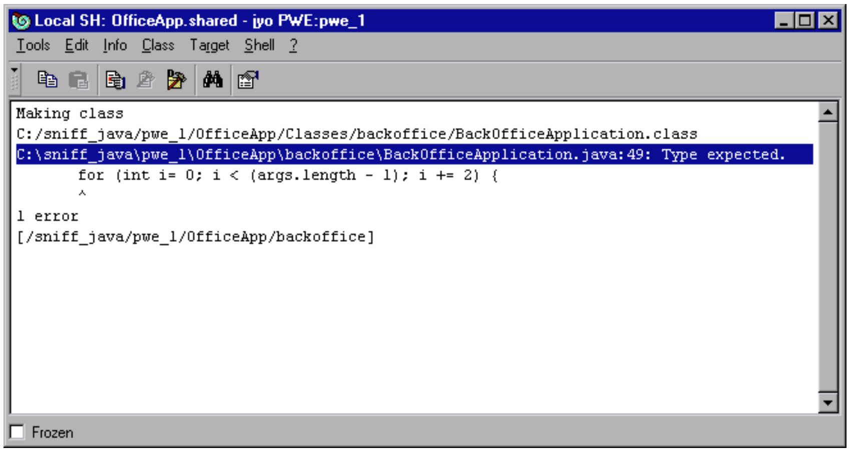
2. To comment out the line with the `main` method, click into it and choose **Edit > Comment**.

   Note that you can comment out any number of lines by highlighting them and choosing this menu item.

3. For the changes to take effect, choose **File > Save**.

4. To compile the file, choose **Target > Make File BackOfficeApplication.class**.

A Shell Tool appears and SNiFF+ tries to compile the modified file.
The error is reported in the Shell Tool in the form `file:line: error`, followed by the actual line.


Local SH: OfficeApp.shared - jyo PWE:pwe_1
Tools Edit Info Class Target Shell ?
Frozen

```
Making class
C:/sniff_java/pwe_1/OfficeApp/Classes/backoffice/BackOfficeApplication.class
C:\sniff_java\pwe_1\OfficeApp\backoffice\BackOfficeApplication.java:49: Type expected.
        for (int i= 0; i < (args.length - 1); i += 2) {
        ^
1 error
[/sniff_java/pwe_1/OfficeApp/backoffice]
```

In the Shell tool

- To see the error, click on the reported error (highlighted in the illustration) and choose **Edit > Show Error**.

  The Source Editor positions to detected error, and the affected line is highlighted.

In the Source Editor

To uncomment the line that induced the error:

1. Click into the line you commented out earlier and choose **Edit > Uncomment**.
2. For the changes to take effect, choose **File > Save**.
3. Choose **Target > Make File BackOfficeApplication**.**class** again to re-compile.

- Close the Source Editor and the Shell tool to avoid screen cluttering.

Review

In this chapter you

- checked out a file to get a writable version
- edited a file
- compiled the file
- found and fixed a compilation error

Debugging 3

In this introduction to the sniffjdb debugger, you will see how to:

- set the SNiFF+ Java debugger (sniffjdb) in your Preferences
- set breakpoints
- watch variables
- watch threads
- make run-time changes to a simple variable and watch the results

Setting the debugger in the Preferences

You need to tell SNiFF+ that you will be using the SNiFF+ debugger for Java, sniffjdb. To set sniffjdb as your preferred debugger:

- From any open SNiFF+ tool, choose **Tools > Preferences...**

In the Preferences

1. Select the **Platform** node.
2. In the Platform view, highlight the platform, which is selected in the Default Platform drop-down, in the platform list.
3. Press the **Set Writable** button.

   This makes a writable copy, you can later always revert to the default settings.
4. Select the Debugger tab.

In the Debugger view

1. From the **Adapter drop-down**, choose **sniffjdb(Java)**.
2. Press **Ok** to close the dialog.

The debugger command line

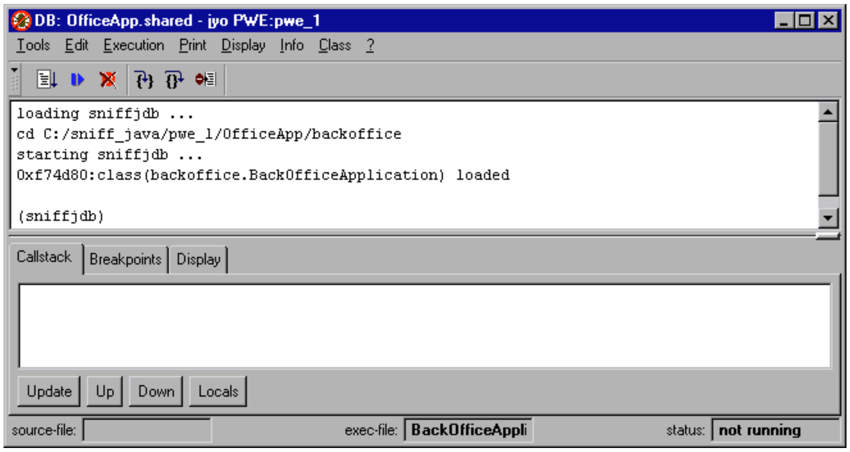
After starting the debugger, you will set two breakpoints and watch what happens to a variable between the first and the second breakpoint.

In the Project Editor

To start the debugger:

1. In the Project Tree, make sure that `backoffice.shared` (the project with the target classes) is highlighted. If it isn't, click on its name to highlight it.
2. Choose **Target > Debug... > BackOfficeApplication.sh**

   The debugger command line shell opens.


DB: OfficeApp.shared - iyo PWE:pwe_1
Tools Edit Execution Print Display Info Class ?
Callstack Breakpoints Display
Update Up Down Locals
source-file:
exec-file: BackOfficeAppli
status: not running

```
loading sniffjdb ...
cd C:/sniff_java/pwe_1/OfficeApp/backoffice
starting sniffjdb ...
0xf74d80:class(backoffice.BackOfficeApplication) loaded

(sniffjdb)
```

In the Debugger

- For a summary of command line commands, type `help` at the command line prompt, `(sniffjdb)`.

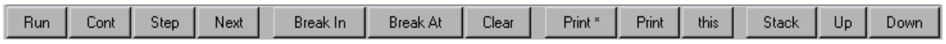
  Note that many of these commands can also be posted from the Source Editor.

In the Project Editor

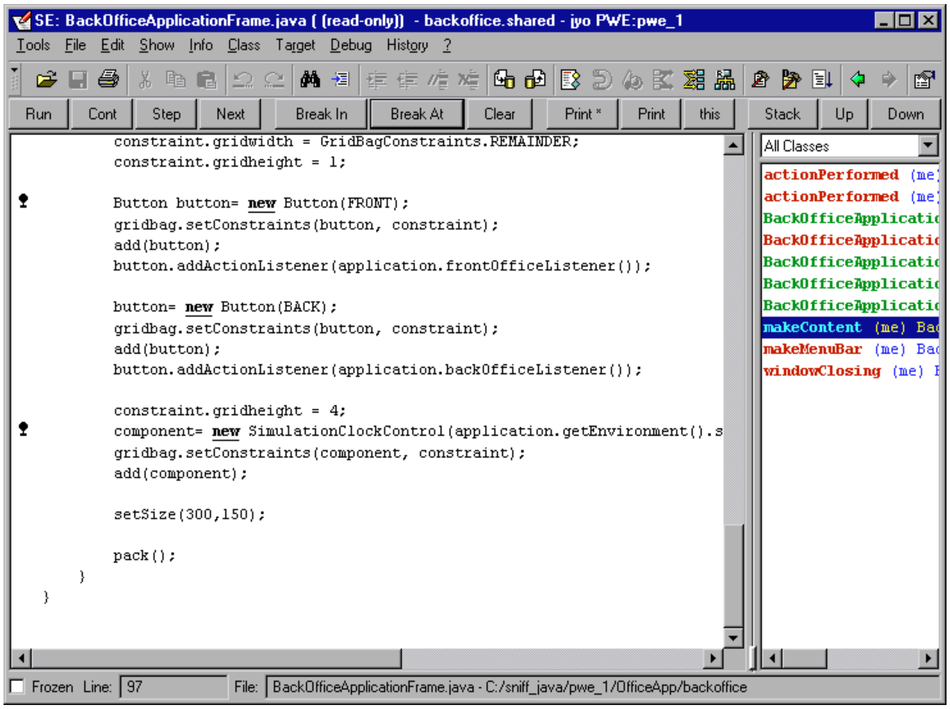
1. You will be setting breakpoints in a source file, `BackOfficeApplicationFrame.java`, which is part of `backoffice.shared`, so make sure that `backoffice.shared` is checkmarked in the Project Editor's Project Tree.

2. To open the file, double click on the `BackOfficeApplicationFrame.java` file in the File List.

The file is opened in the Source Editor and, because you are in debug mode, a row of buttons for the most commonly needed debug commands has been added below the tool bar.



Setting Breakpoints


SE: BackOfficeApplicationFrame.java ((read-only)) - backoffice.shared - jyo PWE:pwe_1
Tools File Edit Show Info Class Target Debug History ?
Run
Cont
Step
Next
Break In
Break At
Clear
Print *
Print
this
Stack
Up
Down
All Classes

```
        constraint.gridwidth = GridBagConstraints.REMAINDER;
        constraint.gridheight = 1;

        Button button= new Button(FRONT);
        gridbag.setConstraints(button, constraint);
        add(button);
        button.addActionListener(application.frontOfficeListener());

        button= new Button(BACK);
        gridbag.setConstraints(button, constraint);
        add(button);
        button.addActionListener(application.backOfficeListener());

        constraint.gridheight = 4;
        component= new SimulationClockControl(application.getEnvironment().s
        gridbag.setConstraints(component, constraint);
        add(component);

        setSize(300,150);

        pack();
    }
}
```

```
actionPerformed (me)
actionPerformed (me)
BackOfficeApplicatio
BackOfficeApplicatio
BackOfficeApplicatio
BackOfficeApplicatio
BackOfficeApplicatio
makeContent (me) Ba
makeMenuBar (me) Ba
windowClosing (me) B
```

Frozen Line: 97 File: BackOfficeApplicationFrame.java - C:/sniff_java/pwe_1/OfficeApp/backoffice

In the above illustration the two breakpoints have already been set. To set these breakpoints:

1. Choose **Edit > Go to line...**

The Goto dialog appears.

2. To go to the line for the first breakpoint, type `90` in the Goto dialog and press **Go to**.

You are positioned at line 90 in the file.

3. To set the breakpoint, press **Break At**.

4. To set the second breakpoint, choose **Edit > Go to line...**, type `101` in the Goto dialog and press **Go to**.

   You are positioned at line 101 in the file.

5. To set the breakpoint, press **Break At**.

Look at the code lines immediately preceding our two breakpoints. You can see that in each case `constraint.gridheight` has been assigned a value (1 and then 4).
After stopping the application at the first breakpoint, we will take a look at the variable's run-time value, then stop at the next breakpoint and use the Variable Viewer to verify that this value has been changed.

The Variable Viewer

In the Source Editor

1. Press **Run**.

   The Program Arguments dialog appears. Press **Ok**, and the application starts and then stops at the first breakpoint.

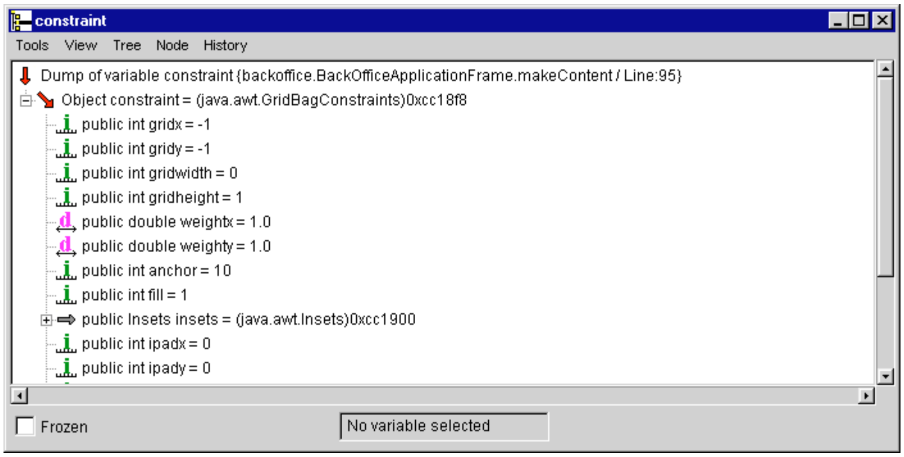
2. Double-click on `constraint` in the line preceding the breakpoint to highlight it.
3. To view `constraint`, press **Print**.

   The `constraint` object is displayed in the Variable Viewer.

In the Variable Viewer

- Expand the node to see the member variables by clicking on the node next to `Object constraint`.

  As you can see, `constraint.gridheight` has the value 1, as expected.



```
Dump of variable constraint {backoffice.BackOfficeApplicationFrame.makeContent / Line:95}
Object constraint = (java.awt.GridBagConstraints)0xcc18f8
    public int gridx = -1
    public int gridy = -1
    public int gridwidth = 0
    public int gridheight = 1
    public double weightx = 1.0
    public double weighty = 1.0
    public int anchor = 10
    public int fill = 1
    public Insets insets = (java.awt.Insets)0xcc1900
    public int ipadx = 0
    public int ipady = 0
```

- Back in the Source Editor, to continue the execution of the application, press **Cont**.

  The application stops at the next breakpoint.
- In the Variable Viewer, choose **View > Update**.

  The Variable Viewer now shows the new value of `constraint.gridheight` to be 4, the value at the previous breakpoint [1] is also shown.

The Threads Viewer

The application is now stopped at the second breakpoint. Now, let the application run its course to start-up before taking a look at the threads.

In the Source Editor

1. Press **Cont**.

   The application window appears in the top left-hand corner of your screen. Check to see that it isn't hidden by any other windows.

2. Close the Source Editor to get a less cluttered screen.

In the Variable Viewer

- To start the Threads Viewer, choose **Tools > Threadsview**.

  You can also start the Threads Viewer from the Debugger Command Line window, by typing `vt` (short for `viewthreads`) at the prompt.

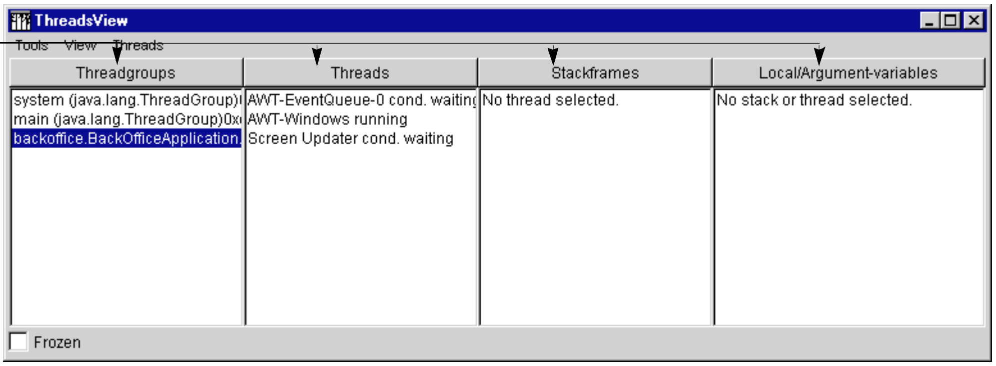
In the Threads Viewer

The first line in the **Threadgroups** column is highlighted. This means you see only the information that applies to this thread group.

1. To see all the threads, highlight each threadgroup by clicking on it.
2. To see only the threads specific to the application itself, deselect all the highlighted items in the **Threadgroups** column, except the last one, by clicking on them.

You now see the threads generated by the application itself.

Press these buttons to update the columns



```
system (java.lang.ThreadGroup)( | AWT-EventQueue-0 cond. waitin( | No thread selected. | No stack or thread selected.
main (java.lang.ThreadGroup)0x( | AWT-Windows running
backoffice.BackOfficeApplication. | Screen Updater cond. waiting
```

Watching a thread

Make sure you can see both the Threads Viewer and the Back Office System window on your screen.

In the **Threads** column of the Threads Viewer, there are only 3 items at present (this may vary, depending on your operating system).

In the Back Office System window, there is a button on the left called **Start**. This button starts a timer thread, which increments the date by the number of days set on the right. And this is the new thread you will start.

1. In the Back Office System window, press **Start**.

   The time, as can be seen above the button, starts incrementing by one day per second.

2. To update the **Threads** column in the Threads Viewer, press the **Threads** button at the top of the **Threads** column.

   A new item is added to the list: `Thread-2 cond. waiting` (the name of the thread and its condition, in this case: waiting).

   Note that you can also choose an automatic update option from the Threads Viewer's **View** menu.

Testing run-time changes in variables

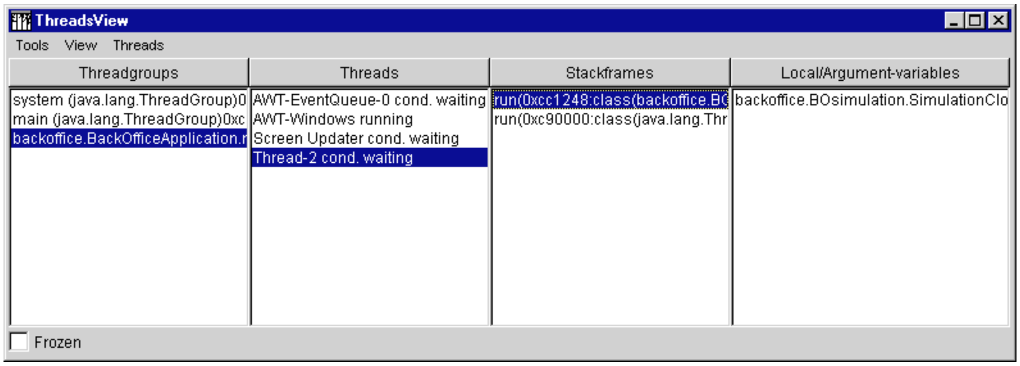
You can change the contents of simple variables and test the resulting changes while the program is running in the debug mode. We will test changes in the time increment value of the simulated clock.

1. In the **Threads** column, highlight the new thread that appeared when you started the timer thread.

2. In the **Stackframes** column, highlight the first item.

   The local variables of the SimulationClock class appear in the last column of the Threads Viewer.

3. To look at these variables, double-click on the item in the last column.



system (java.lang.ThreadGroup)0
main (java.lang.ThreadGroup)0xc
backoffice.BackOfficeApplication.r

AWT-EventQueue-0 cond. waiting
AWT-Windows running
Screen Updater cond. waiting
Thread-2 cond. waiting

run(0xcc1248:class(backoffice.BO
run(0xc90000:class(java.lang.Thr

backoffice.BOsimulation.SimulationClo

In the Variable Viewer

We will use the Variable Viewer to do a run-time test of a new value for the time increment, so please make sure you can see both the Variable Viewer and the Back Office System window.

1. In the Variable Viewer, expand the node of the object.

You can now see, among other things, the variable:

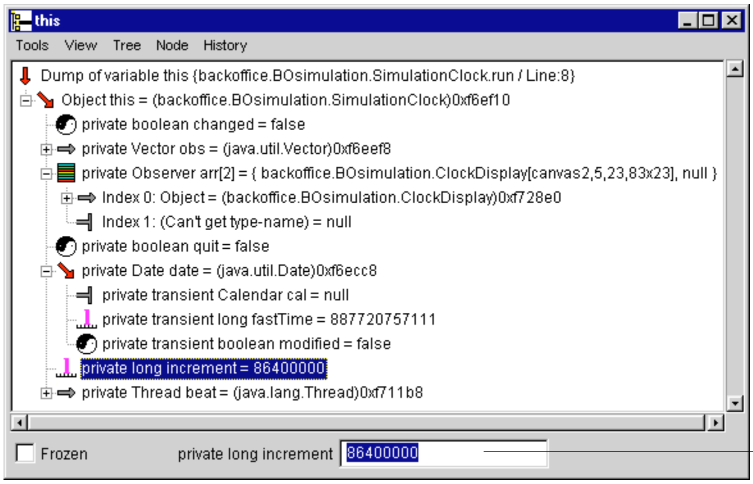
`private long increment = 86400000`

2. Double-click on this variable.

The variable value appears highlighted in the **Variable** Field at the bottom of the tool. The value represents one day. If you add a zero to this number, you will have the value for 10 days.

3. Click into the **Variable** field and add a zero to the value, then hit `<Return>`.

Take a look at the Back Office System window. The time value in the Back Office System window is now incremented by 10 days, instead of 1.
Note that this value is not persistent, and applies only for this debugging session.



```
Dump of variable this {backoffice.BOsimulation.SimulationClock.run / Line:8}
Object this = (backoffice.BOsimulation.SimulationClock)0xf6ef10
  private boolean changed = false
  private Vector obs = (java.util.Vector)0xf6eef8
  private Observer arr[2] = { backoffice.BOsimulation.ClockDisplay[canvas2,5,23,83x23], null }
    Index 0: Object = (backoffice.BOsimulation.ClockDisplay)0xf728e0
    Index 1: (Can't get type-name) = null
  private boolean quit = false
  private Date date = (java.util.Date)0xf6ecc8
    private transient Calendar cal = null
    private transient long fastTime = 887720757111
    private transient boolean modified = false
  private long increment = 86400000
  private Thread beat = (java.lang.Thread)0xf711b8
```

private long increment 86400000

Closing the debugger

To close the debugger, choose **Tools > Close Tool** from its menu.

Review

This was the last chapter of the edit/compile/debug part. In this introduction to the sniffjdb debugger, you learnt how to:

- set the SNiFF+ Java debugger (sniffjdb) in your Preferences
- set breakpoints
- watch variables
- watch threads
- make run-time changes to a simple variable and watch the results

What’s next

Later in the tutorial, you will create a new working environment for a new team member. Since your team member needs access to the most current source code, you must check in the files you’ve modified. To do so:

In the Project Editor

1. Choose **View > Select Project Set > MySources**.
2. In the File Status drop-down, choose **Writable**.

   Only writable files in the project set are displayed.
3. Choose **File > Select All** and then **File > Check In**.

In the Check In dialog

1. In the **Change Set** field, enter a name for the change set e.g., `second_file_set`.
2. In the **Comment** field, enter a comment e.g., `modified files`.
3. Press **Ok**.

   The files in the File List are no longer in bold typeface. This means that they are now read-only.
4. Close all tools except for the Launch Pad.

Freezing the Project 4

All your working environments are now up-to-date, your source files are compilable, and you have a stable version of your byte-code. In this chapter, you will learn how to create a “virtual snapshot” of the project (or, to be exact, of its source files). You do this in SNiFF+ by associating the current state (configuration) of all project source files with a single symbolic name. The process of creating a single configuration and associating it with a symbolic name is called “freezing a configuration”.

You can freeze configurations in the Configuration Manager. You can also use this tool to view the lists of configurations of your projects and compare two configurations with each other. To learn more about the Configuration Manager, please refer to the *User’s Guide* and the *Reference Guide*.

This chapter is about:

- freezing a project
- looking at a file’s history information

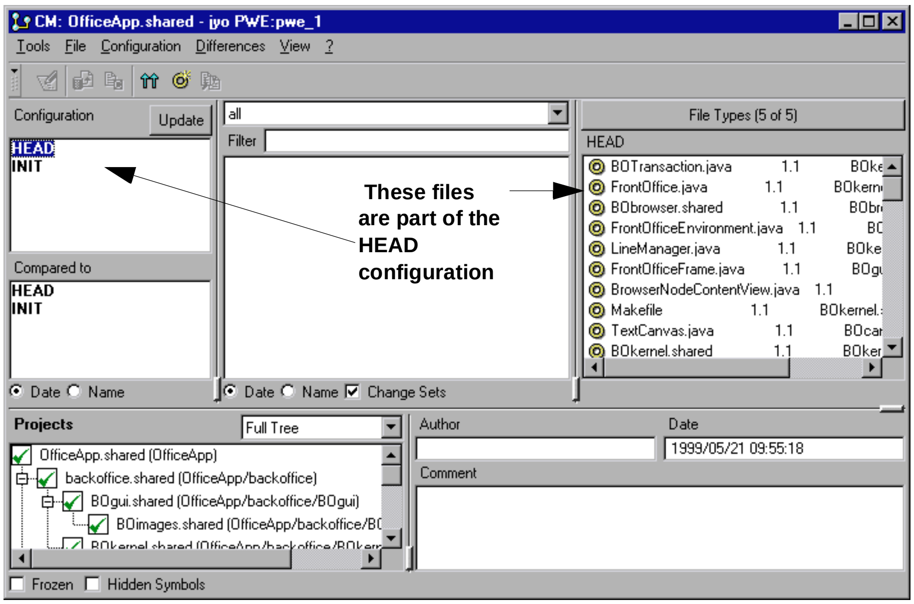
Freezing the project

- From any open SNiFF+ tool, choose **Tools > Configuration Manager**.

In the Configuration Manager

1. In the Configuration List, select the **HEAD** configuration.

   The project's configuration information is loaded into the Configuration Manager. Your Configuration Manager should look something like the following:



Configuration: HEAD, INIT

Compared to: HEAD, INIT

HEAD

BOTransaction.java	1.1	BOke
FrontOffice.java	1.1	BOkern
BObrowser.shared	1.1	BObr
FrontOfficeEnvironment.java	1.1	BC
LineManager.java	1.1	BOke
FrontOfficeFrame.java	1.1	BOgu
BrowserNodeContentView.java	1.1	
Makefile	1.1	BOkernel.:
TextCanvas.java	1.1	BOcar
BOkernel.shared	1.1	BOker

Projects:
- OfficeApp.shared (OfficeApp)
  - backoffice.shared (OfficeApp/backoffice)
    - BOgui.shared (OfficeApp/backoffice/BOgui)
      - BOimages.shared (OfficeApp/backoffice/BC
    - BOkernel.shared (OfficeApp/backoffice/BOkern

Date: 1999/05/21 09:55:18

2. Choose the **Configuration > Freeze Head...**.

   The Freeze Head dialog appears.

3. Enter a name for the new configuration in the **Configuration** field of the dialog (e.g. `OfficeApp_V1.0`) and press **Ok**.

   The Configuration List is now updated to include the newly created configuration.

Looking at the history of a file

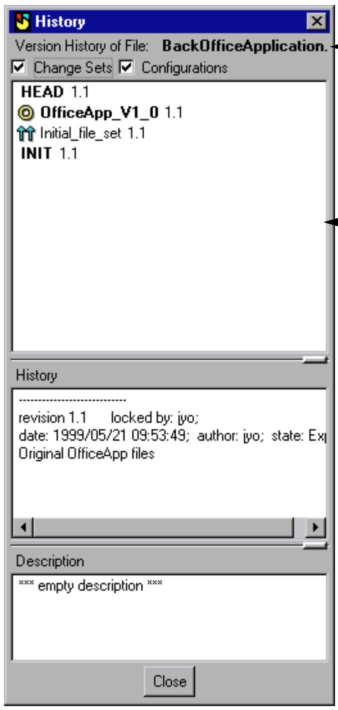
We will now take a look at the history of one of the project files in the Project Editor.

- In the Configuration Manager, choose **Tools > Project Editor**.

In the Project Editor

1. In the File List, highlight `BackOfficeApplication.java`.
2. Select the **History** check box at the bottom of the tool.

   A File History dialog appears:



```
------------------------------
revision 1.1      locked by: jyo;
date: 1999/05/21 09:53:49;  author: jyo;  state: Ex
Original OfficeApp files
```

```
*** empty description ***
```

   In the Symbols view, the Version Tree of the selected file is displayed. Since only one version of the project files has been checked in so far, the Version Tree only displays this version (`INIT`).

   `INIT` is used by SNiFF+ to refer to the initial version of a file in the Repository. The version number of the `INIT` version of a file is always `1.1`. The latest version on the main trunk or branch of a file's version tree is called HEAD. In this example, the HEAD and `INIT` versions of the file will naturally be the same.

   A circle next to the file's version in the Version Tree indicates that the version is part of a configuration. The configuration name comes after the circle, followed by the version number.

3. Press the **Close** button to close the dialog.

In the Launch Pad

- Close `OfficeApp.shared` by choosing **Project > Close Project OfficeApp.shared**.

Review

In this chapter you

- froze a stable version of the project in your PWE
- looked at a file’s history information

Adding new team members 5

This chapter is about

- Preparing the Environment
- Adding new working environments
- Initializing the new working environment
- Updating working environments

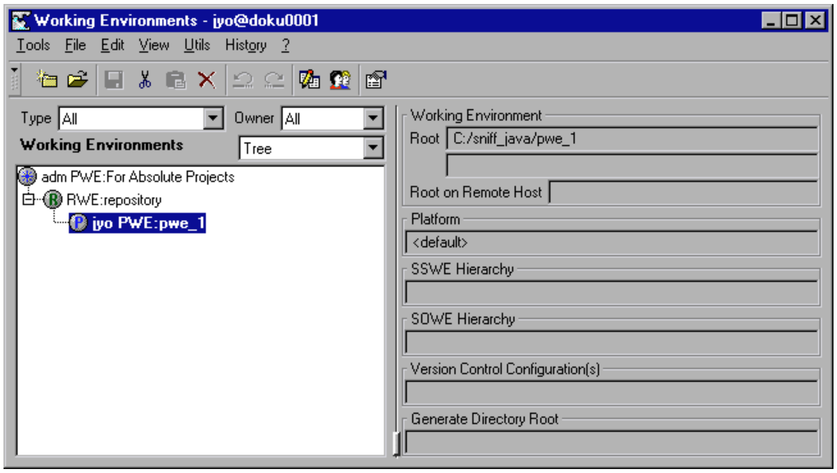
Preparing the Environment

- Copy the `OfficeApp.shared` root project from the `<sniff_java>/pwe_1/OfficeApp` directory to the `<sniff_java>/pwe_2/OfficeApp` directory.

Adding new working environments

- To open the Working Environments tool from any tool, choose

  Tools > Working Environments.


Working Environments - iyo@doku0001

Type: All
Owner: All
Working Environments: Tree

adm PWE:For Absolute Projects
RWE:repository
iyo PWE:pwe_1

Working Environment
Root: C:/sniff_java/pwe_1
Root on Remote Host:
Platform: <default>
SSWE Hierarchy:
SOWE Hierarchy:
Version Control Configuration(s):
Generate Directory Root:

1. Click on **RWE: repository** to highlight it.
2. Choose **Edit > New Private based on Repository**.

   The New Private based on Repository dialog appears.

In the New Private based on Repository dialog

1. In the Working Environment field, enter the name for the second working environment, e.g., **pwe_2**.
2. Press the **Directory...** button to the right of the **Root** field.
3. Navigate to the `<sniff_java>/pwe_2` directory and press **Open** and then **Select**.
4. In the **Owner** field, enter the User name for the new team member e.g., `eric`.

   He is now the owner of the new working environment.
5. Press **Ok**.

In the Working Environments tool

- Double-click on `eric PWE: pwe_2`.

  An empty Open Project dialog appears.

Initializing the new working environment

In the Open Project dialog

1. Press the **Update List** button to display all the projects in the Project List.
2. From the Project List, select `OfficeApp.shared` and press **Open**.
3. In the Open Project dialog that appears, press **Yes**.
4. In the dialog that appears, checkmark **Repeat** and press **Check Out**.
5. In the Check Out dialog that appears, checkmark **Repeat** and press **No Lock**.
6. In the Project File dialog that appears, checkmark **Repeat** and press **Check Out**.

   The project's contents and structure are displayed in the Project Editor. This working environment is now ready for your new team member.

Updating working environments

When a team member checks out a file, the checked-out version is locked in the Repository, and a local copy is made in the team member's PWE. When a team member is satisfied with changes he/she has made to a checked-out file (compilable!), he/she checks it back in. This means that the new (checked-in) version replaces the older (checked-out) version in the Repository. However the other PWE's have the older version of the file and thus the working environments are no longer consistent with each other and need to be synchronized. Updates should be done on a regular (daily) basis, especially if you have a large development team. To update working environments:

- In the Project Tree of the Project Editor, choose

  Context menu > Select from All Projects.

- In the Project Editor, choose

  Project > Synchronize Checkmarked Projects...

Review

In this part of the SNiFF+J Java Tutorial you were introduced to:

- Adding new working environments for team members
- Initializing the new working environment
- updating working environments

Part V
Technical Reference

Introduction and Basic SNiFF+ Concepts 1

Introduction

The hands-on tutorial with example code shows you how to set up projects using the Project Setup Wizard and introduces you to the various SNiFF+ tools.

This Technical Reference is intended as a reference to Java-specific aspects of using SNiFF+. As such, you will not find much information relating to SNiFF+ in general; please refer to the SNiFF+ *User's Guide* and *Reference Guide* for more information in this respect. All SNiFF+ documentation is available online under the Launch Pad's **Help(?)** menu.

- This chapter introduces two basic SNiFF+ concepts, *Projects* and *Working Environments,* and sketches a typical SNiFF+ Java development system.
- The following chapters look at source code directory structures and the implications for project setup and updating in SNiFF+, as well as working environment and project set up for Java projects.
- Further chapters cover various technical aspects such as compilation, execution, debugging, the class path concept, Java IDL, JNI and RMI generation etc., all in the context of SNiFF+. A reference description of the SNiFF+ Java debugger is also included.
- A chapter on upgrading issues looks at SNiFF+ backward compatability among different SNiFF+ versions, as well as a few points regarding upgrading libraries (e.g. JDK).
- The manual closes on a description of how to get started with the **Visaj GUI Builder** integration. More information on SNiFF+ integration features, as well as on Visaj itself, is available online in the Visaj Class Editor.

SNiFF+ Java Shared Projects and Working Environments

This section very briefly introduces the SNiFF+ *Project* and *Working Environment* concepts as they apply to Java software systems. For more detailed information about projects and working environments, please refer to the *User's Guide*.

Projects

Shared Projects

A SNiFF+ Shared Project is, as the name suggests, suitable for team development.

Team members must be able to share and access source code to make changes to files and/or structure, regardless of what other team members are doing.

This means that the integrity of the project system as a whole needs to be maintained in some way, which is why Shared Projects are always used in conjunction with Working Environments and a configuration management and a version control (CMVC) tool.

Browsing-Only Projects

Browsing-only projects are quick and easy to set up, and require no further maintenance. You would use this type of project for stable libraries (e.g., the JDK sources) that are not subject to constant change, and where you do not require Make and dependency information to be generated.

For Java software systems, you would typically add the JDK sources to your own sources as a Browsing-Only subproject. This allows you to follow references and hierarchies all the way back to the root.

Working Environments

For Java projects, you need only two types of Working Environment, a Repository and Private Working Environments. These are described below, and how to go about setting them up is summarized in the following chapter.

Repository Working Environment (RWE)

You and your team members access and modify a permanent data Repository using commands provided by your underlying configuration management and version-control (CMVC) tool. SNiFF+ provides an interface to your CMVC tool.

In the Java tutorial, we use RCS, the CMVC tool provided with the SNiFF+ package.

Private Working Environment (PWE)

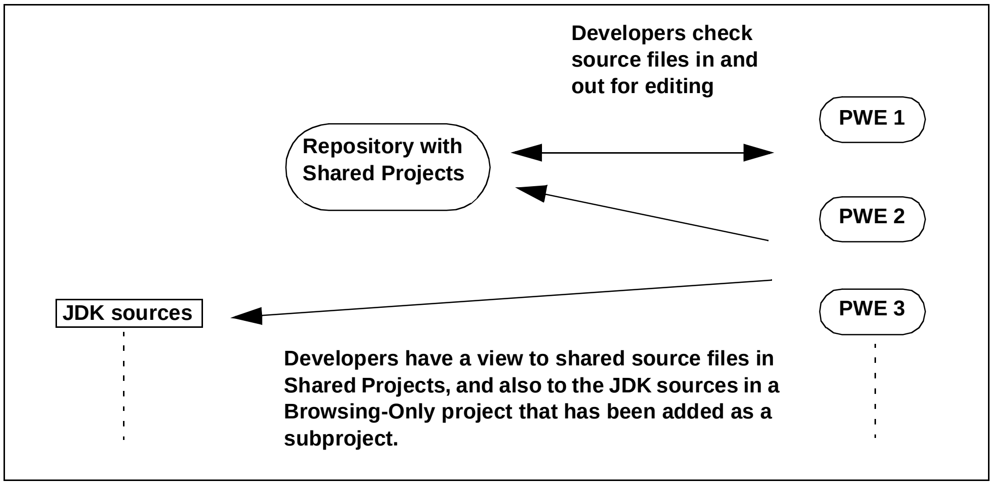
Developers must be able to work in isolation from other team members. They need their own workspaces where they can edit, compile and debug projects without interfering with the work of other team members. Furthermore, they continually need to have access to their software system's most current source code base.

SNiFF+ supports this by allowing each member of a team to work in an isolated Private Working Environment (PWE).

You can go through the entire edit/compile/debug cycle in your PWE. In your PWE, you have a read-only view to the shared source files located in your team's Repository. When you need to modify shared source files, you check out the necessary files from the Repository. When you're satisfied that the changes you've made are error free, you check the modified files back into your Repository. The next time that your colleagues update (synchronize) their PWEs, these changes are incorporated, and the shared source files once again reflect the most current state of your software system.

Typical Java development system

The figure below illustrates a typical Java development system with SNiFF+. Each developer works in his/her Private Working Environment. Shared source code files are checked out of the Repository for modification, then checked back in. Regular updates of the PWEs ensure that the individual developers stay up-to-date.


Developers check source files in and out for editing
PWE 1
Repository with Shared Projects
PWE 2
JDK sources
PWE 3
Developers have a view to shared source files in Shared Projects, and also to the JDK sources in a Browsing-Only project that has been added as a subproject.

The JDK sources (and other libraries) are added to the Shared (Root) Project as Browsing-Only subprojects. These libraries are not version controlled, nor is it necessary to generate Make information for these files.

How you would go about setting up this system is summarized in the following chapter. A step-by-step demonstration using the Project Setup Wizard is included in the hands-on tutorial example.

Java Working Environments and Projects 2

Introduction

The physical organization of your source code directories will affect how easy or complicated you make things for yourself in terms of project setup and, later, update. It is therefore worth looking at your existing source code directory structure before starting on setting up projects. Basically, we recommend that you copy (if necessary) all source directories to one common root directory, if at all possible. This directory should not contain code that is part of named packages, so that the class path points to this directory. This directory will also hold your SNiFF+ Root Project.

After looking at directory structures, the Working Environment and Project setup for Java are summarized. The essential difference in Working Environment organization for Java, as opposed to other programming languages, is that the Java compiler and interpreter can produce inconsistent results if Working Environments are layered. This means that all Private Working Environments should directly access the Repository.

There are a number of Project level settings which you may want/have to set either during Project Setup or afterward. These settings are discussed in some detail in the following chapter.

Source code directory structure

The physical organization of your source code directories affects how you set up, and later update your project system.

Essentially, we distinguish between two types of directory structures:

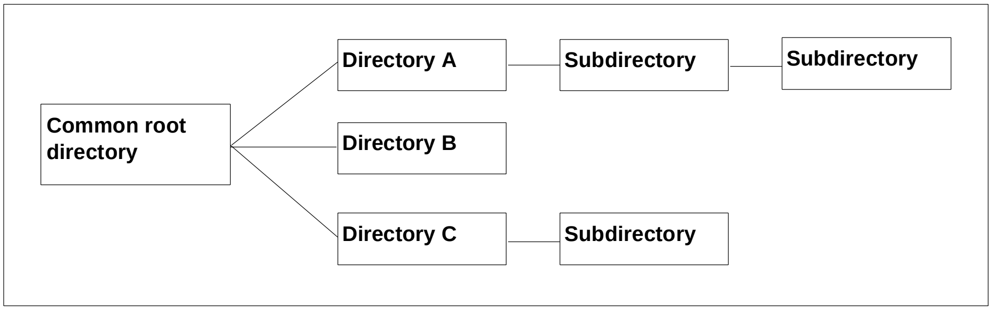
- code packages that have a common root, that is, the structure is *homogenous* or
- code packages that do not have a common root, that is, the structure is *non-homogenous*

The first step is therefore to establish what type of directory/package structure you have.

Homogenous directory structures

We strongly recommend that you use this type of structure if possible.

Directory structures are "homogenous" if all source code packages start from a common root directory.



This common root directory does **not** itself hold source code that is part of any named package, it is simply a common relative starting point, and corresponds to the Java package root. Remember that the Java language specification requires the package root to be one directory level **higher** than the highest-level directory containing named packages.

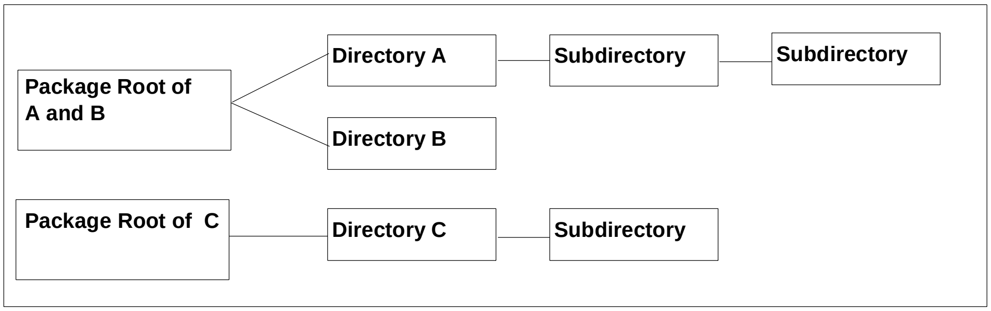
The advantages of this type of project:

- You can create a project for the common root directory and have SNiFF+ generate the subproject tree for all subprojects in one step.
- When you set the build options for the entire project structure, you set the root project as the starting point for all source and generated byte-code packages. SNiFF+ can then generate the correct relative paths for all subprojects in one step.
- To synchronize all projects in a Working Environment, you just have to update the root project, since SNiFF automatically updates all the subprojects.
- All paths are relative.

Non-homogenous project structures

This type of project structure demands more complicated settings than the homogenous project type described above.

Project structures are “non-homogenous” if related code packages start from different root directories.



In the above illustration, A and B are homogenous, that is, the package starts from the same root directory. This assumes that no named package code is contained in the root directory

The structure as a whole is, however, non-homogenous because the package scope of C and subdirectories starts from a different root directory.

This has the following implications:

- Projects A and B can be treated as one homogenous project.
- Project C can be treated as one homogenous project.

  In this structure, you would have to set up two SNiFF+ Projects. All settings have to be repeated, and a one-step update of all projects is no longer possible.

Java Working Environments and Shared Project setup

The following is an abbreviated summary of Working Environment and Project setup. We assume a homogenous directory structure as described above. The hands-on tutorial example describes the procedure using the Project Setup Wizard in detail. For an overview of general project setup procedures, please refer to the *User's Guide*.

Working Environments set up

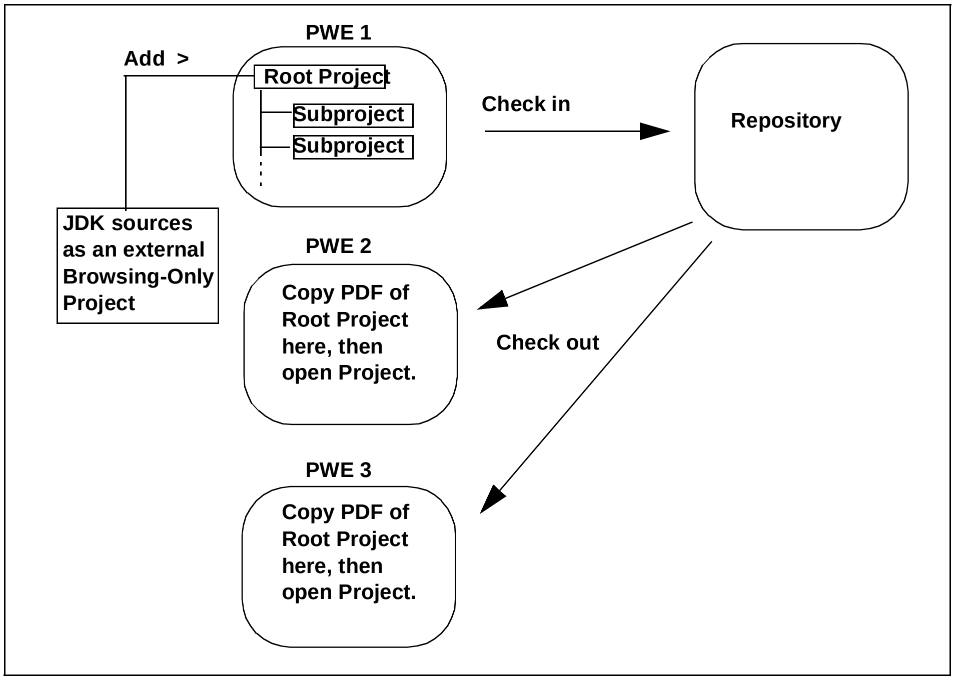
- For Shared Java Projects, always set up Private Working Environments (PWEs) that directly access the Repository.

  This is because equivalent packages that are entered separately in the class path (as is the case in layered Working Environments) are not explicitly examined by the JDK tools for each file. The first entry that is matched is cached and reused by the compiler and interpreter. This may lead to inconsistent results if Working Environments are layered.
- If you already have SNiFF+ Java Projects in layered Working Environments (with an SSWE and SOWE), please continue with Moving existing Working Environments — page 101.

SNiFF+ Java Projects from scratch

- Set up the Shared Project for your root directory in one PWE (see above) and allow SNiFF+ to generate all subprojects. Then, create Browsing-Only Projects for the JDK Sources and any other libraries where source code is available. Add these as subprojects of any Shared Project in the first PWE. See also Adding source code library projects — page 102.
- Make sure you have set all Project Attributes you want/need, either during or after project setup. See also Java Project-Level Settings — page 103.

- Copy the Project Description File (PDF) of the Root Project in the first PWE to the Working Environment root directory of each additional PWE. Then, open the root projects in each PWE.


PWE 1
Add >
Root Project
Subproject
Subproject
Check in
Repository
JDK sources as an external Browsing-Only Project
PWE 2
Copy PDF of Root Project here, then open Project.
Check out
PWE 3
Copy PDF of Root Project here, then open Project.

Moving existing Working Environments

- If you already have SNiFF+ Java Projects in layered Working Environments (with an SSWE and SOWE), move your PWEs so that they directly accesses the Repository (in the Working Environments tool, drag-and-drop them onto the Repository).
- Check out all files to each moved PWE, the SSWE and SOWE will be ignored.

Adding source code library projects

Typically, you would create a Browsing-Only Project for the JDK sources and then add this to your root project. How to go about this is described in the Tutorial.

The procedure is the same for any source code library.

Note that you can add the subproject to **any** project in the tree, symbol information will be available throughout.

See also Upgrading source code library projects (e.g. JDK) — page 132.

Multi-Language Projects

If you are using SNiFF+ Make Support, each directory should contain only Java files. It is not possible to build a project correctly if source files in other programming languages are located in the same directory. To avoid this problem, store these files in separate directories and create separate projects for them.

Java Project-Level Settings 3

Introduction

This chapter concentrates on a description of Java-specific Project Attributes. After a note on the File Types you would generally include in a SNiFF+ Java project, the focus is on build features.

A few of the settings described here are necessary (e.g. location of targets), others are optional, and may or may not be convenient for your particular needs.

The settings can generally be made during project setup and/or in the Project Attributes dialog. Some settings can also be made by directly editing the Project Makefiles (see e.g. Java IDL, JNI and RMI — page 115).

Project Attributes dialog

Many attributes can be set globally during Project Setup. For subsequent changes in global attributes, it is easiest to checkmark all Projects and then choose **Project > Attributes of Checkmarked Projects...** (after making sure the PDFs are writable).

All settings described here are made in the Project Attributes dialog.

File Types

Generally, you would include at least the following file types in a SNiFF+ Java project:

- Java — these are the Java source files (`*.java`)
- HTML — HTML files to embed applets
- Visaj_Project — project files for the integrated Visaj GUI builder.

Project Targets

This section describes each field in the **Build Options > Project Targets > Java** tab.

- If you have opened the Project Attributes dialog for multiple targets, make sure the correct project for the corresponding target is selected in the Project List at the right of the dialog.
- For related technical information, see also Compilation, Compiler Options and Execution — page 109.
- Because of changes in the behavior of the JDK appletviewer between **JDK 1.1.x** and **JDK 1.2,** alternative procedures are described below.

  To avoid confusion, first check which JDK version you are using, then follow the steps under the appropriate heading. Ignore everything under the non-applicable heading.

If you’re using JDK 1.1.x

If you’re using **JDK 1.2**, ignore everything below and continue with If you’re using JDK 1.2 — page 105.

1. In the **Application Class** field, enter the name of the class implementing the `main` method in the following form: `classname`.

> **Note**
>
> - For GUI entries, never use fully qualified class names. Simply enter the class names without qualification or extension.
> - Enter HTML file names and library files with extension.

2. In the **Applet Class(es)** field, enter the name(s) of the classes implementing an `init` method in the following form: `classname1 classname2` ... (use a space as separator). **Note**: These classes will only be generated if you enter a name for the HTML file (see below).
3. In the **HTML File** field, enter the name for the HTML file that embeds the applet(s) in the following form: `filename.html` (with extension). The file will be automatically generated in the selected project directory by SNiFF+.
4. In the **Library (JAR)** field, enter a name (with extension) for a library if you want to build one. The library will be built at the package root to ensure correct package scope.
5. In the **+ JAR Filelist** field, enter a list of files that you want archived e.g., `*.html *.class test.doc` separated by blanks. A specific project directory path can also be entered if you want to archive only a part of the project, e.g., `utilities/*.class`. Note that the path must point into the project.
6. In the **Target class(es)** field enter the name(s) of individual classes you might want to compile, e.g., to use as beans, in the following form: `classname1 classname2` ... (use a space as separator). The classes must be in the currently selected project.

   If you have projects with classes that are not explicitly referenced by targets, enter their names in this field to make sure that these files are also recompiled if they are out of date.
7. Your next step is to check the structural information for compilation.

   Select **Build Options > Build Structure > Java** tab, and continue with Build Structure — page 106.

If you're using JDK 1.2

If you're using **JDK 1.1.x**, ignore everything below and continue with If you're using JDK 1.1.x — page 104.

1. In the **Application Class** field, enter the name of the class implementing the `main` method in the following form: `classname`.

   > **Note**
   >
   > - For GUI entries, never use fully qualified class names. Simply enter the class names without qualification or extension.
   > - Enter HTML file names and library files with extension.

2. Leave the **Applet Class(es)** field blank.

   This feature does not work if you are using JDK 1.2. In this case, enter any applet classes the **Target class(es)** field (see below). See also Upgrading from JDK 1.1.x to JDK 1.2.x — page 132.

3. Leave the **HTML File** field blank.

   This feature does not work if you are using JDK 1.2. See also Upgrading from JDK 1.1.x to JDK 1.2.x — page 132.

4. In the **Library (JAR)** field, enter a name (with extension) for a library if you want to build one. The library will be built at the package root to ensure correct package scope.

5. In the **+ JAR Filelist** field, enter a list of files that you want archived e.g., `*.html *.class test.doc` separated by blanks. A specific project directory path can also be entered if you want archive only a part of the project, e.g., `utilities/*.class`. Note that the path must point into the project.

6. In the **Target class(es)** field enter the name(s) of individual classes you might want to compile, e.g., to use as beans, and also of applet classes (where `init` is implemented) in the following form: `classname1 classname2` ... (use a space as separator). The classes must be in the currently selected project.

   If you have projects with classes that are not explicitly referenced by targets, enter their names in this field to make sure that these files are also recompiled if they are out of date.

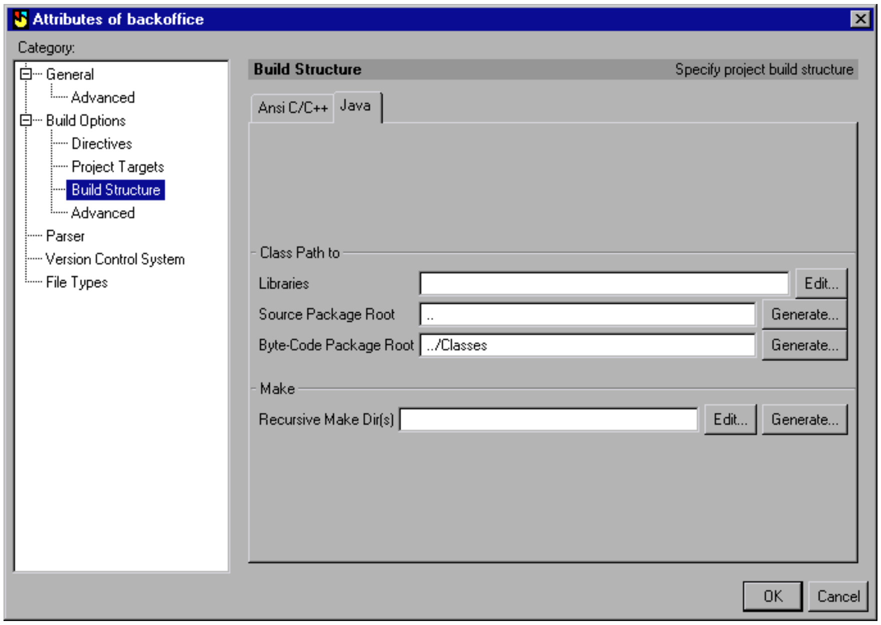
7. Your next step is to check the default structural information for compilation.

   Select **Build Options > Build Structure > Java** tab, and continue with Build Structure — page 106.

Build Structure

This section describes the Java-specific fields in the **Build Structure > Java** tab in some detail. Depending on your project structure and/or whether or not you want to accept the defaults, some of the entries may simply allow customized enhancements. Others will be necessary, again depending on your project structure. Note that you can use environment variables in all fields.

- For more information on the class path concept as used by SNiFF+, please refer also to Class Path — page 113.


Attributes of backoffice

Category:

Build Structure Specify project build structure

Class Path to

Libraries

Source Package Root ..

Byte-Code Package Root ../Classes

Make

Recursive Make Dir(s)

- Class Path to **Libraries**.

  This setting is used to provide a list of absolute paths to library byte-code, zip or jar files used in you project, if there is no source code available for them.

  However, if the source code **is** available for the libraries, as is the case for the JDK library, it is best to create SNiFF+ Projects for them and then add these as subprojects. There is then no need to enter anything in the Classpath field for these projects.

  The advantage of creating and adding subprojects is that you can then also correctly browse inheritance relationships and cross-references etc.

  You can also enter the class path to external source code packages that you have **not** added to your SNiFF+ project as subprojects. This will allow correct compilation, but you will not see any symbol information.

  This applies also to libraries where no sources are available; SNiFF+ will recognize the data types in such files, but not the symbol names.

  If you use **RMI** (see also Java IDL, JNI and RMI — page 115), you have to enter the path to the JDK class files (**JDK 1.1.x**: `classes.zip`; **JDK 1.2**: `rt.jar)` either in this field or in the system CLASSPATH environment variable.

- Class Path to **Source Package Root** field.

  This setting is the class path as you would enter it after `-classpath`. You can specify this relative to the current project or, in the case of an absolute project, as an absolute path.

  The class path to the Source Package Root ends where the package begins. That is, it is always one directory level **higher** than the highest-level directory containing code in named packages. Bear this in mind especially also during Project Setup. Once the project has been set up, this can be set relative to the project, as seen from the package (e.g. `..` or `../..` etc.)

  If you specify the class path to the Source Package Root for a root project, it is generated correctly for all sub-projects. For subsequent modifications, it is easiest to checkmark all projects in the Project Editor and to choose **Project > Attributes of Checkmarked Projects**. Then press the **Generate** button to the right of the field.

- Class Path to **Byte-Code Package Root** field.

  By default, byte-code is generated to the same directories as the source code. If you prefer to keep your source and byte-code separately, enter a root directory (relative or absolute) where you would like your byte-code packages to start, then press the **Generate** button. SNiFF+ will create the specified directory, and recreate the package structure. Byte-code will be generated to this directory when you build your project.

  If you simply enter a name for a directory, it will be created in the current project directory.

Compilation, Compiler Options and Execution 4

This chapter describes the SNiFF+ compilation concept for Java, followed by notes on setting compiler options, and how to use a compiler of your choice (by default SNiFF+ uses the JDK javac complier). The chapter closes with a note on the execution of Java applications in SNiFF+.

Compilation

SNiFF+ does not use the same compilation concept for Java as for other languages. Dependency checking is not done by the Java compiler because it is handled differently on different platforms as well as by different compilers.

Checking dependencies is done entirely by the `sniffjmake` wrapper application, this wrapper calls the Java compiler without the dependency checking flag. All computing is done in the wrapper, and the Java compiler only compiles the files obtained from the wrapper.

If you compile an individual file with the **Target > Make > *FileName*.class** command (or in a Shell with `gmake` <*`Filename`*>), however, only this file is compiled. Referenced class files are only compiled if they do not yet exist.

If you compile an application or applet (i.e., a target), the `sniffjmake` wrapper makes sure that all referenced files are up-to-date, and passes a file list to the compiler for (re)compilation if necessary.

The information needed by the `sniffjmake` wrapper is contained in the following file

```
<ProjectDirectory>/.sniffdir/.sniffjmake
```

Unfortunately, there is a bug in the JDK javac compilers (tested up to version 1.6). If the source file is compiled more than once with the `-depend` flag set, a second byte-code file per anonymous class is generated. For this reason the `-depend` flag is disabled in the SNiFF+ standard distribution. If you have a compiler that does not have this bug, you can enable the full dependency check as described under The -depend option — page 111. Please note that you can only use this option with the Java compiler and not with the `sniffjmake` wrapper since because checking is handled by the wrapper itself.

Because SNiFF+ delegates dependency checking to the `sniffjmake` wrapper, there is a special case to be aware of. If you use classes that are dynamically loaded without being explicitly referenced, specify these classes in the Project Attributes dialog as described under Project Targets — page 103.

Compiler options

In SNiFF+, compiler options and Make information are stored in a set of pre-configured Makefiles in your `<sniff_installation_dir>/make_support` directory. These include a:

- Language Makefile: `general.java.mk`

> **Note**
>
> The `general.java.mk` file works for almost all compilers. If for some reason you want your compiler to do the dependency checking itself, or you cannot switch off dependency checking in your compiler, then use the additional Makefile called `general.javapure.mk`.

- Platform Makefile e.g.

  `i386-unknown-win32.mk` or

  `sparc-sun-solaris4.1.mk`
- A Project Makefile (named `Makefile`) is generated by SNiFF+ for each project and stored in the corresponding project directories.

Settings at platform level

To set compiler options at the platform level, you need to edit your Platform Makefile in the `<sniff_installation_dir>/make_support` directory. To get to the information for Java, look for the following text block in your `<platform>.mk` file

```
#
# java compiler
#
JAVAC = $(SNIFF_DIR)/BIN/sniffjavac.exe
JAVA_DEPEND_FLAG =
JFLAGS = -g
JAVA_INTERPRETER = java
JAVA_APPLET_VIEWER = appletviewer
```

The -depend option

Note that you can only use this option with the Java compiler and not with the `sniffjmake` wrapper since dependency checking is handled by the wrapper itself.

The following line in your `<platform>.mk` file

```
JAVA_DEPEND_FLAG =
```

is where you can set the `-depend` option.

The `-depend` option causes automatic recompilation of all class files on which source files recursively depend. Without this option (default), only out-of-date files that are directly depended on will be recompiled. Missing or out-of-date files only depended on by already up-to-date class files are not (re-)compiled.

To set the option, change the above line to read:

```
JAVA_DEPEND_FLAG = -depend
```

Note that, if you set this option, the JDK Java compiler will report errors for already compiled anonymous classes - and create new (additional) ones. Therefore, if you use this option, be sure to delete the already compiled anonymous classes before each new build. Also, be aware that your builds will be slower.

Debug information: the -g option

The `-g` option enables generation of debugging tables (default) and is set in the line:

```
JFLAGS = -g
```

If you want a release version of your software (no debug information), change the line to:

```
JFLAGS =
```

Other options

You can also set other options (see your compiler documentation) in the line

```
JFLAGS =
```

by leaving a space between each option you enter after the “=” sign.

Using a compiler other than the JDK javac compiler

The following line in your `<platform>.mk` file

```
JAVAC = javac
```

specifies the JDK javac compiler. Change this line to conform to:

```
JAVAC = <path_to_your_compiler_dir>/yourCompilerName
```

Setting options at project level

Note that compiler options set at project level are valid only for the project and override those set at the platform level. Overriding platform-specific settings is **not** recommended, as Project Makefiles should remain platform-independent.

To add options for a particular project, look in the project's `Makefile` for the line starting with

```
#OTHER_JFLAGS =
```

Enter the required options (separated by a space) after the "=" sign and remove the "#" at the beginning of the line.

Execution

Java byte-code is not compiled into executable files as is the case for most other programming languages supported by SNiFF+. For this reason, two files per target are generated in the target directory, namely `<TargetName>.sh` and `<TargetName>.env`.

These files are generated when you execute the **Target > Make *TargetName*** command and are for internal use only.

Class Path 5

This chapter offers a description of central concepts relating to how Java projects are handled in SNiFF+.

The class path and package concepts are used by SNIFF+ in the same way as they are used by the JDK tools. Excerpts from the original JDK documentation are included below to clarify this.

In the SNiFF+ Project Attributes dialog and the Project Setup Wizard, there are fields where the class path to the Source Package Root and to the Byte-code Package Root can be entered. The semantics are the same as in the class path as used by the JDK tools.

The class path in the Project Attributes can be entered as a relative path — as seen from the package (e.g. `..` or `../..` etc.) once the projects have been set up.

For more Java language-specific information, please refer also to the Sun Java documentation, available at:

http://java.sun.com/

Class path

JDK tools use the class path concept to find source and byte-code files during compilation, execution and debugging etc. The same concept is used by the SNiFF+ Java parser.

SNiFF+ generates and sets this class path transparently for you, so that you do not have to worry about this when you work on several parallel projects. You can specify a different class path for each (sub)project. All the settings are therefore project-specific and can be set in the Project Attributes dialog.

- **Note**: The class path always ends one directory level **higher** than the highest-level directory containing named package code.

The following excerpts from the original JDK documentation illustrate the class path concept as it is used by the JDK tools. SNiFF+ uses this concept in exactly the way.

Synopsis [excerpt from original JDK documentation]

"The class path can be set using either the `-classpath` option with the a JDK tool (the preferred method) or by setting the `CLASSPATH` environment variable.

```
C:> jdkTool -classpath path1;path2...
C:> set CLASSPATH=path1;path2...
```

Each class path ends with a file name or directory depending on what you are setting the class path to:

- For a `.zip` or `.jar` file that contains `.class` files, the path ends with the name of the `.zip` or `.jar` file.

- For `.class` files in an unnamed package, the path ends with the directory that contains the `.class` files.
- For `.class` files in a named package, the path ends with the directory that contains the "root" package (the first package in the full package name).

  ..."

  (To restate the point, the class path always ends one directory level **higher** than the highest-level directory containing named package code.)

Example on class path and package names [excerpt from original JDK documentation]

"Java classes are organized into packages which are mapped to directories in the file system. But, unlike the file system, whenever you specify a package name, you specify the *whole package name -- never part of it.* For example, the package name for `java.awt.Button` is *always* specified as `java.awt`."
For example, suppose you want the Java runtime to find a class named `Cool.class` in the package `utility.myapp`.
If the path to that directory is

```
C:\java\MyClasses\utility\myapp
```

you would set the class path so that it contains

```
C:\java\MyClasses
```

To run that app, you could use the following JVM command:

```
C:>  java -classpath C:\java\MyClasses utility.myapp.Cool
```

When the app runs, the JVM uses the class path settings to find any other classes defined in the `utility.myapp` package that are used by the `Cool` class.
Note that the entire package name is specified in the command. It is not possible, for example, to set the class path so it contains `C:\java\MyClasses\utility` and use the command, `java myapp.Cool`. The class would not be found."

- The above excerpts illustrate the class path/package concepts as used by the JDK tools. SNiFF+ uses these concepts in exactly the same way.

Troubleshooting

Bugs in the class path as used by SNiFF+ can be difficult to identify. The relevant information about what SNiFF+ generates is in the following file

```
ProjectDirectory/.sniffdir/macros.incl
```

(search for `SNIFF_JAVA_CLASSPATH`)

Java IDL, JNI and RMI 6

This chapter outlines the necessary steps for generating

- Java files from IDL files
- Java native interfaces (JNI)
- remote method invocations (RMI)

Java IDL Generation

To generate Java files from IDL files, make sure your compiler and the appropriate flags are set in `<sniff_installation_dir>/make_support/general.java.mk` under the `IDL` section.

To generate Java files from IDL files, enter `idltojava` followed by by the input file names, or enter `gmake <filename>.idl`.

JNI (Java Native Interface) Generation

The Java Native Interface generator can be called

- explicitly on the command line by entering `gmake jni`.
- automatically with each Java compiler make command.

  If you want automatic JNI support, change the following line in the `<sniff_installation_dir>/make_support/general.java.mk` file:

  ```
  JNI_SUPPORT = 0
  ```

  to

  ```
  JNI_SUPPORT = 1
  ```

  Caution

  If the word "Native" appears in Javadoc comments (`**/...*/`), the file containing this comment is also native compiled and an empty file `<classname>.h` is created.

- To delete JNI generated files, enter `gmake clean` on the command line, or choose **Target > Make > clean** if you are using SNiFF+ Make Support.

RMI (Remote Method Invocation) Generation

This is a client/server stub/skeleton generator.

CLASSPATH

For `rmic` support, set the system CLASSPATH environment variable to point to:

- **JDK 1.1.x**: `classes.zip` (typically in `<JDK_directory>/lib>`
- **JDK 1.2**: `rt.jar` (typically in `<JDK_directory>/jre/lib>`

Makefiles

To define remote methods, open your project Makefiles

```
<project_directory>/Makefile
```

and add the names of the files containing remote methods to the `SNiFF_JAVA_RMI_FILES` macro (use a space as separator).

RMI calls

- RMI can be called from the command line by entering `gmake rmi`.
- You can automatically call RMI with with each Java compiler make command.

  In `<sniff_installation_dir>/make_support/general.java.mk`

  change the following line

  ```
  RMI_SUPPORT = 0
  ```

  to

  ```
  RMI_SUPPORT = 1
  ```

- Note that, if you use the `gmake clean` command (or choose **Target > Make > clean**, using SNiFF+ Make Support) RMI generated files will also be deleted.

Symbol Information and Automated Updates 7

This chapter simply informs you that SNiFF+ Java projects should always be opened with symbol information, and consequently the default in the script for unattended updates also needs to be edited accordingly. A note on how certain symbols are represented in SNiFF+ for Java is also included.

Opening Projects with Symbols

SNiFF+'s Make Support for Java uses symbol information. This is because more than one class can be defined per file, whereas one byte-code file is generated for each defined class. So be sure to always open Java projects with the **With Symbols** check box in the enabled (default).

Representation of Java Symbols in SNiFF+

In SNiFF+, a static initializer method is called `<static_init>` and is treated like a class method.
An instance initializer is called `<init>` and is treated like a final instance method.

Unattended Updates

SNiFF+'s Make Support for Java uses symbol information (see above). For unattended unattended Working Environment updates with Java projects, open the file:
`<your_sniff_installation_dir>/ws_support/updateWS.sh`

- Change the following line

  ```
  echo open_project \"$2\" WITHOUT_SYMBOLS NO_CACHE
  ```

  to:

  ```
  echo open_project \"$2\" NO_CACHE
  ```

- Change the following line:

  ```
  $MAKE -i symbolic_link_to_dependencies_file
  ```

  to:

  ```
  #$MAKE -i symbolic_link_to_dependencies_file
  ```

- For more information about unattended updates, please refer to the *User's Guide*.

SniffJdb Debugger 8

The SNiFF+ debugger for Java, SniffJdb, implements 3 user interface tools

- the Debugger Command Line Shell
- the Variable Viewer
- the Threads Viewer

Debug commands can be entered at the command line, from the button bar in the source editor (appears only when the debugger is started), and from the graphical tools' menus.

To use the SniffJdb debugger ...

To use the SniffJdb you need to

1. Set SniffJdb as your preferred debugger in the SNiFF+ Preferences. This is described under Setting the debugger in the Preferences — page 73.
2. Make sure that debugging tables are generated by the compiler (`-g` option). SNiFF+ is delivered with this option set in the Platform Makefiles. How to set the debugging information option is described under Debug information: the -g option — page 111.
3. If you start debugging from the Project Editor, make sure that the project containing the relevant target is highlighted in the Project Tree.

The debugger command line

Commands include the jdb commands, enhanced by a subset of the gdb commands.
The quickest way to get an overview of the debugger command line commands is to

- type `help` at the prompt `(sniffjdb)` in the Debugger Command Line Shell.

  All available commands are then listed in alphabetical order together with syntax and a short description.

Functionality differences between sniffjdb and jdb

Added functionality

- `vt` - starts the graphical Threads Viewer.
- `print [*] <id> | this` - the print commands open the graphical Variable Viewer and show the values of specified (member) fields. Information is not updated automatically at each breakpoint. To update, choose the Variable Viewer's **View > Update**.

- `display, undisplay` - these accept the same syntax as the `print` commands. The difference is that the information is automatically updated in the graphical Variable Viewer every time a breakpoint is reached.
- `step out` - executes `next` until a breakpoint in another method is reached. Does not trace execution.
- `finish` - executes `next` until a breakpoint in another method is reached. Traces execution.
- `show` - prints SniffJdb version number
- `about` - opens an About dialog
- extended `break/clear` syntax (see below)

Removed functionality

- `use`, `list` - these commands only make sense for a pure command line debugger.

Command Reference

Display

Command	Description
`print *<id>`	Print information about object pointed to.
`locals`	Print values of parameter and local variables of the current stack frame
`memory`	Report memory usage.
`print <id>`	Print value of specified variable.
`print this`	Print local variables

Automatic display

Command	Description
`display`	Lists all display expressions.
`display <id>`	Display value of the specified object.
`display *<id>`	Display the object pointed to.
`undisplay [*]<id>`	Stop displaying the specified object.

Breakpoints

Command	Description

`break [at] <class id>:<line>`
`stop [at] class id:line`
	Break and stop can be used as synonyms.

`break [at] <file name>:<line>`
`stop [at] <file name>:<line>`
	Set a breakpoint in the specified file at the specified line. This command works only if the public class of that file is already loaded (see load command).

`break [in] <class id>:<method name>`
`stop [in] <class id>:<method name>`
	Set a breakpoint at the specified line in the file where the specified class is defined.

`clear [at] <class id>:<line>`	Remove a breakpoint at the specified line in the specified file.
`clear [at] <file name>:<line>`	Remove a breakpoint at the specified line in the file where the specified class is defined.
`clear [in] <class id>:<method name>`	Remove a breakpoint at the first line of the specified method.

Execution Control

Command	Description
`cont`	Continue execution from breakpoint.
`exit`	Terminate execution.
`finish`	Execute next until a breakpoint in another method is reached. Do not trace execution.
`next`	Execute the next statement, step over methods.

`run [<class>]`
`[args]`
	Start execution of a loaded Java class.

`step`	Execute next statement, step into methods.
`step out`	Execute next until a breakpoint in another method is reached. Trace execution.
`!!`	Repeat last command

Exceptions

Command	Description
`catch <exception class name>`	Break when the specified exception is thrown.
`ignore <excception class name>`	Do not break when the specified exception is thrown.

Other

Command	Description
`gc`	Trigger garbage collection.
`help`	List commands. For space reasons this listing does not contain all possible ways to express the same.
`itrace [ | on | off ]`	Toggle method trace mode. Does not currently work with `sun.tools.debug`.
`load <class name>`	Load Java class.
`trace [ | on | off ]`	Toggle instruction trace mode. Does not currently work with `sun.tools.debug`.
`reload`	Reload the debugged class. Makes sense after recompilation of source. Keeps breakpoints.
`show [version]`	Print version information.

Stack

Command	Description

`bt [ thread id | all ]`
`backtrace [ thread id | all ]`
	Dump thread stack

`where [ thread id | all ]`	Dump thread stack.
`down [n frames]`	Move down thread stack.
`frame <number>`	Move to the specified frame of thread stack.
`up [n frames]`	Move up thread stack.

Remote Debugging

Command	Description
`connect <hostname> <password>`	Connect debugger to a remote process. You get the password on starting the application with the debug option.

Symbolic information

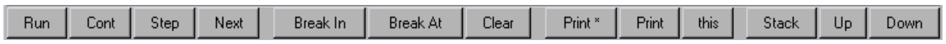
Command	Description
`classes`	List currently known classes.
`methods <class id>`	List a class's methods.

Threads

Command	Description
`vt`	Start Threads Viewer
`viewthreads`	Start Threads Viewer
`kill <thread(group)`	Kill a thread or threadgroup.
`resume [thread id(s)]`	Resume threads (default: all).
`suspend [thread id(s)]`	Suspend threads (default: all).
`thread <thread id>`	Set default thread.
`threadgroups`	Suspend threads (default: all).
`threadgroup <name>`	Set current threadgroup.
`threadgs`	List threads.

Debugging from the Source Editor

In debug mode, files loaded in the Source Editor are read-only and a button bar is added to the tool.



- When you press one of these buttons, the corresponding command is executed in the Debugger Command Line Shell.
- Object IDs are specified by highlighting the object names in the Source Editor.

Run	Runs the application being debugged from scratch.
Cont	Continues interrupted execution.
Step	Single-steps into the next function/method.
Next	Single-steps over the next function/method.
Break In	Sets a break point at the first execution line of a selected method.
Break At	Sets a breakpoint at the current cursor position.
Clear	Clears the breakpoint in the current line. The cursor must be positioned to a line with a breakpoint.
Print *	Prints the value pointed to by the current selection. The selection must evaluate to a valid pointer.
Print	Prints the value of the current selection. The selection must evaluate to a valid variable.
this	Prints the local variables of the current object.
Stack	Opens a Stack window and displays the current call stack
Up	Goes one stack frame up in the call hierarchy. A reusable Source Editor is automatically positioned at the source location of the new stack frame.
Down	Goes one stack frame down in the call hierarchy. A reusable Source Editor is automatically positioned at the source location of the new stack frame.

- The **Print ***, **Print**, and **this** buttons show highlighted objects in the Variable Viewer.

The Variable Viewer

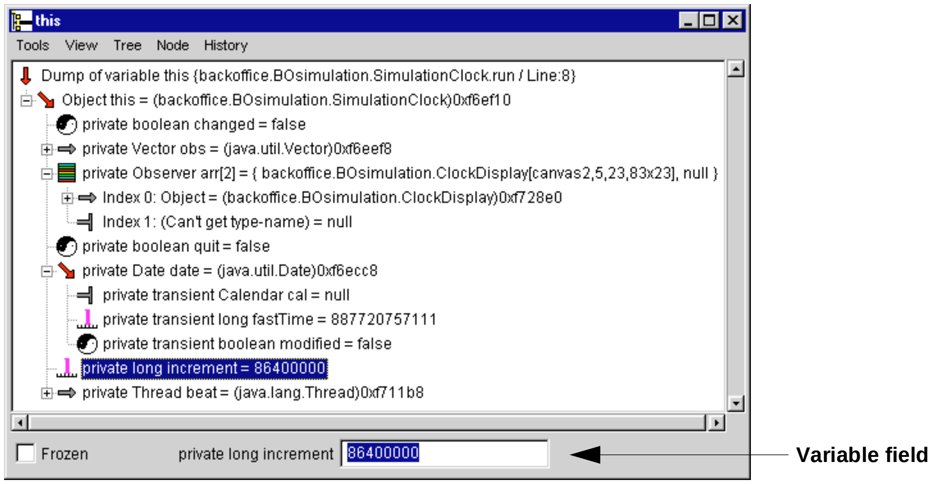
The Variable Viewer shows a specified variable, class or package in a tree.

To see variables or objects in the Variable Viewer at a breakpoint

- in the Source Editor (debug mode), highlight an object and press the **Print *** or the **Print** button or
- in the Debugger Command Line Shell, type `print <id>` or type `print* <id>` or
- in the Debugger Command Line Shell, type `display <id>` to see variables displayed and automatically updated at each breakpoint.

To see all the local variables in the Variable Viewer at a breakpoint

- in the Source Editor (debug mode), press the **this** button or
- in the Debugger Command Line Shell, type `this`



```
Dump of variable this {backoffice.BOsimulation.SimulationClock.run / Line:8}
Object this = (backoffice.BOsimulation.SimulationClock)0xf6ef10
    private boolean changed = false
    private Vector obs = (java.util.Vector)0xf6eef8
    private Observer arr[2] = { backoffice.BOsimulation.ClockDisplay[canvas2,5,23,83x23], null }
        Index 0: Object = (backoffice.BOsimulation.ClockDisplay)0xf728e0
        Index 1: (Can't get type-name) = null
    private boolean quit = false
    private Date date = (java.util.Date)0xf6ecc8
        private transient Calendar cal = null
        private transient long fastTime = 887720757111
        private transient boolean modified = false
    private long increment = 86400000
    private Thread beat = (java.lang.Thread)0xf711b8
```

private long increment 86400000

- By default, SNiFF+ reuses tools to prevent screen cluttering. To open another Variable Viewer, you can freeze the already open tool by enabling **Frozen**.
- A shift-click on a tree-node zooms in on the selected variable.
- A click on a simple data-type makes this entry editable in the **Variable** field (this does not work with array elements, and variables which are not in a record, a class, or an interface). Editing can be cancelled by hitting `<Esc>`, or by clicking into another node.

The Variable Viewer menu

Tools

> Close Tool: Closes this tool
> Duplicate Tool: Sets this Variable Viewer to frozen and opens a new Variable Viewer
> Threadsview: Opens a Threads Viewer
> About SNiFF+ Jdb: Opens an About dialog

View

> Update: Updates the contents of the selected variable in the current view
> Show typed names on/off: Shows the selected variable with or without modifiers and types. If no node is selected, the base node is shown
> List all classes: Shows all loaded classes
> Show source: Shows the source of the variable (the position of the breakpoint when the Variable Viewer was opened)

Tree

> Expand one level: Expands all currently visible unexpanded nodes one level
> Collapse all: Collapses all expanded nodes

Node

> Show father: Shows the father of the currently shown variable (if it exists)
> Expand: Expands the selected node
> Collapse: Collapses the selected node
> Array size 5 - 50: Defines how arrays are subdivided on being opened

History

> 1-10: The last 10 variables shown by this Variable Viewer (the last entry is the currently shown variable)

The Variable Viewer icons

Simple data types

Integer types: s - short, i - int, l long
Real numbers: f - float, d - double
Special: ◐ - boolean, b - byte, ch - char

Expandible nodes (first icon: collapsed / second icon: expanded)

⇨ , ➘ - record
▤ , ▤ - array (grayscale, color)
▤ , ▤ - part of an array (grayscale, color)
C , C - class (black, red)
I , I - interface (black, green)
⇩ , ⇩ - base node with current path (gray, red)

Non-expandible nodes

⊣ - null pointer (in record, array, class or interface)
M - method, shown in classes
○ - instance field, shown in classes, without data
ϟ - error node or null pointer (local or static)

Other information

✓ , ✓ , ... icons are checkmarked if already shown at a higher level
● , ● ,... icons are marked with a red dot after a forced update if the value has changed.

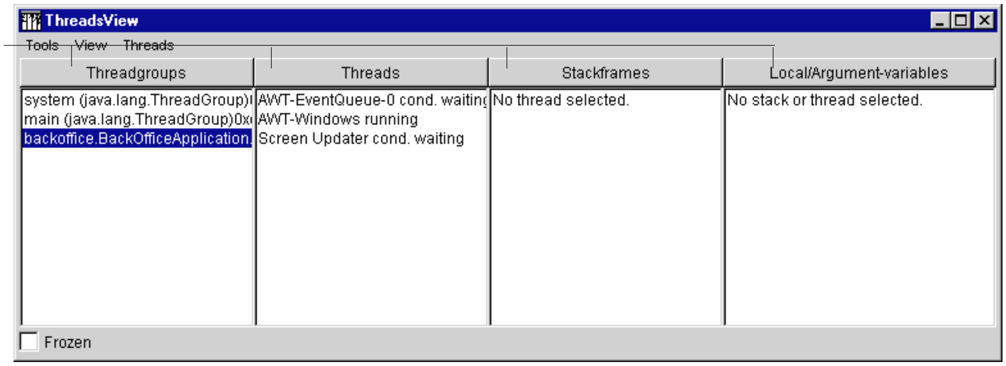
The Threads Viewer

The Threads Viewer shows the existing threadgroups, threads, stackframes and local variables.

To open the Threads Viewer:

- In the Debugger Command Line Shell, type `vt` or `viewthreads`
- In the Variable Viewer, choose **Tools > Threadsview**

Press these buttons to update the columns



system (java.lang.ThreadGroup)
main (java.lang.ThreadGroup)0x
backoffice.BackOfficeApplication.

AWT-EventQueue-0 cond. waiting
AWT-Windows running
Screen Updater cond. waiting

No thread selected.

No stack or thread selected.

The Threads Viewer consists of four lists:
Threadgroups: All existing threadgroups.
Threads: The threads in the highlighted threadgroup(s).
Stackframes: The stackframes from the highlighted thread.
Local/Argument-variables: Local variables in the highlighted stackframe. If no stackframe is selected, the variables in the highlighted thread (i.e. from the top-most stackframe) are listed.

- The button above each list forces an update of the corresponding list.
- By default, SNiFF+ reuses tools to prevent screen cluttering. To open a further Threads Viewer, you can freeze the already open tool by enabling **Frozen**.
- If the selection in a list changes, the lists to the right are updated.
- A double-click on an item in the Threads or the Stackframes list opens the Source Editor at the corresponding position in the source code (if available).
- A double-click on an item in the Local/Argument-variables list opens the Source Editor at the corresponding position in the source code (if available).

The Threads Viewer menu

Tools

> Duplicate Tool: Sets this Threads Viewer to frozen and opens a new Threads Viewer
> Close Tool: Closes this tool
> Variableview: Opens a Variable Viewer to show the selected variable
> About SNiFF+ Jdb...: Opens an About dialog

View

> Update: Updates all lists
> Auto-Update off: Turns off the automatic update of the lists
> Auto-Update (slow): Turns on the slow automatic update of the lists
> Auto-Update (fast): Turns on the fast automatic update of the lists
> Show variable: Opens a Variable Viewer with the selected variable
> Show source: Shows the source of the selected stackframe or thread

Threads

> Stop threadgroup(s): Stops the selected threadgroup(s)
> Suspend thread: Suspends the selected thread
> Resume thread: Resumes the selected thread
> Stop thread: Stops the selected thread

Upgrade Issues 9

This chapter covers two different kinds of upgrading issues.

- The first section covers Upgrading SNiFF+ 2.x to 3.x — page 131.
- The second section covers Upgrading source code library projects (e.g. JDK) — page 132 (Browsing-Only Projects) that have been added to SNiFF+ Shared Projects. Upgrading JDK versions is discussed as an example, but only as a general case as it applies to any source code libraries.

  Apart from a note on the changes in applet viewing, added because SNiFF+ supports both JDK 1.1.x and JDK 1.2, you are referred to the JDK documentation for details about JDK version changes.
- If your Java Projects are at present in layered Working Environments (PWE - SOWE - SSWE - RW), this can lead to problems. Please refer to Working Environments set up — page 100 for more information.

Upgrading SNiFF+ 2.x to 3.x

In Java projects set up in versions of SNiFF+ prior to 3.0, source code packages were relative to the Working Environment root. As of SNiFF+ 3.0, source code packages are relative to the project root directory by default. To use existing Java projects with SNiFF+ 3.x, change the settings accordingly in the Project Attributes.

Upgrading to SNiFF+ 3.1

To use your existing SNiFF+ Java projects with SNiFF+ 3.1 copy

```
<sniff_installation_dir>/config/template.java.Makefile
```

to each project directory and rename it to `Makefile`. That is, replace the existing project Makefiles. Then, update Makefiles.

Note

If you have made changes in your old Makefiles, merge these changes into the new Makefiles.

Upgrading source code library projects (e.g. JDK)

Here, the JDK sources are used as an example, but the procedure applies to any source code library project that you have added to your own SNiFF+ Java projects.

- Upgrade your JDK version.
- Then create a SNiFF+ Browsing-Only Project for it.
- Add this to your own SNiFF+ Java Projects (replacing the old JDK project).
- Delete all old class files (**Target > Make > clean**).
- Rebuild all targets (**Target > Make > all**).

Upgrading from JDK 1.1.x to JDK 1.2.x

This section offers a few notes regarding upgrading of the JDK. For more information, please refer to the JDK documentation.

Applet viewing with JDK 1.2

The JDK appletviewer behaviour has changed in JDK 1.2. To quote the JDK documentation: "In JDK 1.2, `appletviewer` ignores your CLASSPATH environment setting (which it did not ignore in 1.1)."

Because of this, the SNiFF+ feature for automatically generating an HTML file to embed and view applets no longer works if you are using JDK 1.2.

If you are using **JDK 1.2**, write an HTML file to embed the applet and save this in the directory that *contains* the "root" package (the first package in the full package name). That is, one directory level higher than the top-most directory containing named package code.

Command line length option (-c)

The `-c` (command line length) option for passing files to the `javac` compiler is set in

```
<sniff_install_dir>/make_support/general.java.mk
```

at 1280 bytes by default.

If the CLASSPATH is longer than the set length, compilation problems can occur.

For **JDK 1.1.x** tools (compiler), this option **must** be set, and can be increased if higher values are supported by the shell used.

For JDK tools as of **JDK 1.2** this option is not necessary, and is therefore best removed. Problems caused by CLASSPATHS that are too long are thus avoided, and compilation is faster without this option, because a filelist (without repeats) is passed to the compiler.

Visaj GUI Builder Integration 10

The Visaj integration with SNiFF+ allows you to combine Visaj's graphical GUI design features with SNiFF+'s source code engineering functionality. The symbol information and inheritance relationships of Visaj generated source code can therefore be directly browsed and edited in SNiFF+.

Installation

Note

The Visaj Resource Bundle Editor, Image Editor and Project Window are currently not part of the SNiFF+ Visaj integration.

Note

Once Visaj is selected as part of the SNiFF+ installation, it is automatically installed on your computer. You need not install Visaj separately.

Requirements

- Integrating SNiFF+ with Visaj is only possible since SNiFF+ 3.1.
- You need JDK 1.1.3 or higher installed on your computer.
- JDK must be in your path.

Note

You can download the JDK from
http://java.sun.com

For JDK 1.2 users

We suggest that you do the following to improve overall performance.
In the `SNiFF_DIR/bin/runvisaj.sh script`, change:

```
javaw com.pacist.visaj.ClassAppLoader $SNIFF_DIR/Visaj
TakeFivePlugin "$1" "$2"
```

to

```
javaw -Djava2d.font.usePlatformFont=true
com.pacist.visaj.ClassAppLoader $SNIFF_DIR/Visaj
TakeFivePlugin "$1" "$2"
```

Selecting Visaj as part of your SNiFF+ Installation

- **On Windows**, Visaj is part of the Java package. To integrate SNiFF+ and Visaj, make sure that you select the **Java package** as part of your SNiFF+ installation.
- **On Unix**, to integrate SNiFF+ and Visaj, make sure that you select the Visaj package as part of your SNiFF+ installation.
- For more information on how to install SNiFF+, please refer to the SNiFF+ Installation Guide for Windows/Unix.

Adding Visaj projects to a SNiFF+ project

Adding a new Visaj project

In the SNiFF+ Project Editor:

1. Make sure that the relevant SNiFF+ project is highlighted.
2. From the menu, choose either

   Project > Add Visaj Project to *Projectname*, or choose **Tools > Visaj**
3. In the dialog that appears, enter a name for the new project and press **Ok**.
4. Choose **Project > Save *Projectname*** to save the modified project.

Adding an existing Visaj project

First, copy your Visaj project to your SNiFF+ project directory.
Then, in the SNiFF+ Project Editor:

1. Make sure that the relevant SNiFF+ project is selected in the Project Tree.
2. From the menu, choose **Project > Add/Remove Files to/from *Projectname***.
3. In the Add/Remove Files dialog that appears, select the Visaj project file, press the **Add** button, then press **Ok**.
4. Choose **Project -> Save *Projectname*** to save the modified project.

Working with SNiFF+ in Visaj

Loading the Visaj project into the Visaj Class Editor

In any SNiFF+ tool, choose **Tools > Visage**
OR
In the SNiFF+ Project Editor, double-click on the Visaj project file

Java code generation

Java code is automatically generated and stored in your SNiFF+ project directory when you

- save a file in the Visaj Class Editor
- execute commands in the SNiFF+ menu of the Visaj Class Editor
- modify the properties of a class in the Visaj Class editor

Online documentation

The Visaj Class Editor's **Help** menu provides more information on SNiFF+J Visaj integration features, as well as on using Visaj.

Colophon

This manual was produced with FrameMaker.

sniff \’snif\ *vb* -ED/-ING/-S
[ME *sniffen*; prob. akin to ME *snivelen* to snivel]
vt (14c)
3: to recognize or detect by or as if by smelling
<German shepherd dogs are parachuted in the Austrian Alps to *sniff* out survivors of avalanches — P.T.White>

Webster’s Unabridged Third New International Dictionary