

Editing Java elements

Contents

1. [Opening a Java editor](#)
2. [Using quick views](#)
3. [Adding new methods](#)
4. [Using content assist](#)
5. [Identifying problems in your code](#)
6. [Using code templates](#)
7. [Organizing import statements](#)
8. [Using the local history](#)
9. [Extracting a new method](#)

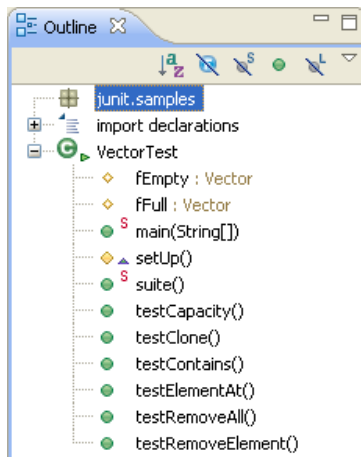
1. Opening a Java editor

In this section, you will learn how to open an editor for Java files. You will also learn about some of the basic Java editor features.

1. Expand the package `junit.samples` and select the file `VectorTest.java`. You can open `VectorTest.java` in the Java editor by double clicking on it. In general you can open a Java editor for Java files, types, methods and fields by simply double clicking on them. For example, to open the editor directly on the method `testClone` defined in `VectorTest.java`, expand the file in the **Package Explorer** and double click on the method.
2. Notice the syntax highlighting. Different kinds of elements in the Java source are rendered in unique colors. Examples of Java source elements that are rendered differently are:
 - Regular comments 绿色
 - Javadoc comments 蓝色
 - Keywords 红色
 - Strings 蓝色

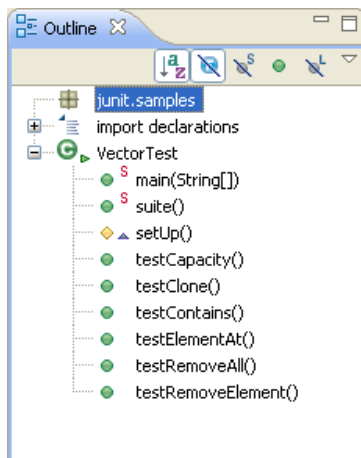


3. Look at the **Outline** view. It displays an outline of the Java file including the **package declaration, import declarations, fields, types and methods**. The Outline view uses icons to annotate Java elements. For example, icons indicate whether a Java element is static (s), abstract (A), or final (F). Different icons show you whether a method overrides a method from a base class (▲) or when it implements a method from an interface (▲).

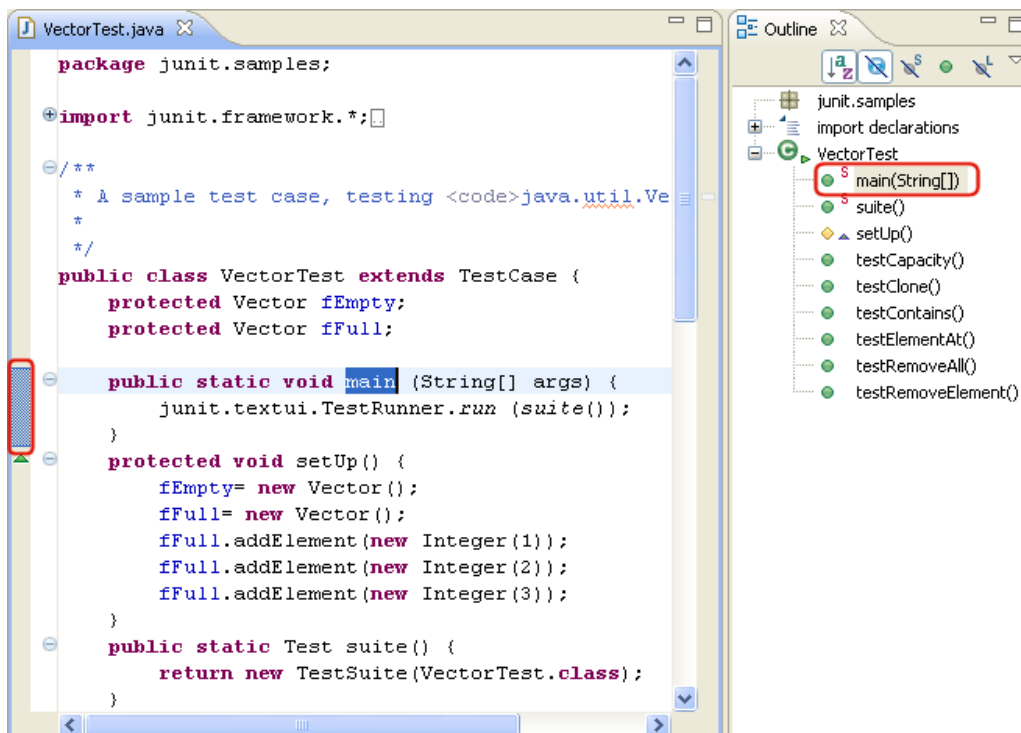


public	绿色
protected	黄色
default	蓝色
private	红色

4. Toggle the **Hide Fields**, **Hide Static Members**, and **Hide Non-Public Members** buttons in the Outline view toolbar to filter the view's display. Before going to the next step make sure that the **Hide Non-Public Members** and **Hide Static Fields and Methods** buttons are not pressed.



5. In the Outline view, select different elements and note that they are again displayed in a whole file view in the editor. The Outline view selection now contains a range indicator on the vertical ruler on the left border of the Java editor that indicates the range of the selected element.



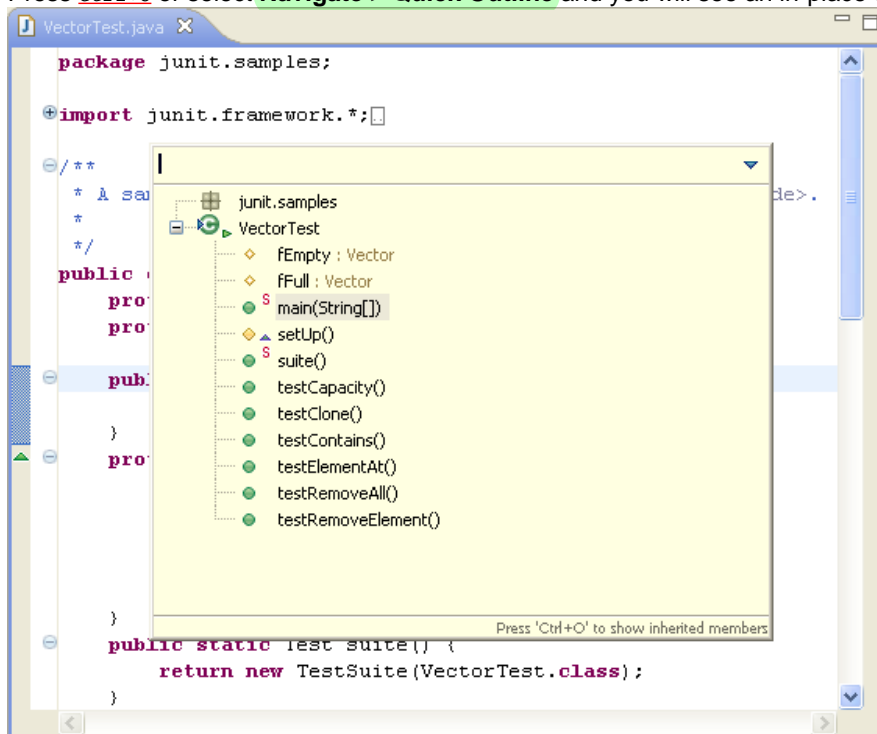
2. Using quick views


In this section, you will be introduced to the **quick outline view**. Quick views are in-place views which are shown on top of the editor area and can easily be controlled using the keyboard. A second quick view will be introduced in the [Type Hierarchy section](#).

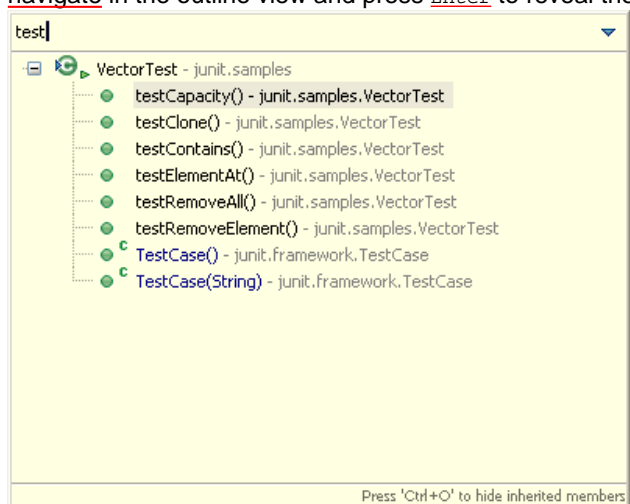
Quick Outline

To use the quick outline view in the Java editor:

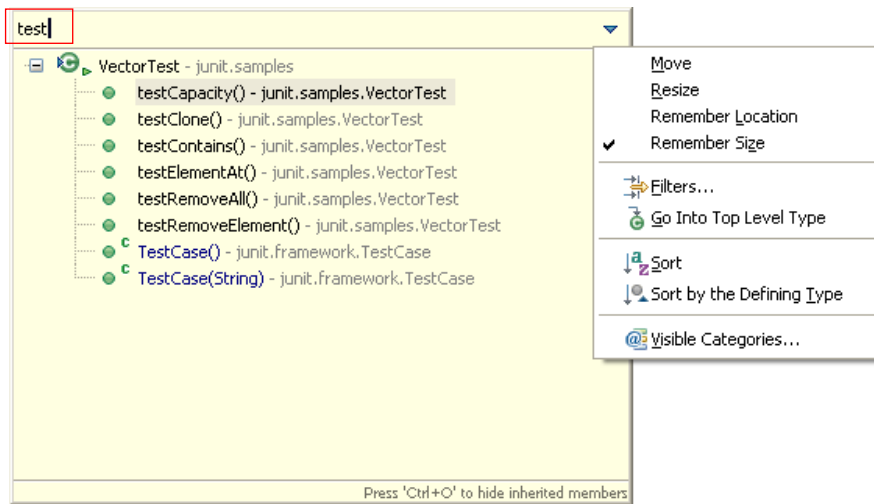
1. Open *junit.samples.VectorTest.java* file in the Java editor if you do not already have it open.
2. Press **Ctrl+O** or select **Navigate > Quick Outline** and you will see an in-place outline of the current source file.



3. Press **Ctrl+O** a second time and all inherited fields, types and methods are shown as well (for the types marked with  on the left). Inherited members are shown in blue.
4. Start typing while the quick outline view is shown to filter the list of displayed elements. Further, use the arrow keys to navigate in the outline view and press **Enter** to reveal the selected element in the Java editor.



5. Click the triangle in the upper right corner to see the quick view menu:



The menu items can be divided into 3 categories:

- *Position* - Allows you to resize and move the quick view and to remember these settings
- *Filter* - Define filters so that not all members are shown in the quick outline.
- *Sort* - Sort the members by their defining type or alphabetically.

Note: Ctrl+O always opens the outline for the current Java editor. Press Ctrl+F3 to open the quick outline for the currently selected type.

3. Adding new methods

1. Start adding a method by typing the following at the end of the *VectorTest.java* file (but before the closing brace of the type) in the Java editor:

```
public void testSizeIsThree()
```

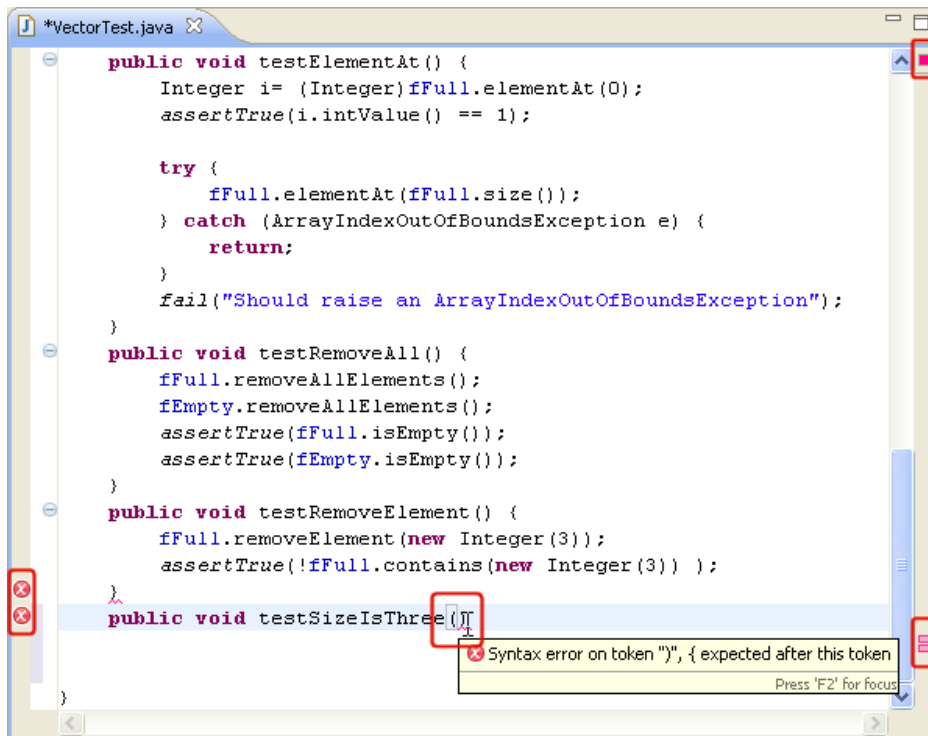
As soon as you type the method name in the editor area, the new method appears at the bottom of the Outline view.



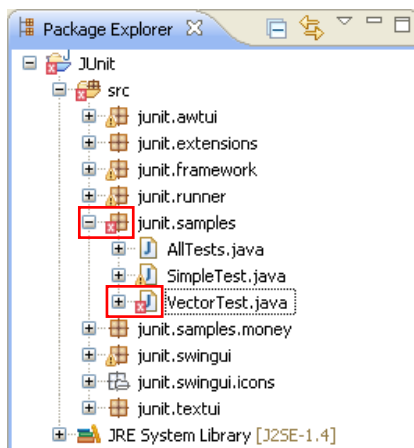
In addition,

- error annotations (red boxes) appear in the overview ruler positioned on the right hand side of the editor,
- error icons appear in the vertical ruler positioned on the left of the editor,
- an error indicator appears in the top right corner of the editor,
- errors are marked in the text.

These error annotations indicate that the compilation unit is currently not correct. If you hover over the error in the text, a tool tip appears: *Syntax error on token ")", { expected after this token*. This is correct since the method doesn't have a body yet. Note that error annotations in the editor are updated as you type. This behavior can be controlled via the **Report problems as you type** option located on the [Java > Editor](#) preference page.



2. Click the **Save** button. The compilation unit is compiled automatically and errors appear in the Package Explorer view, in the Problems view and on the vertical ruler (left hand side of the editor). In the Package Explorer view, the errors are propagated up to the project of the compilation unit containing the error.



3. Complete the new method by typing the following:

```
{
    //TODO: Check size
```

Note that the closing curly brace has been inserted automatically.

4. Save the file. Notice that the error indicators disappear since the missing brace has been added.

4. Using content assist

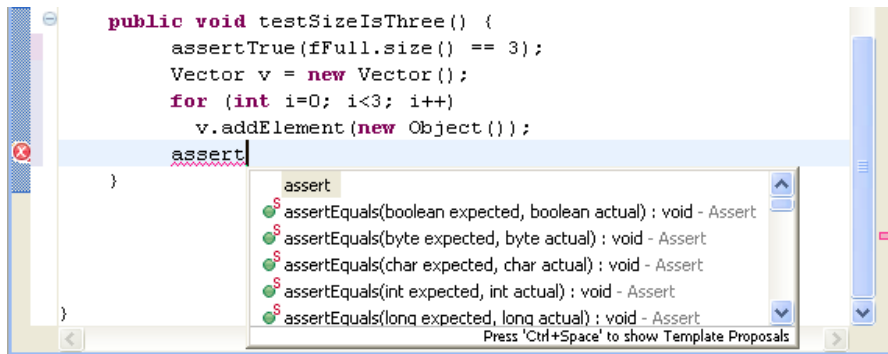
In this section you will use **content assist** to finish writing a new method. Open the file *junit.samples/VectorTest.java* in the Java editor if you do not already have it open and select the `testSizeIsThree()` method in the Outline view. If the file doesn't contain such a method see [Adding new methods](#) for instructions on how to add this method.

1. Replace the `TODO` comment with the following lines.

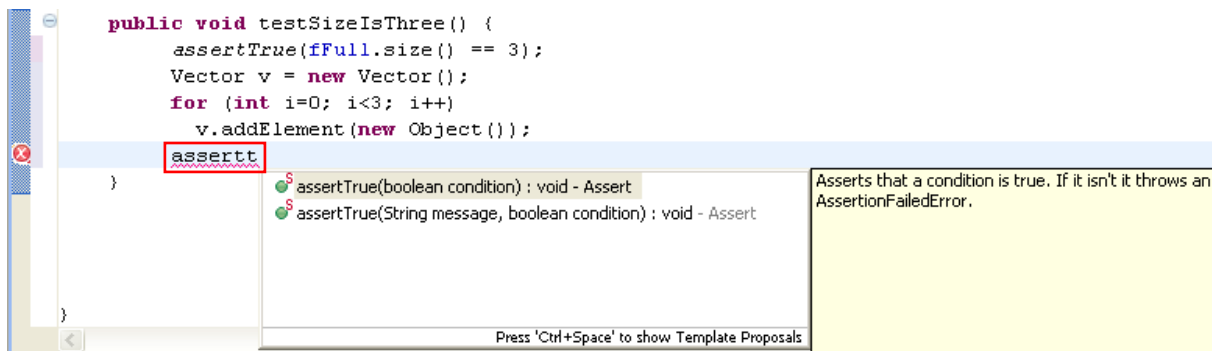
```
assertTrue(fFull.size() == 3);
Vector v = new Vector();
for (int i=0; i<3; i++)
    v.addElement(new Object());
assert
```

Alt + /

2. With your cursor at the end of the word `assert`, press `Ctrl+Space` to activate content assist. The content assist window with a list of proposals will appear. Scroll the list to see the available choices.



3. With the content assist window still active, type the letter 't' in the source code after `assert` (with no space in between). The list is narrowed and only shows entries starting with 'assert'. Single-click various items in the list to view any available Javadoc help for each item.



4. Select `assertTrue(boolean)` from the list and press `Enter`. The code for the `assertTrue(boolean)` method is inserted.
5. Complete the line so that it reads as follows:

```
assertTrue(v.size() == fFull.size());
```

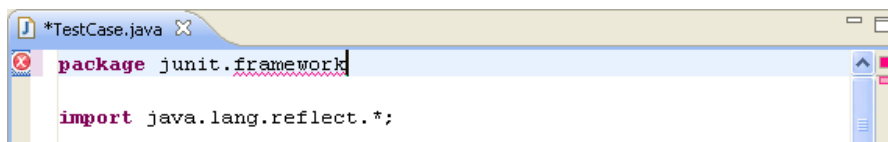
6. Save the file.

5. Identifying problems in your code

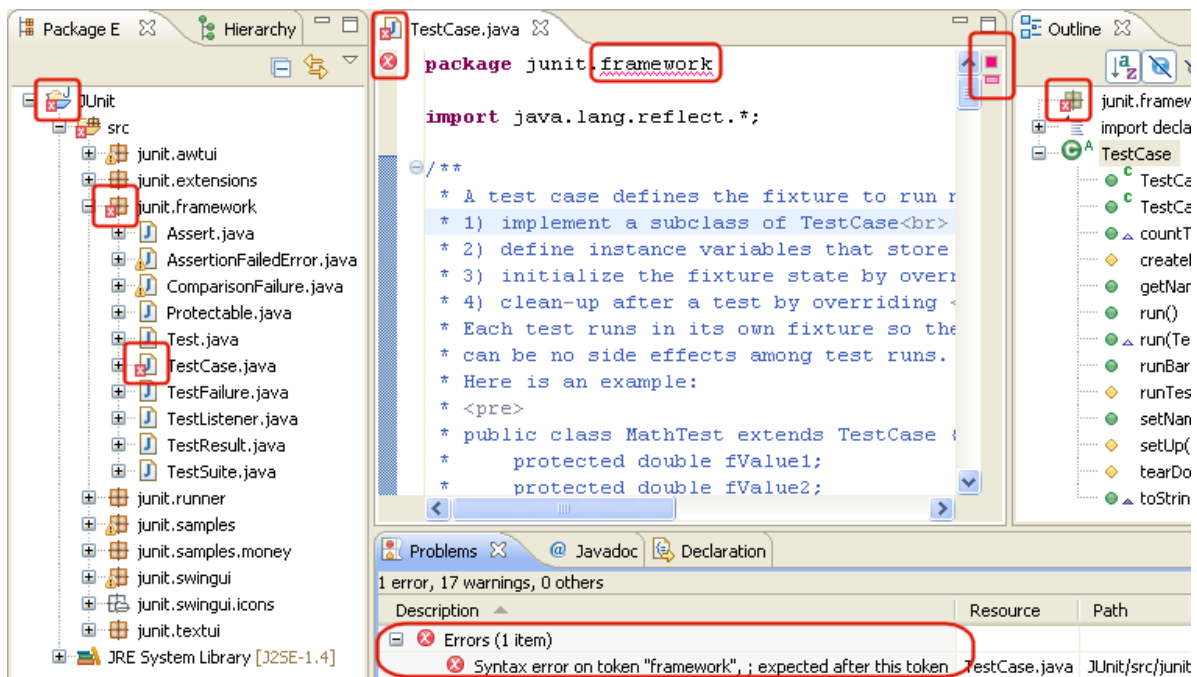
In this section, you will review the different indicators for identifying problems in your code.

Build problems are displayed in the Problems view and annotated in the vertical ruler of your source code.

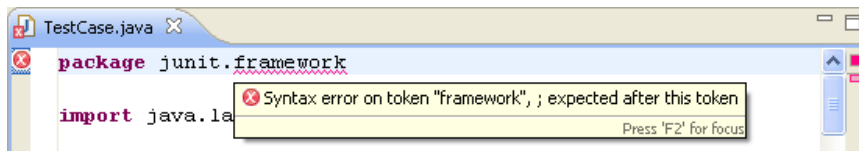
1. Open `JUnit4/junit4/TestRunner.java` in the editor from the Package Explorer view.
2. Add a syntax error by deleting the semicolon at the end of the package declaration in the source code.



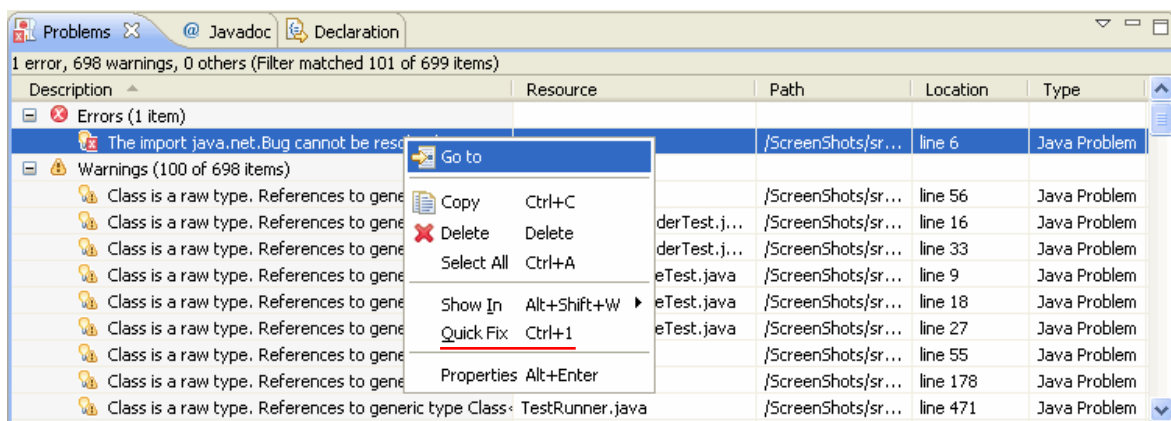
3. Click the **Save** button. The project is rebuilt and the problem is indicated in several ways:
 - In the Problems view, the problems are listed,
 - In the Package Explorer view, the Type Hierarchy or the Outline view, problem ticks appear on the affected Java elements and their parent elements,
 - In the editor's vertical ruler, a problem marker is displayed near the affected line,
 - Squiggly lines appear under the word which might have caused the error, and
 - The editor tab is annotated with a problem marker.



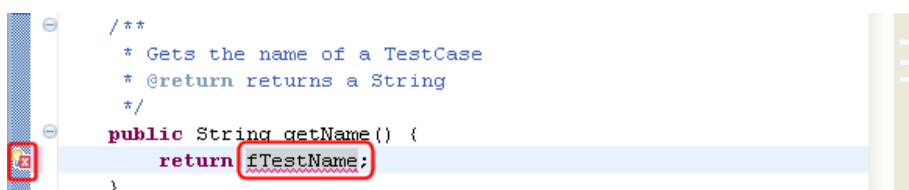
4. You can hover over the marked word in the editor to view a description of the problem. You can also hover over the problem markers in the vertical or overview rulers.



5. Click the **Close** ("X") button on the editor's tab to close the editor.
6. In the Problems view, select a problem in the list. Open its context menu and select **Go To**. The file is opened in the editor at the location of the problem.

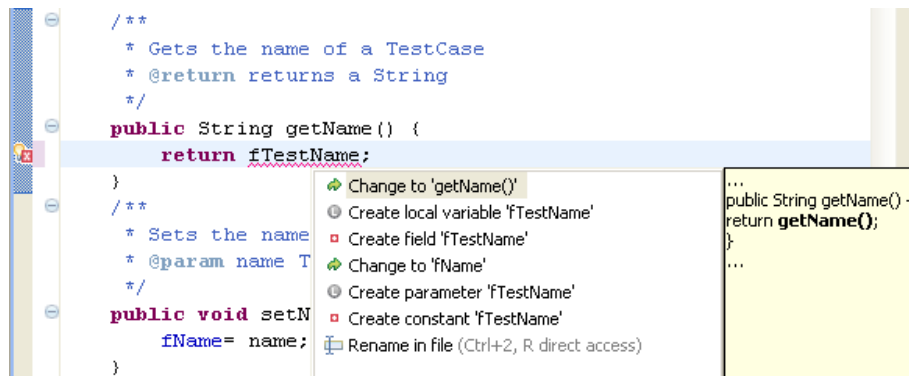


7. Correct the problem in the editor by adding the semicolon. Click the **Save** button. The project is rebuilt and the problem indicators disappear.
8. In the Outline view, select the method `getName()`. The editor will scroll to this method.
9. On the first line of the method change the returned variable `fName` to `fTestName`. While you type, a problem highlight underline appears on `fTestName`, to indicate a problem. Hovering over the highlighted problem will display a description of the problem and applicable quick fixes.
10. On the marker bar a light bulb marker appears. The light bulb signals that correction proposals are available for this problem.



11. Click to place the cursor onto the highlighted error, and choose **Quick Fix** from the Edit menu bar. You can also press

Ctrl+I or left click the light bulb. A selection dialog appears with possible corrections.



12. Select 'Change to fName' to fix the problem. The problem highlight line will disappear as the correction is applied.
13. Close the file without saving.
14. You can configure how problems are indicated on the [General > Editors > Text Editors > Annotations](#) preference page.

6. Using code templates

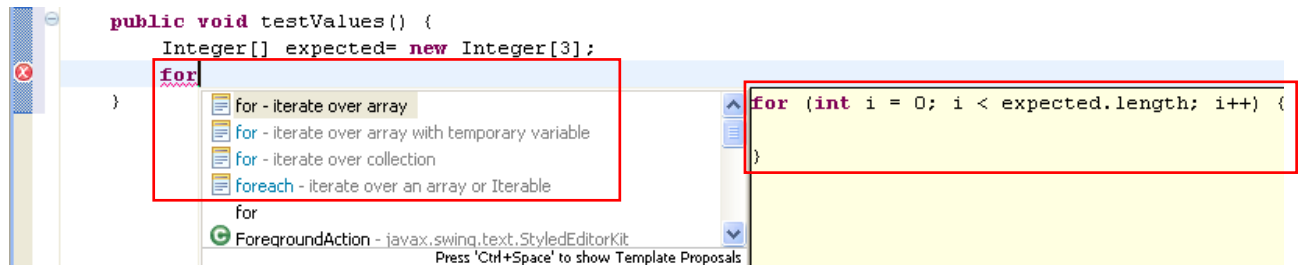
In this section you will use content assist to fill in a template for a common loop structure. Open `JUnit.samples/VectorTest.java` file in the Java editor if you do not already have it open.

1. Start adding a new method by typing the following:

```
public void testValues() {  
    Integer[] expected= new Integer[3];  
    for
```

Alt + /

2. With the cursor at the end of for, press Ctrl+Space to enable content assist. You will see a list of common templates for "for" loops. When you single-click a template, or select it with the Up or Down arrow keys, you'll see the code for the template in its help message. Note that the local array name is guessed automatically.



3. Choose the for - iterate over array entry and press Enter to confirm the template. The template will be inserted in your source code.

```
public void testValues() {  
    Integer[] expected= new Integer[3];  
    for (int i = 0; i < expected.length; i++) {  
        |  
    }  
}
```

4. Next we change the name of the index variable from *i* to *e*. To do so simply press *e*, as the index variable is automatically selected. Observe that the name of the index variable changes at all places. When inserting a template all references to the same variable are connected to each other. So changing one changes all the other values as well.

```
public void testValues() {  
    Integer[] expected= new Integer[3];  
    for (int e = 0; e < expected.length; e++) {  
        |  
    }  
}
```

5. Pressing the tab key moves the cursor to the next variable of the code template. This is the array `expected`.


```

public void testValues() {
    Integer[] expected= new Integer[3];
    for (int e = 0; e < expected.length; e++) {
    }
}

```

Since we don't want to change the name (it was guessed right by the template) we press tab again, which leaves the template since there aren't any variables left to edit.

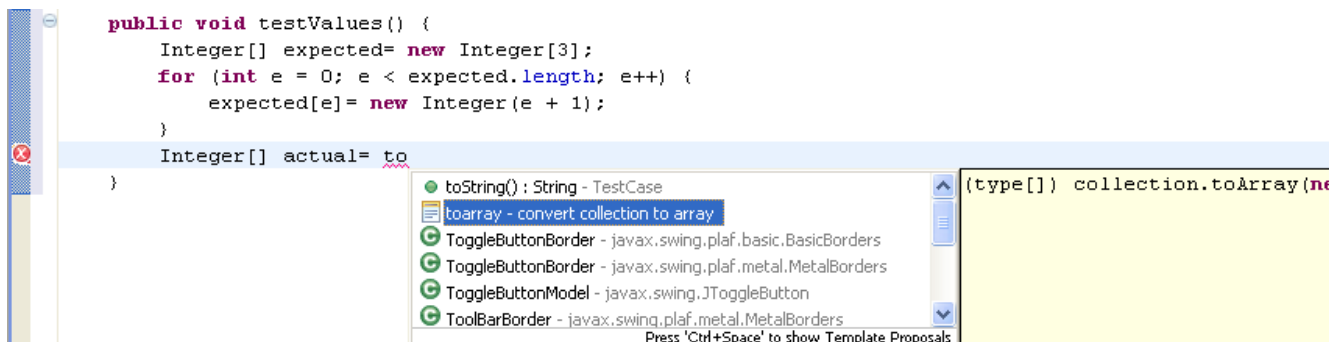
6. Complete the for loop as follows:

```

for (int e= 0; e < expected.length; e++) {
    expected[e]= new Integer(e + 1);
}
Integer[] actual= to

```

7. With the cursor at the end of to, press Ctrl+Space to enable content assist. Pick toarray - convert collection to array and press Enter to confirm the selection (or double-click the selection).



```

public void testValues() {
    Integer[] expected= new Integer[3];
    for (int e = 0; e < expected.length; e++) {
        expected[e]= new Integer(e + 1);
    }
    Integer[] actual= to
}

```

The template is inserted in the editor and type is highlighted and selected.

```

public void testValues() {
    Integer[] expected= new Integer[3];
    for (int e = 0; e < expected.length; e++) {
        expected[e]= new Integer(e + 1);
    }
    Integer[] actual= (type[]) collection.toArray(new type[collection.size()])
}

```

8. Overwrite the selection by typing Integer. The type of array constructor changes when you change the selection.
9. Press Tab to move the selection to collection and overwrite it by typing fFull.

```

public void testValues() {
    Integer[] expected= new Integer[3];
    for (int e = 0; e < expected.length; e++) {
        expected[e]= new Integer(e + 1);
    }
    Integer[] actual= (Integer[]) fFull.toArray(new Integer[fFull.size()])
}

```

10. Add a semicolon and the following lines of code to complete the method:

```

assertEquals(expected.length, actual.length);
for (int i= 0; i < actual.length; i++)
    assertEquals(expected[i], actual[i]);

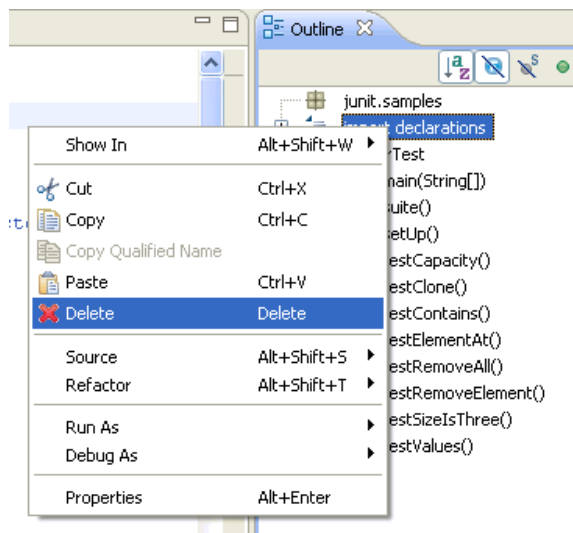
```

11. Save the file.

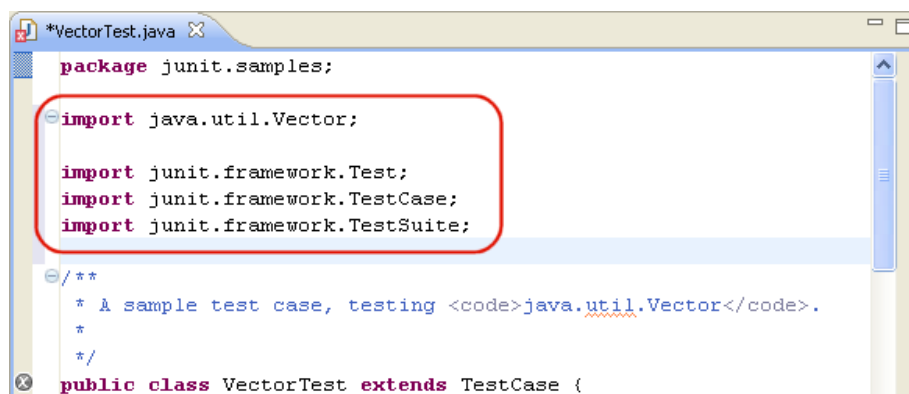
7. Organizing import statements

In this section you will organize the import declarations in your source code. Open *junit.samples/VectorTest.java* file in the Java editor if you do not already have it open.

1. Delete the import declarations by selecting them in the Outline view and selecting **Delete** from the context menu. Confirm the resulting dialog with **Yes**. You will see numerous compiler warnings in the vertical ruler since the types used in the method are no longer imported.



2. From the context menu in the editor, select **Source > Organize Imports**. The required import statements are added to the beginning of your code below the package declaration.



You can also choose **Organize Imports** from the context menu of the import declarations in the Outline view.

Note: You can specify the order of the import declarations using the [Java > Code Style > Organize Imports](#) preference page.

3. Save the file.

8. Using the local history

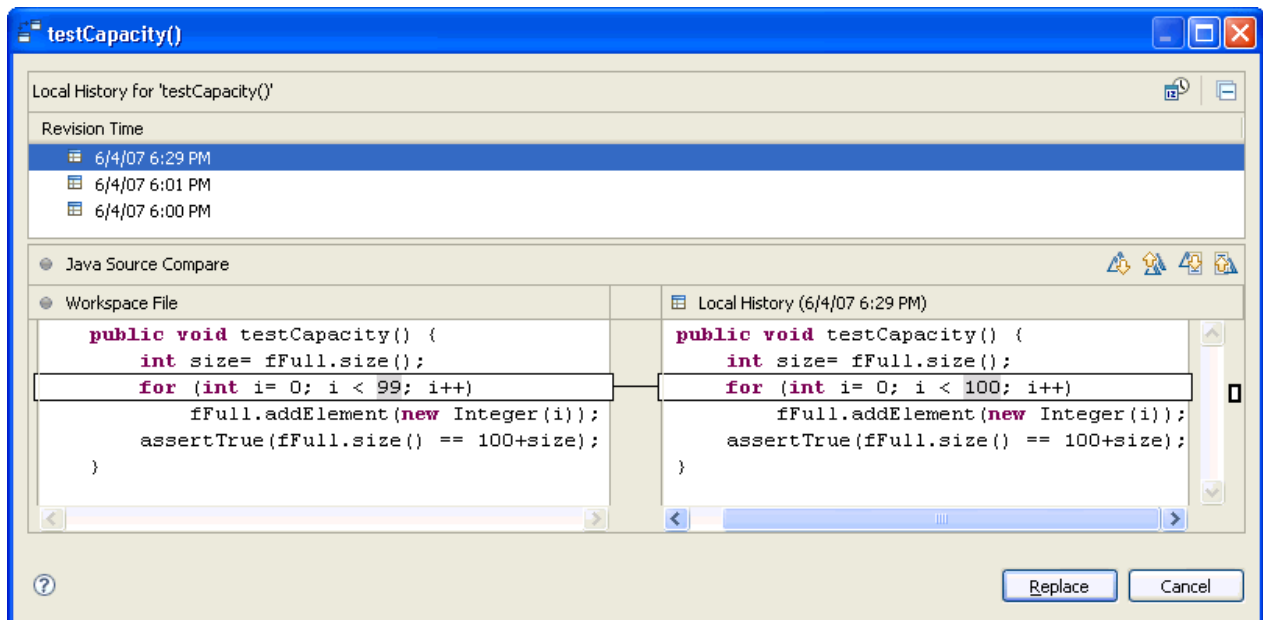
In this section, you will use the local history feature to switch to a previously saved version of an individual Java element.

1. Open *junit.samples/VectorTest.java* file in the Java editor and select the method *testCapacity()* in the Outline view.
2. Change the content of the method so that the 'for' statements reads as:

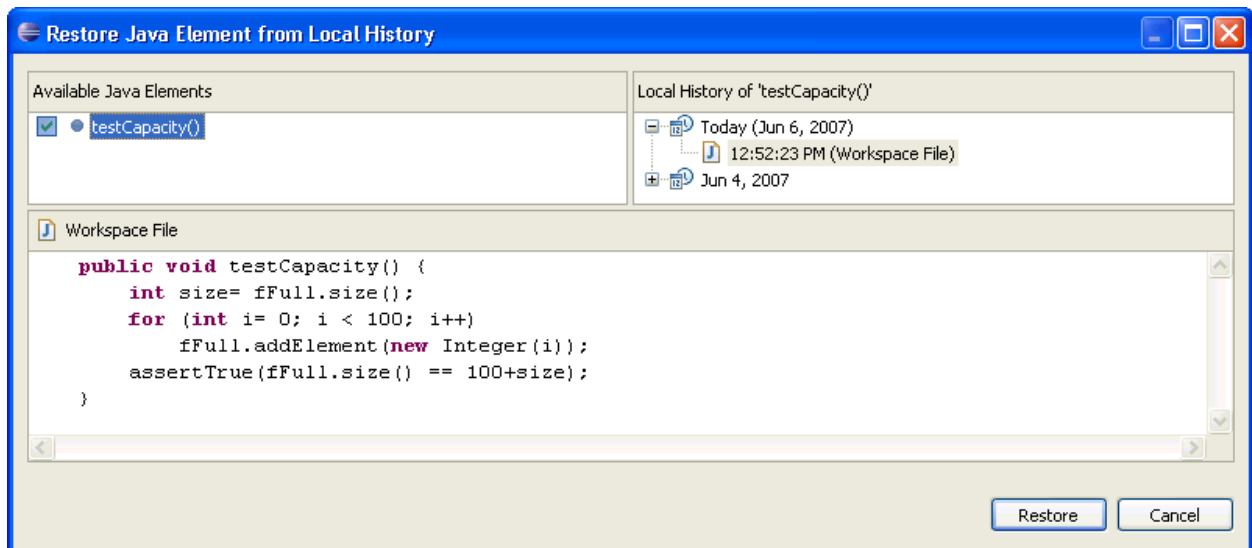
```
for (int i= 0; i < 99; i++)
```

Save the file by pressing **Ctrl+S**.

3. In the Outline view, select the method *testCapacity()*, and from its context menu, select **Replace With > Element from Local History**.
4. In the **Replace Java Element from Local History** dialog, the Local History list shows the various saved states of the method. The Java Source Compare pane shows details of the differences between the selected history resource and the existing workbench resource.



5. In the top pane, select the previous version, and click the **Replace** button. In the Java editor, the method is replaced with the selected history version.
6. Save the file.
7. Beside replacing a method's version with a previous one you can also restore Java elements that were deleted. Again, select the method *testCapacity()* in the Outline view, and from its context menu, select **Delete**. Confirm the resulting dialog with **Yes** and save the file.
8. Now select the type *VectorTest* in the Outline view, and from its context menu, select **Restore from Local History....** Select and check the method *testCapacity()* in the Available Java Elements pane. As before, the Local History pane shows the versions saved in the local history.



9. In the Local History pane, select the earlier working version and then click **Restore**.

9. Extracting a new method

In this section, you will improve the code of the constructor of *junit.framework.TestSuite*. To make the intent of the code clearer, you will extract the code that collects test cases from base classes into a new method called *collectInheritedTests*.

1. In the *junit.framework/TestSuite.java* file, select the following range of code inside the *TestSuite(Class)* constructor:

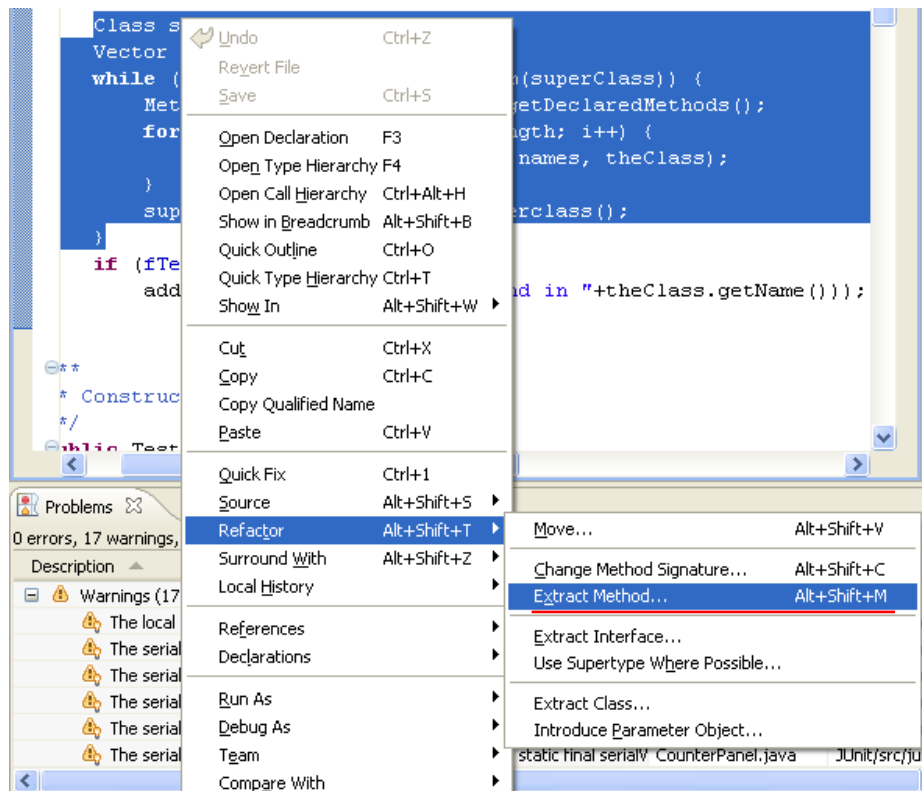
```
Class superClass= theClass;
Vector names= new Vector();
while(Test.class.isAssignableFrom(superClass)) {
    Method[] methods= superClass.getDeclaredMethods();
    for (int i= 0; i < methods.length; i++) {
        addTestMethod(methods[i],names, constructor);
    }
}
```

```

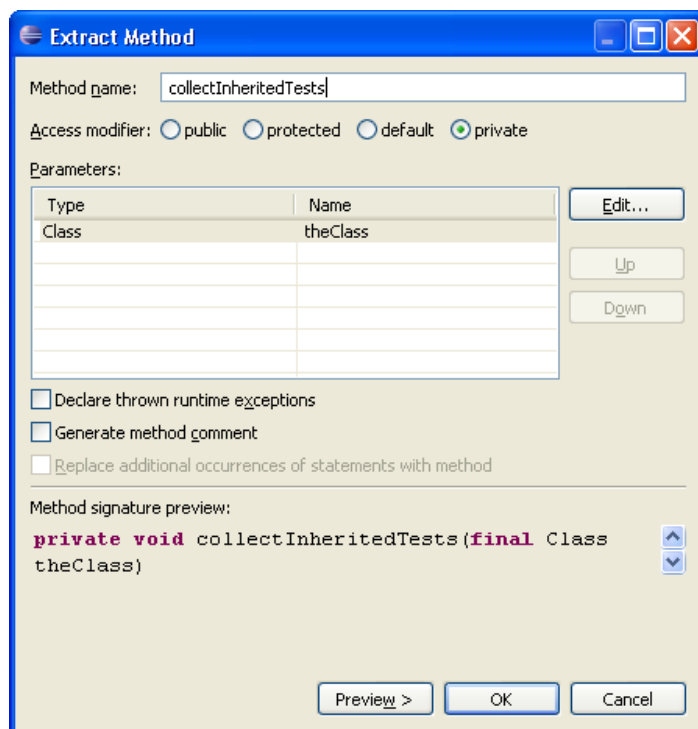
    superClass= superClass.getSuperclass();
}

```

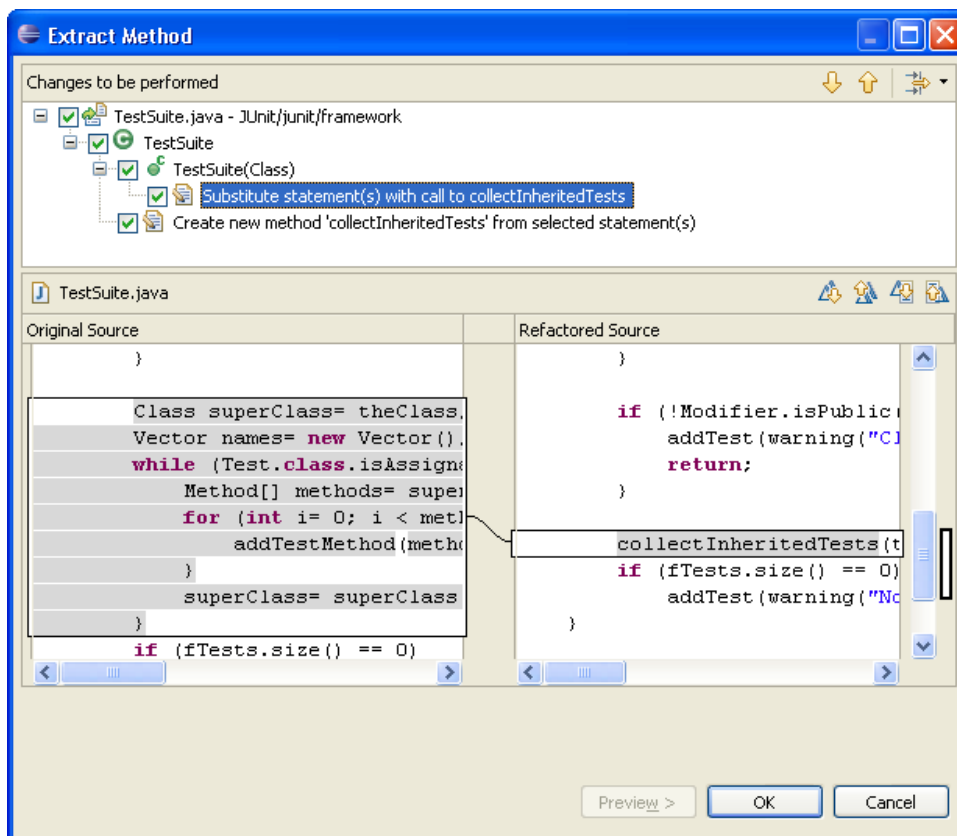
- From the selection's context menu in the editor, select **Refactor > Extract Method...**



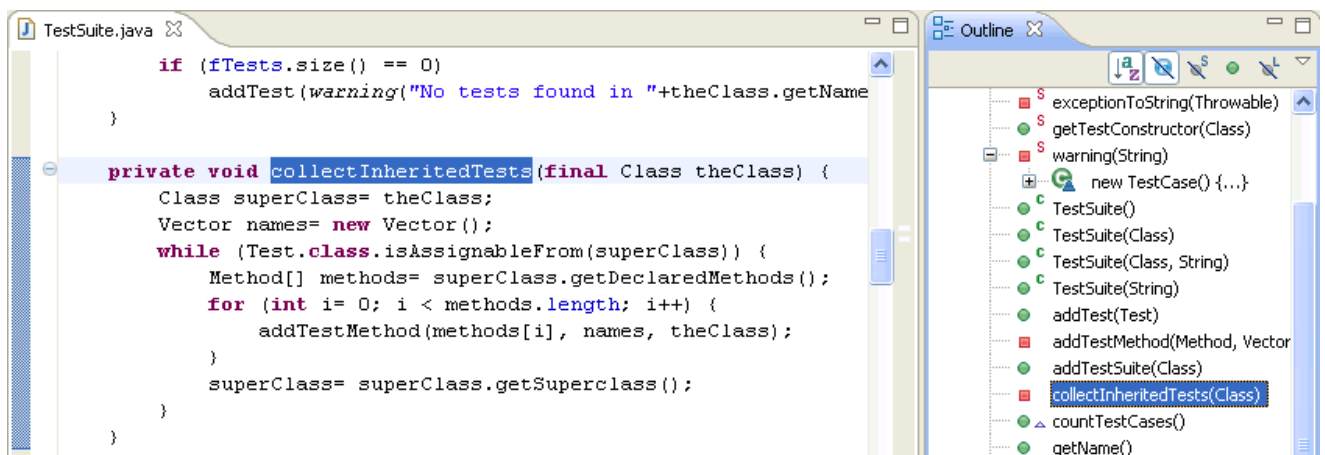
- In the **Method Name** field, type *collectInheritedTests*.



- To preview the changes, press **Preview>**. The preview page displays the changes that will be made. Press **OK** to extract the method.




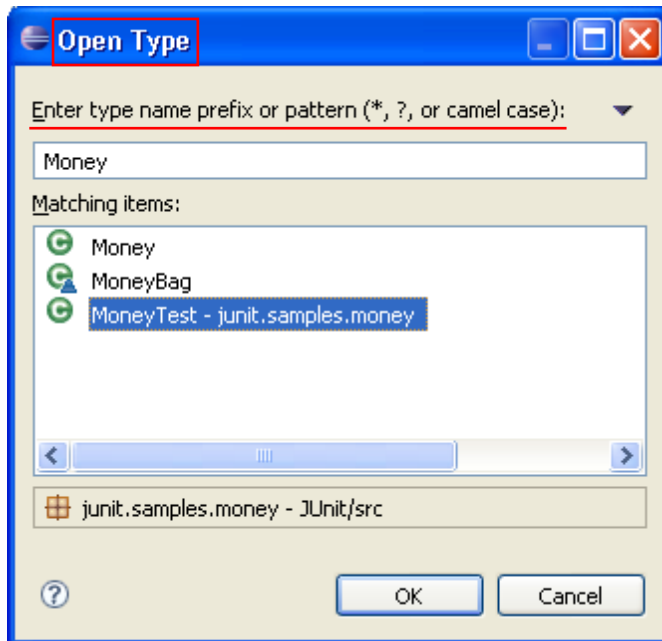
5. Go to the extracted method by selecting it in the Outline view.



Navigate to a Java element's declaration

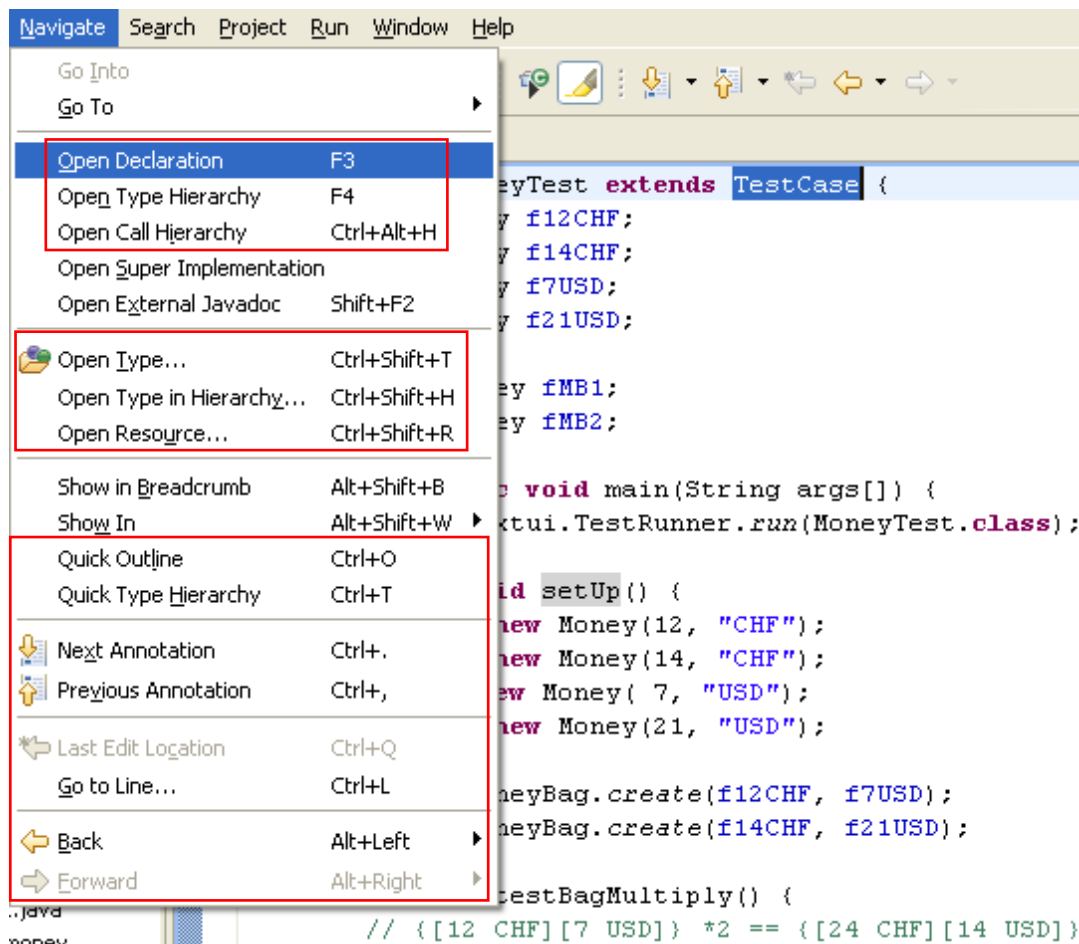
In this section, you will learn how to open a type in the Java Editor and how to navigate to related types and members.

1. Open the **Open Type** dialog by pressing `Ctrl+Shift+T`, choosing **Navigate > Open Type...**, or clicking the toolbar icon (). Type `Money`, press the `Arrow Down` key a few times to select `MoneyTest`, and then press `Enter` to open the type in the Java editor.

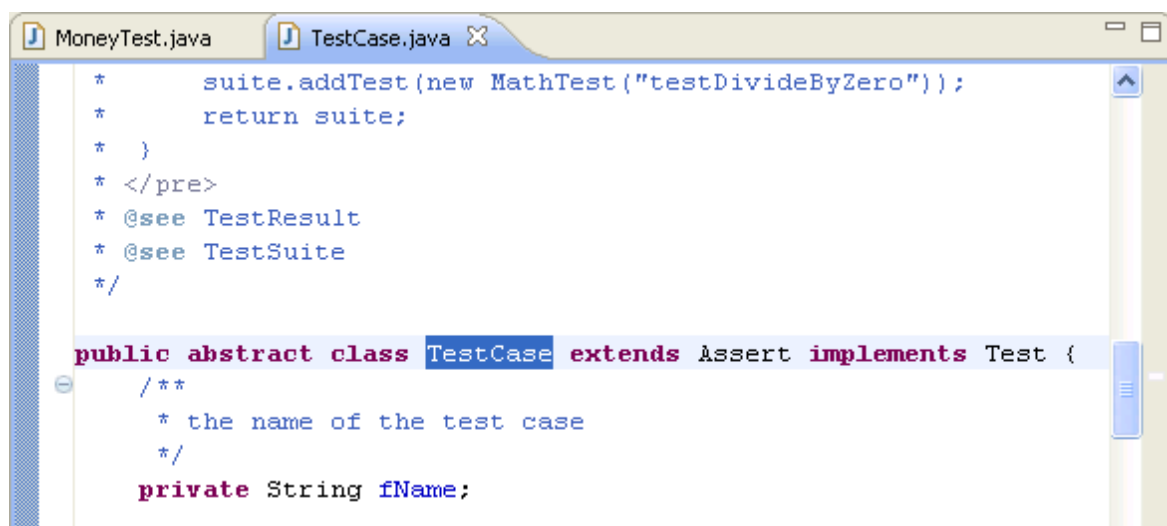


2. On the first line of the `MoneyTest` class declaration, select the superclass `TestCase` and either
 - from the menu bar select **Navigate > Open Declaration** or
 - press `F3`.

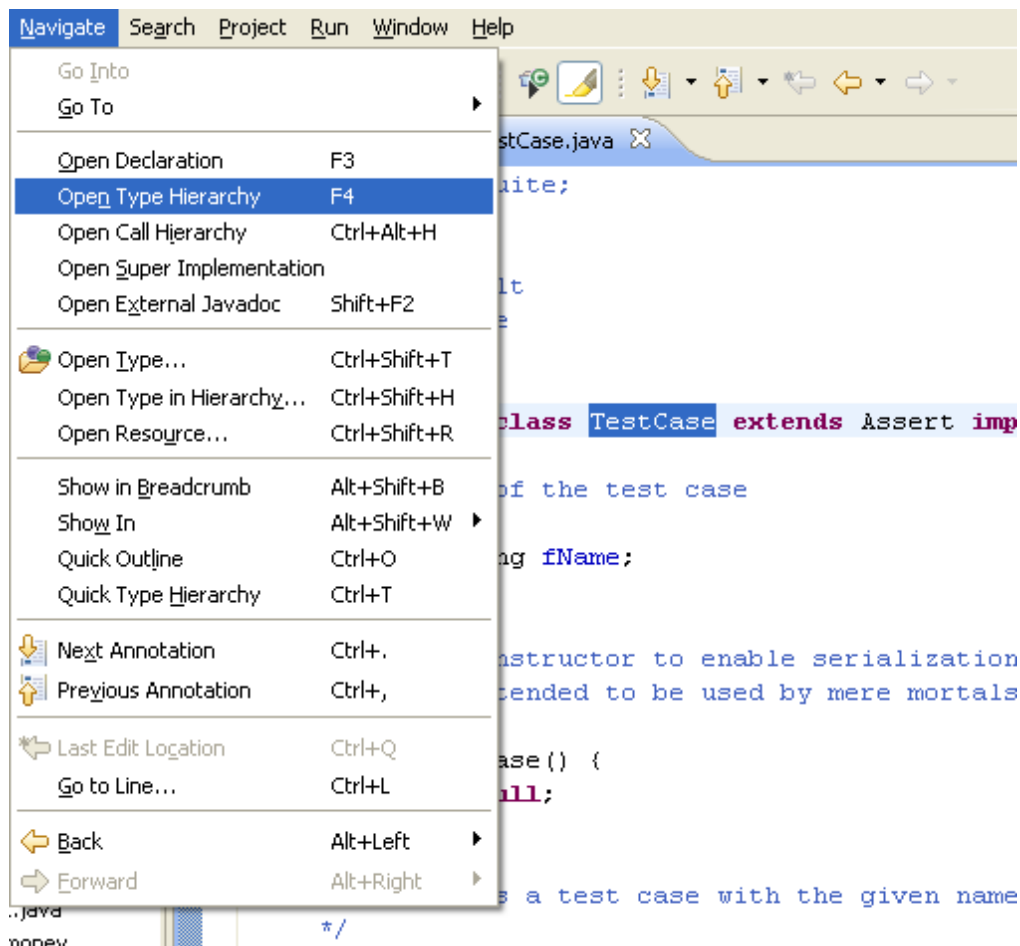
`Ctrl + Mouse`



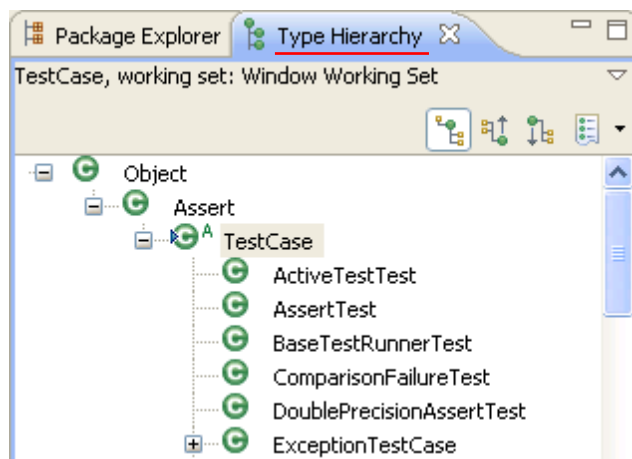
The `TestCase` class opens in the editor area and is also represented in the Outline view.
Note: This command also works on methods and fields.



3. With the `TestCase.java` editor open and the class declaration selected:
 - from the menu bar select **Navigate > Open Type Hierarchy** or
 - press **F4**.




4. The Type Hierarchy view opens with the TestCase class displayed.

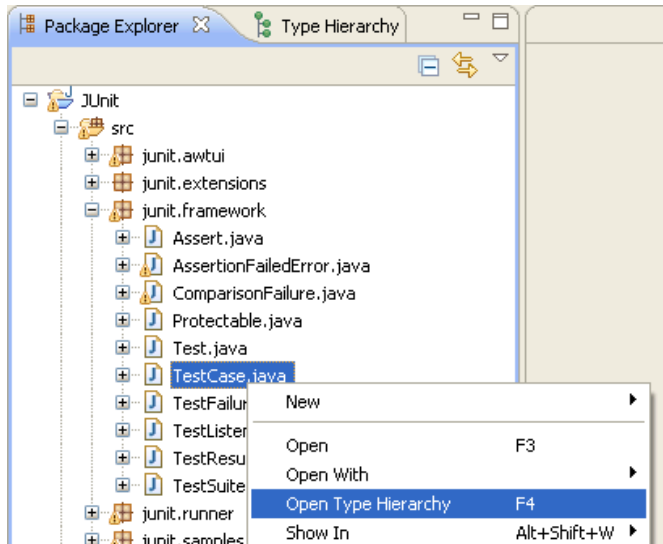


Note: You can also open editors on types and methods in the Type Hierarchy view.

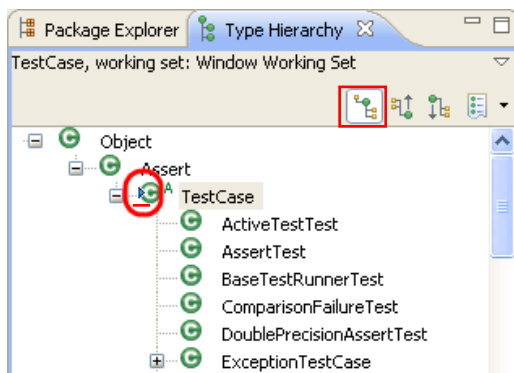
Viewing the type hierarchy

In this section, you will learn about using the Type Hierarchy view by viewing classes and members in a variety of different ways.

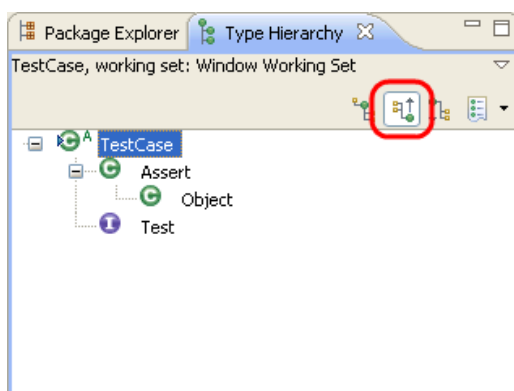
1. In the Package Explorer view, find `junit.framework.TestCase.java`. From its context menu, select  **Open Type Hierarchy**. You can also open type hierarchy view:
 - from the menu bar by selecting **Navigate > Open Type Hierarchy**.
 - from the keyboard by pressing **F4** after selecting `TestCase.java`.



2. The buttons in the view tool bar control which part of the hierarchy is shown. Click the **Show the Type Hierarchy** button to see the class hierarchy, including the base classes and subclasses. The small arrow on the left side of the type icon of `TestCase` indicates that the hierarchy was opened on this type.

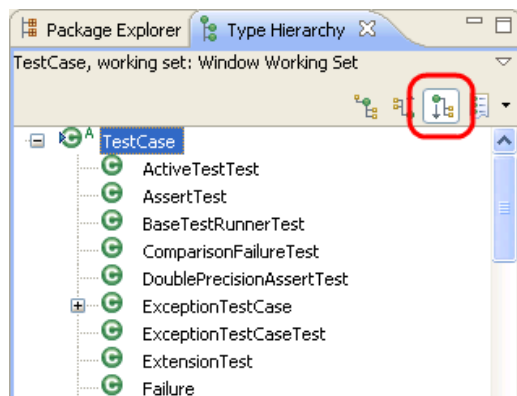


3. Click the **Show the Supertype Hierarchy** button to see a hierarchy showing the type's parent elements including implemented interfaces. This view shows the results of going up the type hierarchy.

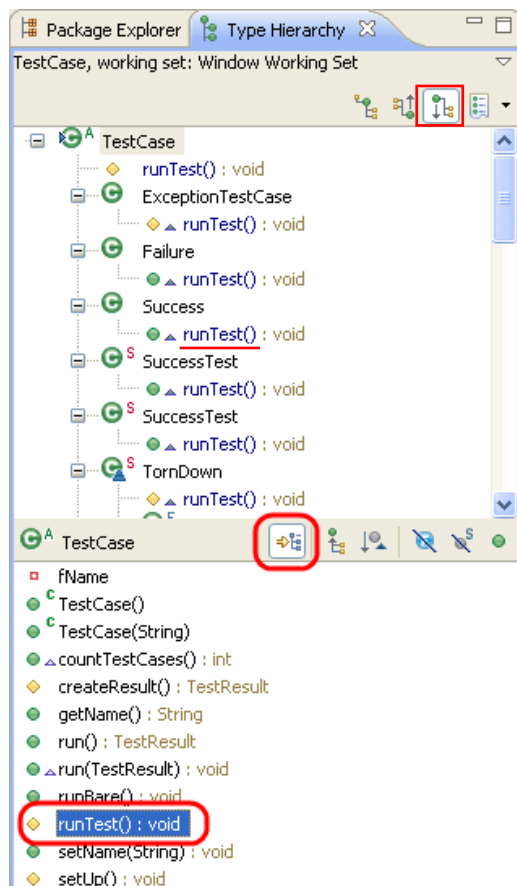


In this "reversed hierarchy" view, you can see that `TestCase` implements the `Test` interface.

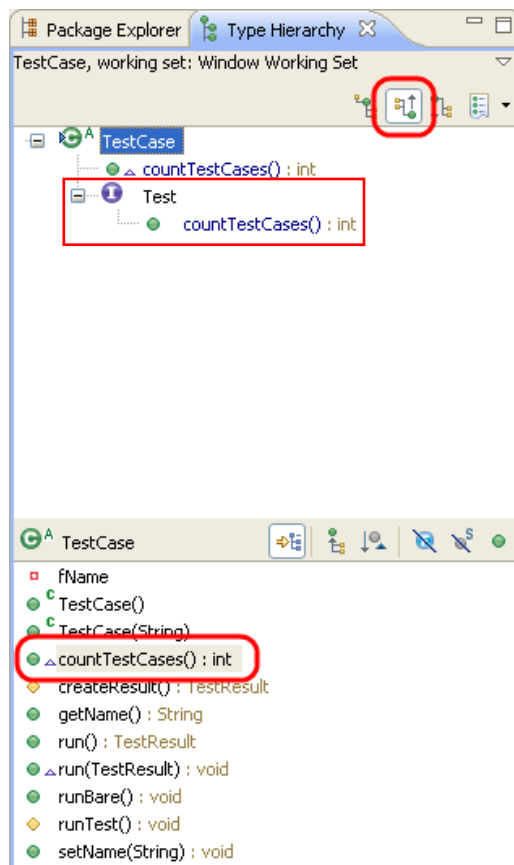
4. Click the **Show the Subtype Hierarchy** button in the view toolbar.



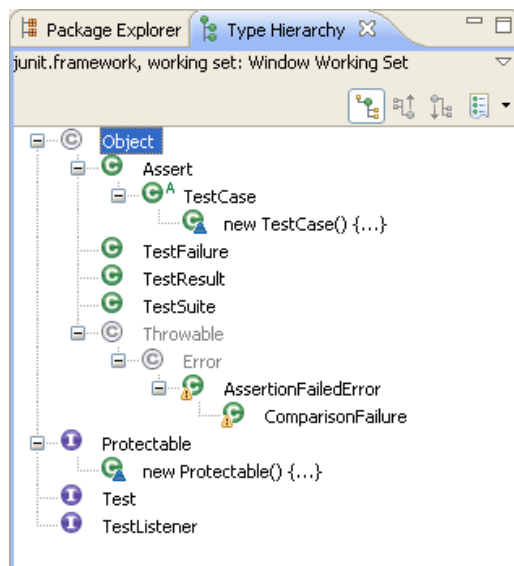
- Click the **Lock View and Show Members in Hierarchy** button in the toolbar of the member pane, then select the runTest() method in the member pane. The view will now show all the types implementing runTest().



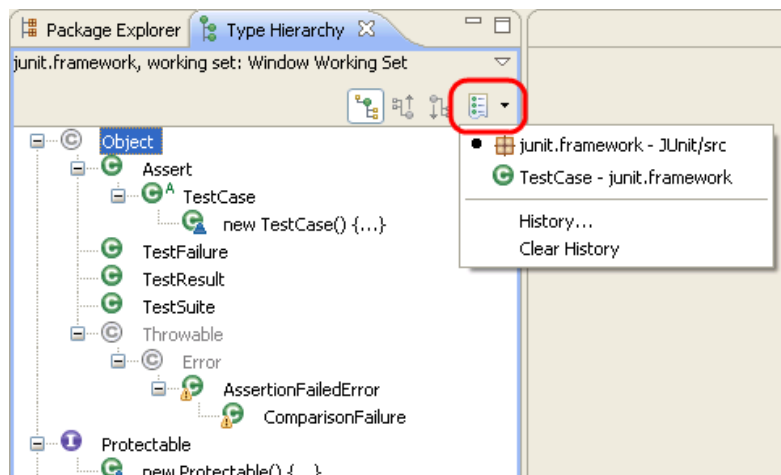
- In the Type Hierarchy view, click the **Show the Supertype Hierarchy** button. Then on the member pane, select countTestCases() to display the places where this method is declared.



7. In the Type Hierarchy view select the *Test* element and select **Focus On 'Test'** from its context menu. *Test* is presented in the Type Hierarchy view.
8. Activate the Package Explorer view and select the package junit.framework. Use **Open Type Hierarchy** from its context menu. A hierarchy is opened containing all classes of the package. For completion of the tree, the hierarchy also shows some classes from other packages. These types are shown by a type icon with a white fill.

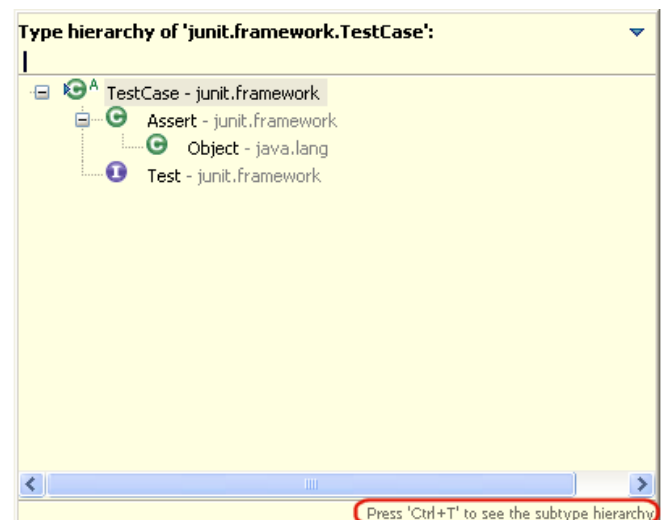
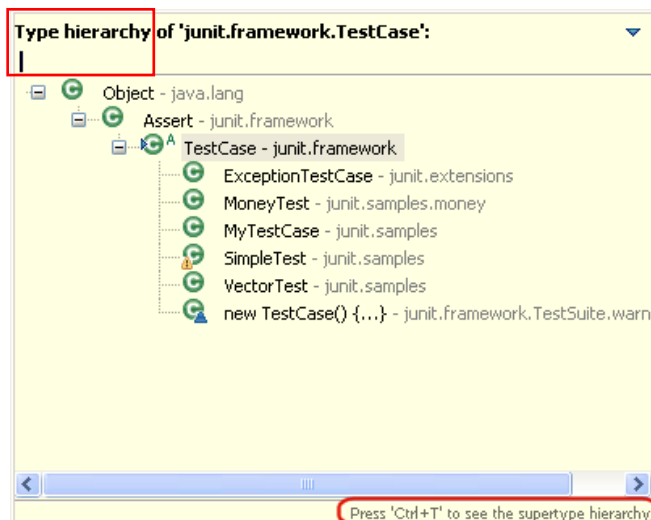


9. Use **Previous Type Hierarchies** to go back to a previously opened element. Click on the arrow next to the button to see a list of elements or click on the button to edit the history list.



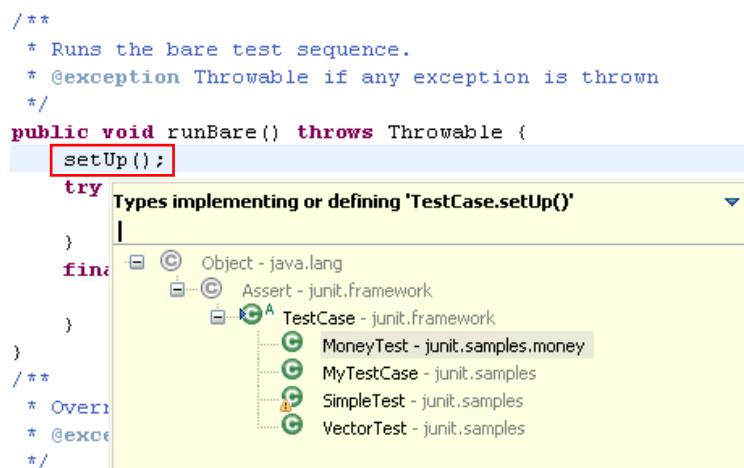
If you are working in the editor and only want to do a quick lookup for a hierarchy you can use the **Quick Type Hierarchy**:

1. Open *junit.framework.TestCase.java* file in the Java editor if you do not already have it open.
2. Select the type name in the Java editor
3. Press **Ctrl+T** or invoke **Navigate > Quick Type Hierarchy** and the in-place type hierarchy view is shown.
4. Pressing **Ctrl+T** while the type hierarchy view is shown will toggle between supertype hierarchy and subtype hierarchy.



To see where a virtual method call can resolve to:

1. In the body of *runBare()* select the invocation of *setUp()*
2. Press **Ctrl+T** or invoke **Navigate > Quick Type Hierarchy** and the in-place type hierarchy view is shown.
3. You can see that *setUp()* is implemented in 3 more classes. *Object* and *Assert* are only shown with a white filled images as are only required to complete the hierarchy but do not implement *setUp()*
4. Select a type to navigate to its implementation of *setUp()*



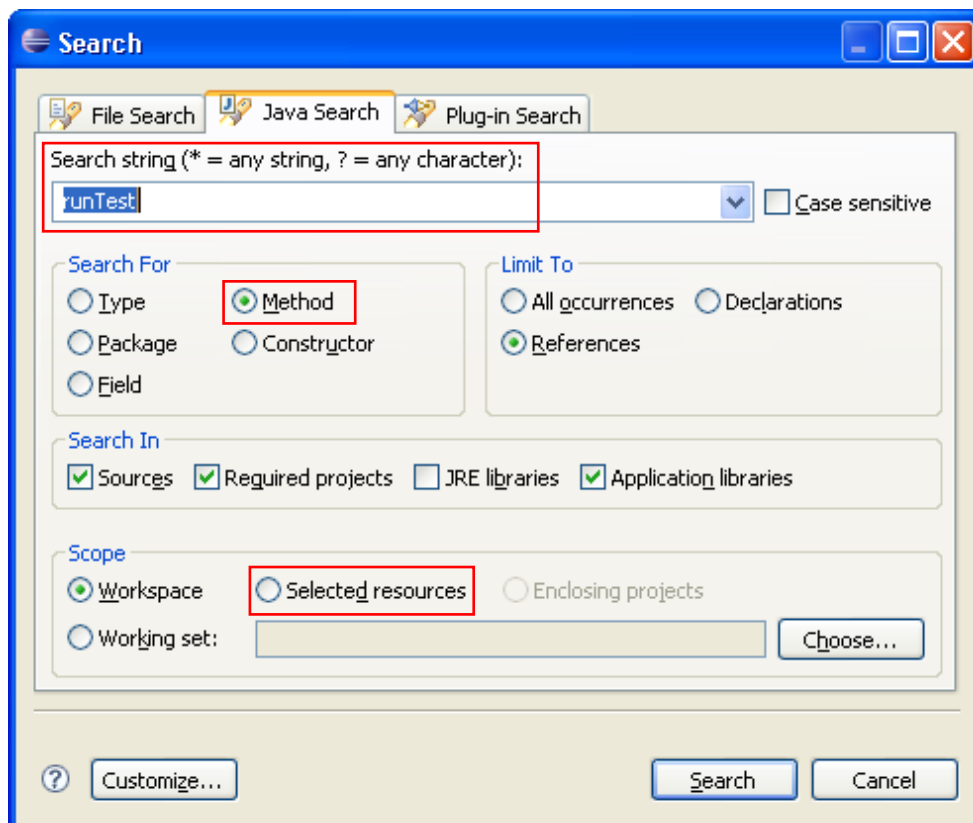
Searching the workbench

In this section, you will search the workbench for Java elements.

In the Search dialog, you can perform file, text or Java searches. Java searches operate on the structure of the code. **File searches** operate on the files by name and/or text content. Java searches are faster, since there is an underlying indexing structure for the code structure. Text searches allow you to find matches inside comments and strings.

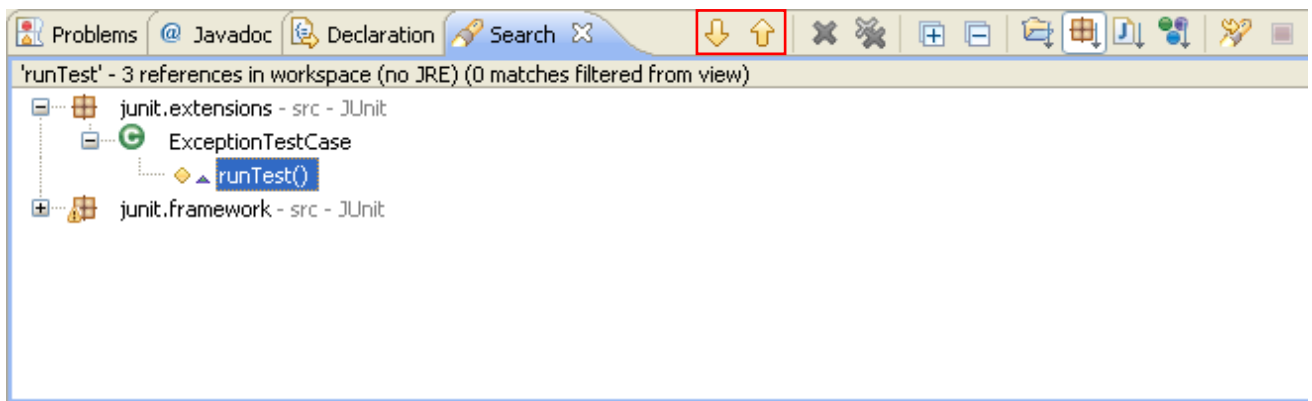
Performing a Java search from the workbench

1. In the Java perspective, click the **Search** (🔍) button in the workbench toolbar or use **Search > Java...** from the menu bar.
2. If it is not already selected, select the **Java Search** tab.
3. In the **Search string** field, type `runTest`. In the **Search For** area, select **Method**, and in the **Limit To** area, select **References**.
Verify that the **Scope** is set to **Workspace**.



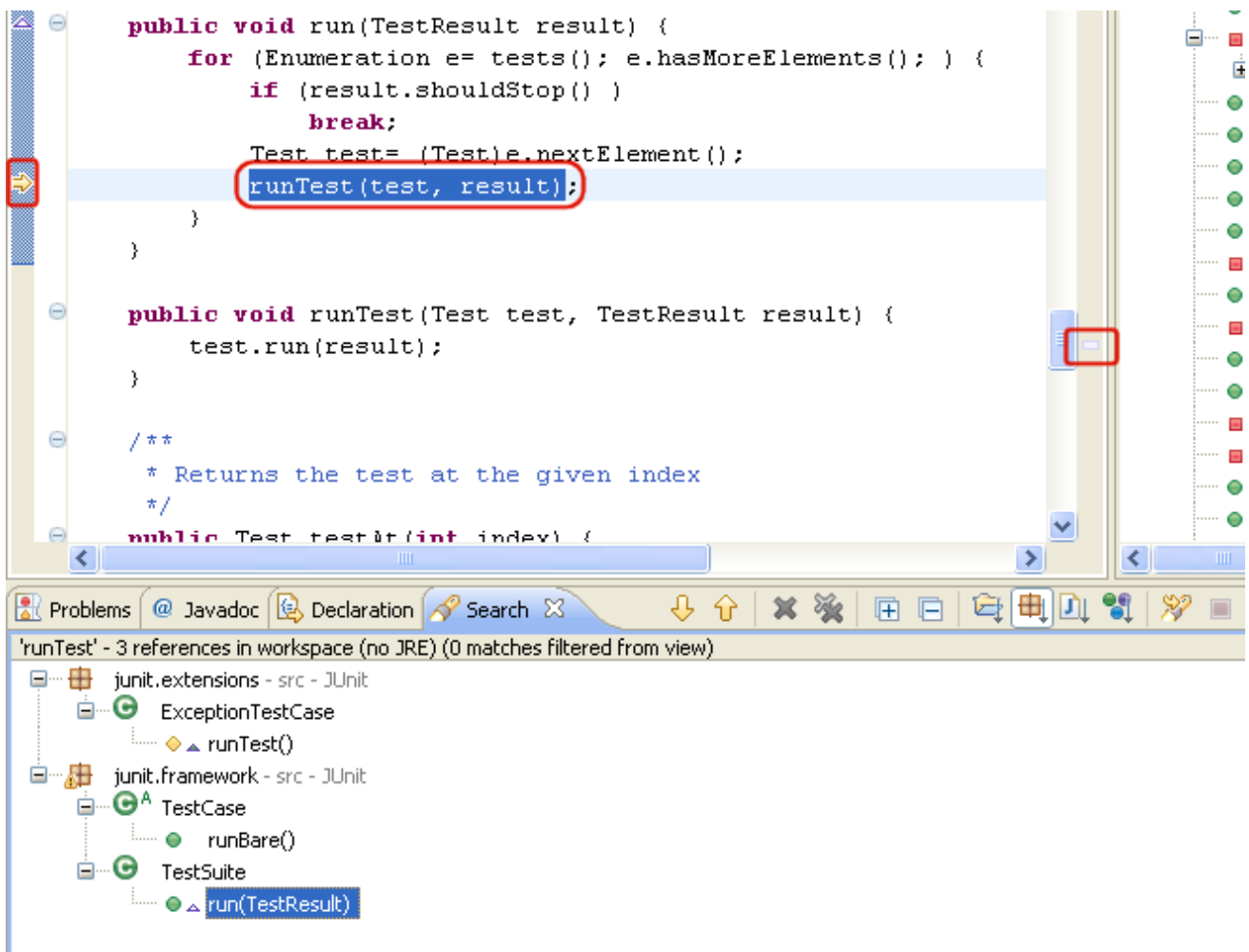
Then click **Search**. While searching you may click **Cancel** at any time to stop the search. Partial results will be shown.

4. In the Java perspective, the Search view shows the search results.



Use the **Show Next Match** (⬇️) and **Show Previous Match** (⬆️) buttons to navigate to each match. If the file in which the match was found is not currently open, it is opened in an editor.

- When you navigate to a search match using the Search view buttons, the file opens in the editor at the position of the match. Search matches are tagged with a search marker in the rulers.

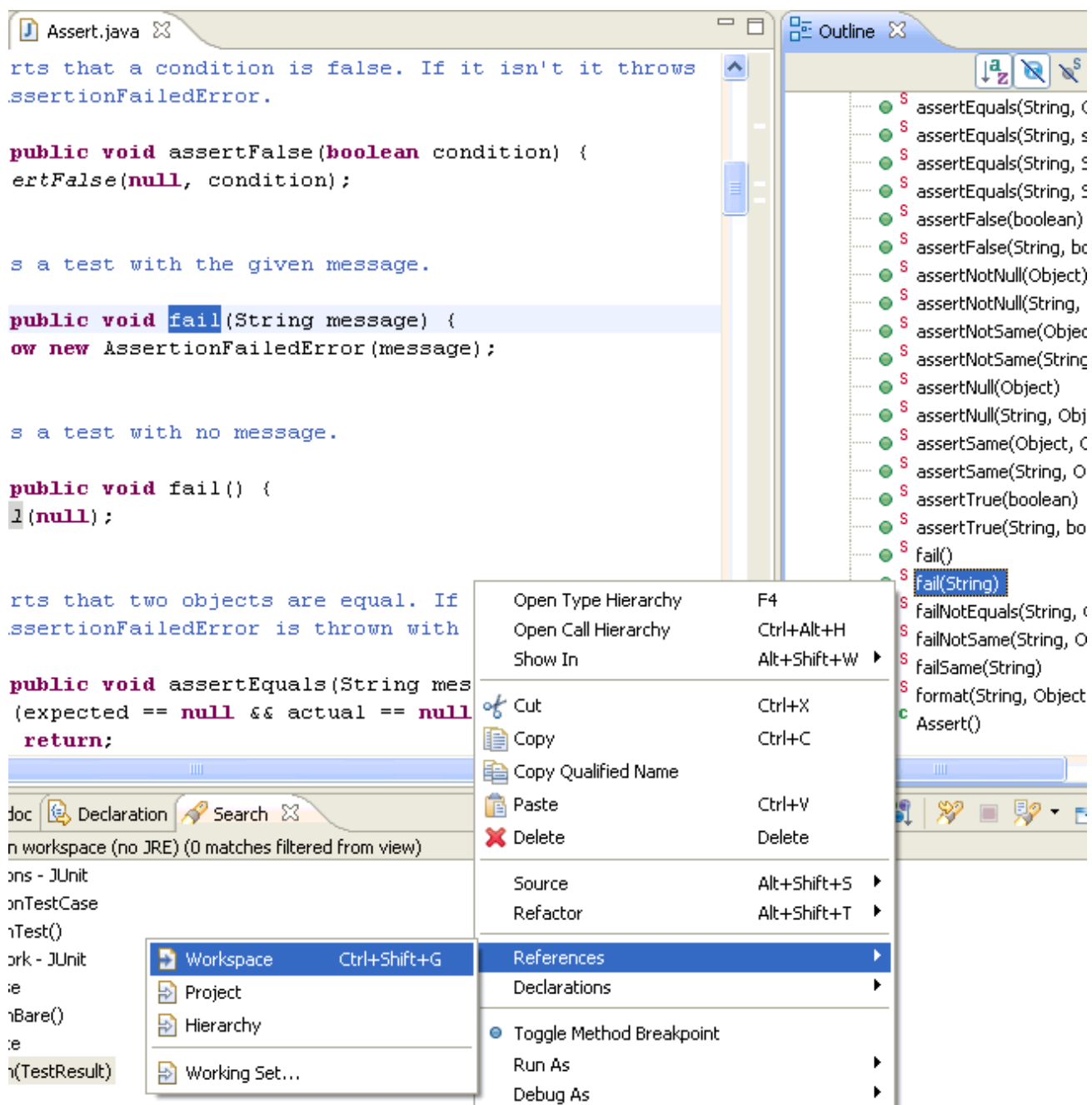


Searching from a Java view

Java searches can also be performed from specific views, including the Outline, Type Hierarchy view and the Package Explorer view.

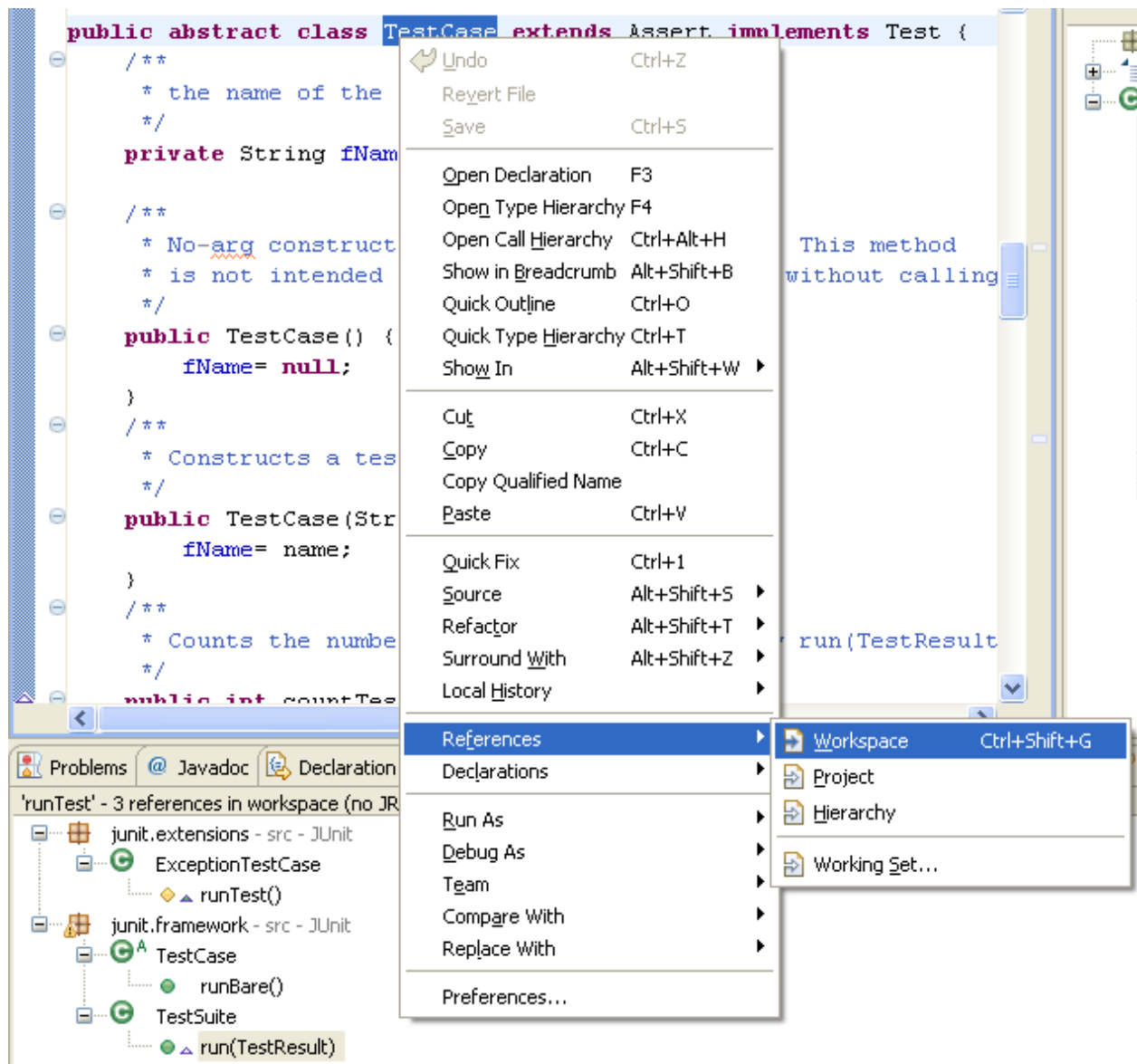
- In the Package Explorer view, double-click *junit.framework.Assert.java* to open it in an editor.

2. In the Outline view, select the fail(String) method, and from its context menu, select **References > Workspace**.



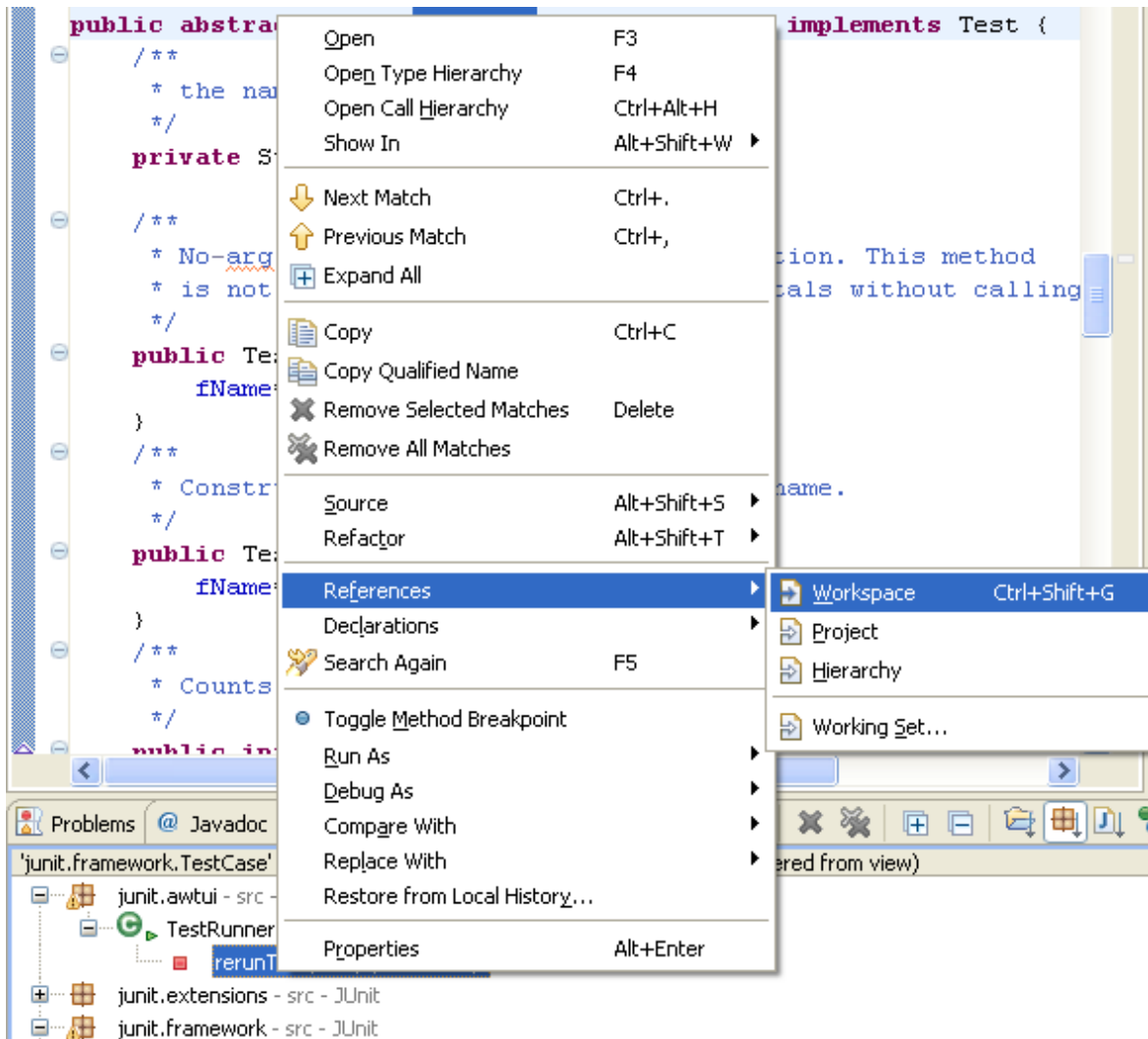
Searching from an editor

From the Package Explorer view, open `junit.framework.TestCase.java` . In the editor, select the class name `TestCase` and from the context menu, select **References > Workspace**.



Continuing a search from the search view

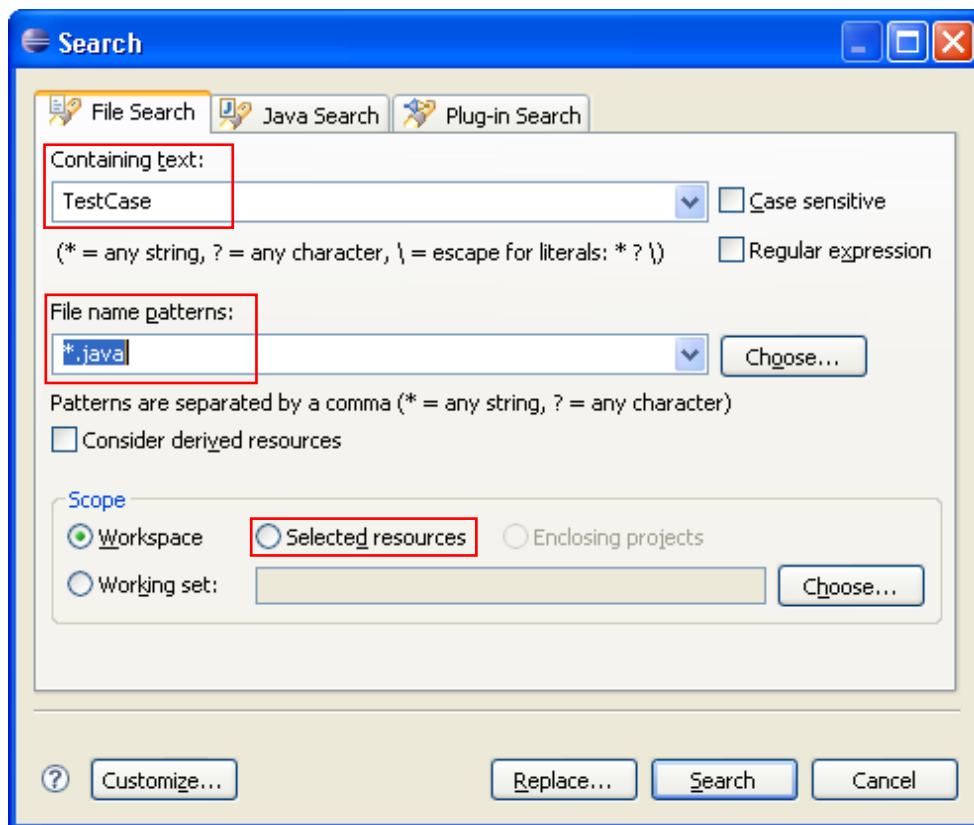
The Search Results view shows the results for the `TestCase` search. Select a search result and open the context menu. You can continue searching the selected element's references and declarations.



If you want to follow multiple levels of method calls, you can also use **Navigate > Open Call Hierarchy**.

Performing a file search

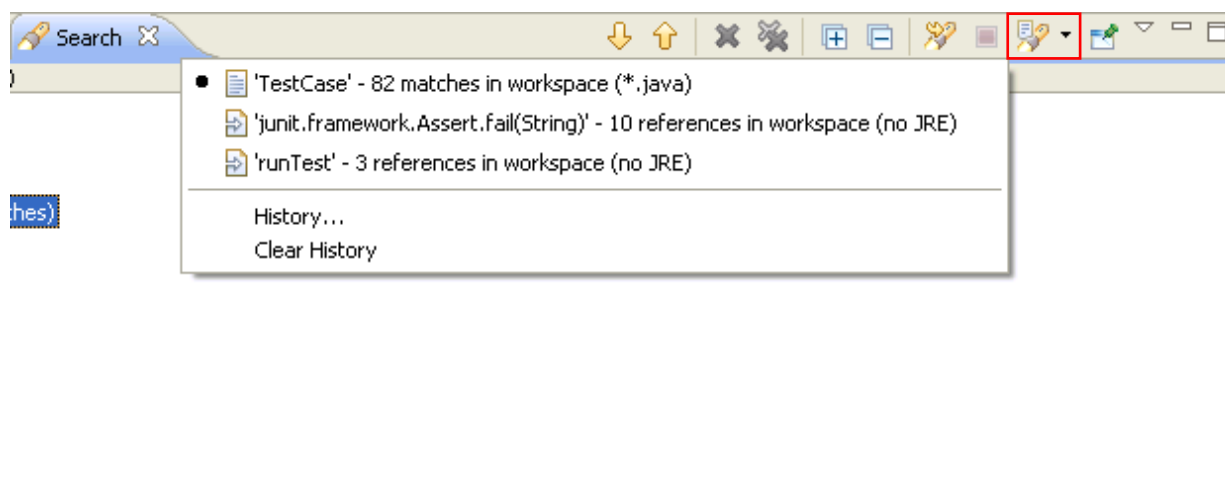
1. In the Java perspective, click the **Search** button in the workbench toolbar or select **Search > File** from the menu bar.
2. If it is not already selected, select the **File Search** tab.
3. In the **Containing text** field, type *TestCase*. Make sure that the **File name patterns** field is set to **.java*. The **Scope** should be set to *Workspace*. Then click **Search**.



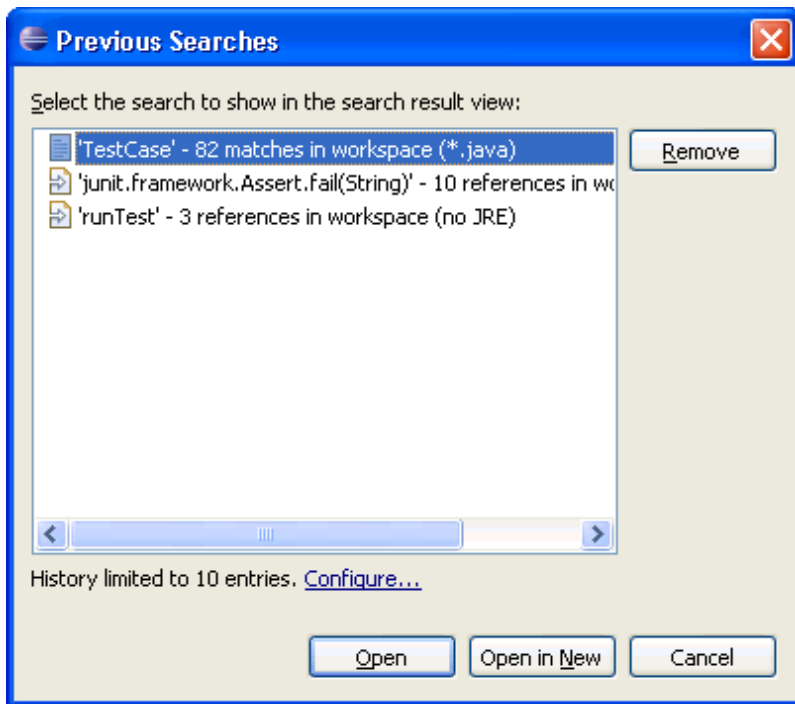
4. To find all files of a given file name pattern, leave the Containing Text field empty.

Viewing previous search results

In the Search Results view, click the arrow next to the **Previous Search Results** toolbar button to see a menu containing the list of the most recent searches. You can choose items from this menu to view previous searches. The list can be cleared by choosing **Clear History**.



The **Previous Search Results** button will display a dialog with the list of all previous searches from the current session.



Selecting a previous search from this dialog will let you view that search.