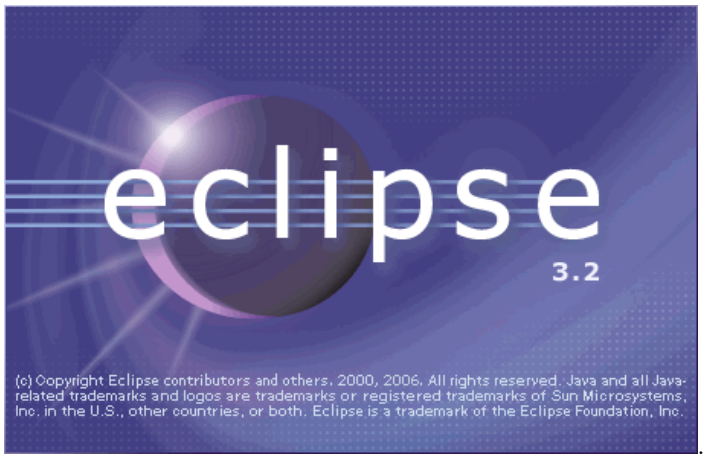
Eclipse IDE

Staring and Stopping Eclipse

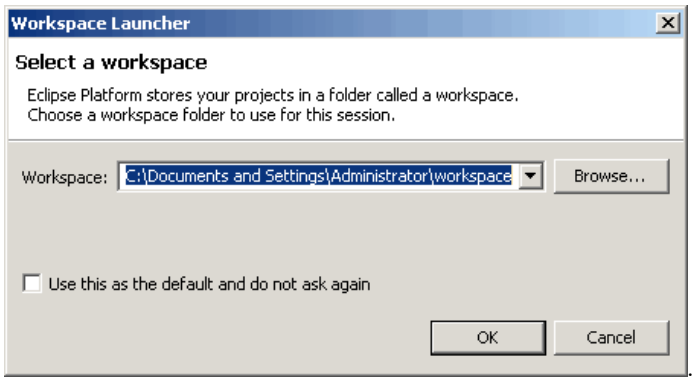
We will be using the Eclipse Integrated Development Environment (IDE) for writing, running, browsing, and debuggng our Java code. The Eclipse project itself is described on the Eclipse.org web page. It begins,

> Eclipse is an open source community whose projects are focused on providing an extensible development platform and application frameworks for building software. Eclipse provides extensible tools and frameworks that span the software development lifecycle, including support for modeling, language development environments for Java, C/C++ and others, testing and performance, business intelligence, rich client applications and embedded development. A large, vibrant ecosystem of major technology vendors, innovative start-ups, universities and research institutions and individuals extend, complement and support the Eclipse Platform.

We start Eclipse by double-clicking (a shortcut to) its icon  . While it is loading, it displays the following splash screen. Note that Version 3.2 is the most recent release of this software, the one that we will be using.



At this point, Eclipse displays the **Workspace Launcher** window. If this is the first time that we have started Eclipse, this window will display a suggestion typed in the **Workspace** pull-down box, as shown below. If we examine this pull-down box, there will be no other items in it.



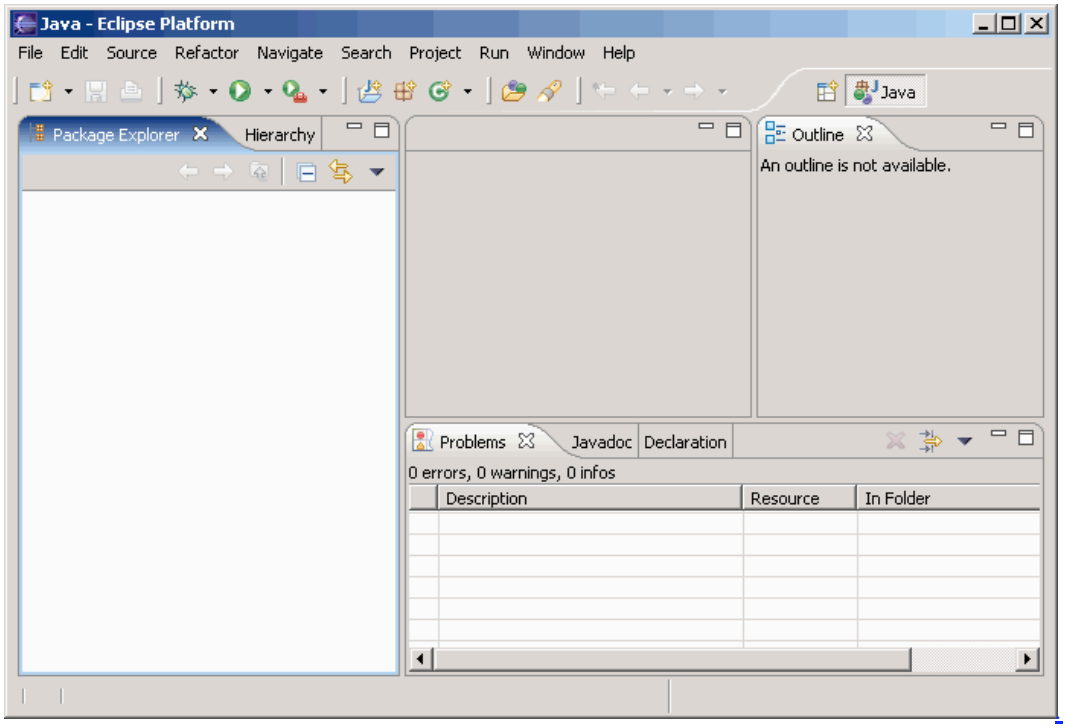
The form of the suggested workspace is **C:\Documents and Settings*username*\workspace**,

where ***username*** appears above as **Administrator**.

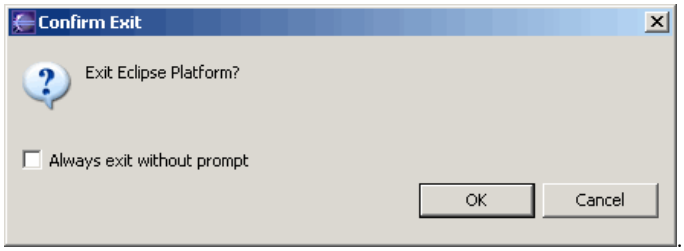
On subsequent statups, the contents of the pull-down box will default to the last workspace that we used. It is possible to create/use multiple workspaces: the most recently used ones will appear as items in the **Workspace** pull-down box, with the visible workspace the one that we used most recently. This information is stored in the **eclipse** folder (created when Eclipse was installed), in the file **configuration\settings\org.eclipse.ui.ide.prefs** (which is a text file that you can read and even edit.

If we use just one workspace (recommended), we can check the box **Use this as the default and do not ask again**, to avoid this window's prompt altogether. Or, we can leave this box unchecked, at the cost of having to click the **OK** button in this window every time that we start Eclipse.

Go ahead now and click the **OK** button to select the default work space (or **Browse...** or type in a folder name). The splash screen will disappear and Eclipse should appear on the screen in the following form (although its windows will be bigger).

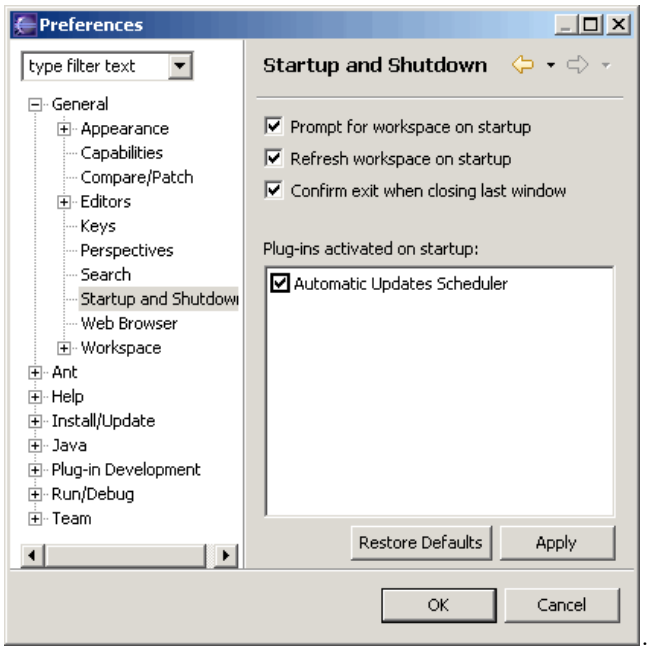


To stop Eclipse, we can either select **File | Exit** from the left-most pull-down menu, or just click the terminate window button in the upper-righthand corner. If the latter, Eclipse will prompt for confirmmation of termination with the following window.



Here too, we can check the box **Always exit without prompt** to avoid this window's prompt altogether. Or, we can leave this box unchecked, at the cost of having to click the **OK** button in this window every time that we stop Eclipse by clicking the terminate window button.

If you ever check one of these "do not prompt again" boxes, but later want to restore these prompts, you can do it as follows. Once Eclipse appears on the screen, select **Windows | Preferences**. Then disclose **General** (click its plus) and click **Startup and Shutdown**. You should ultimately see the following window.



Check or uncheck whatever boxes you desire for your preferences, then click **Apply**, and finally click **OK**.

Practice starting and stopping Eclipse, setting these prompt/confirm preferences as described above, and observing their behavior. After starting Eclipse, change the size of its window, and note that when you stop and then restart it, the window will remember the size you last specified. In fact, once we start using Eclipse for real programming, whenever we start it, it will be in exactly the same state as when we last stopped it. Therefore, it is simple to resume working in exactly the context we were in when we stopped.

Eclipse Nomenclature

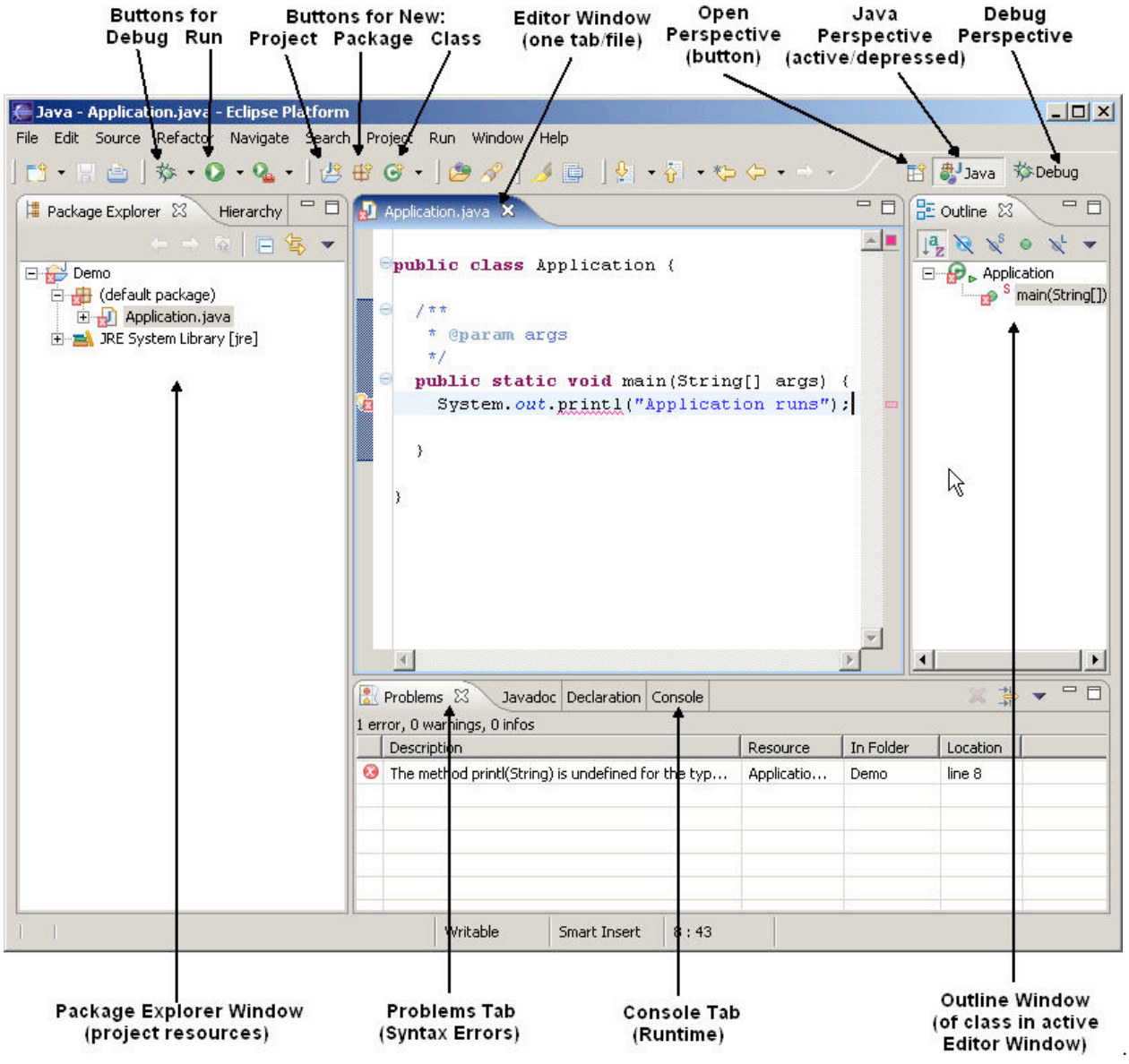
This section contains a terse description of Eclipse using **highlighted** technical terms (become familar with them) to describe its basic layout and operation. Because Eclipse is an *industrial-strength* tool, and we are using it in an academic setting (an early programming course), we will focus on its simpler aspects. The most important terms thate we will discuss and use are **workbench**, **workspace**, **perspective**, **view**, and **tool bar**.

Workbench/Workspace: These two terms are closely connected, to the point of having the same prefix. A **workbench** (or more accurately, a workbench window -see the window below) is the Eclipse interface to a **workspace**. A workspace is a folder that comprises a collection of files/subfolders that store the workspace's preferences (how the workbench window appears on the screen and how it displays/manipulates it contents) and projects (collections of related programming resources -primarily Java classes). We interact with a workspace -view and manipulate its preferences and projects- through a workbench window.

Preferences specify how a workbench window displays a workspace; projects specify the software that we can develop using the workbench window. In the section above, we started Eclipse and created a new, "empty" workspace; actually, it is not really empty: it stores all the standard initial preferences for the workbench window, but no projects. Then, the workbench window displayed this "empty" workspace.

So, a workspace stores information about each of its projects in a separate folder. It knows where all their resources are located, whether they ar inside or outside the workspace. Yes, a project folder (always stored inside a workspace) can store some or all of its resources outside that workspace. The workspace also records whether each project it contains is open or closed for use (for more details, see the section on Closing Workspaces). Finally a workspace stores preferences that apply to all its projects, and to the workbench using the workspace.

Eclipse is general: we can have any number of workbench windows open, each referring to a unique workspace, or groups of windows referring to a common workspace. For simplicity, we will always use just one workbench window, and it will always refer to the same workspace. In fact, in the following discussion we often will say "Eclipse" when we mean "workbench window": e.g. We use Eclipse to interact with a workspace. Below is an example of Eclipse using all the standard preferences, with labels affixed to many of it interesting features. The rest of the this section will explain its layout and operation.



Perspective: At any given time, Eclipse displays one **perspective** of the many that it can display. Each different perspective is suited to one specific programming task. The perspective shown above in

Eclipse is the **Java** perspective, which we use to write Java code. Notice that the name of the perspective, **Java** appears depressed (it is active) on the upper-rightmost tab. It it followed by another tab, indicating the **Debug** perspective; we can switch to this perspective being active by clicking its tab. A third perspective that we will use, less frequently than these two, is named **Java Browsing**.

View: Each perspective contains a variety of **views** that allow us to view, navigate, and edit information about a program. So, views are not just for looking: we can use views to change information too. A view may appear as a single tab in its own window, or it may be one tab in a tabbed notebook window, containing many views, of which only one is active at a time- the top one. The **Java** perspective contains a variety of standard views. Going clockwise from the top left,

- The **Package Explorer** view (it is one tab in a window, along with the less useful **Hierarchy** view tab) shows all the the code (classes and libraries, and their files) under a project name.
- An **Editor** view is one tab (per file being edited) in a window comprising only editor views; here the only file is named **Application.java** so there is only one tab in this editor window. The tab contains the name of the resource being edited; if we hover over the tab that views a class, Eclipses displays the name of the project, the package containing the class (nothing if it is in the default package) and finally the class name.
- The **Outline** view is the only tab in a window that shows a high-level outline (mostly imports, instance variables, methods, and nested classes) of the class specified in the active editor tab. The colored icons to the left of the names specify properties: e.g., access modifiers, whether or not they are overriding an inherited method.
- The **Problems** view (it is one tab in a window, containing other tabs of which **Console** is the most important) shows a list of all the errors the Java compiler found when it tried to compile a project.

Because these views are related, some information is propagated into multiple views: e.g., the red indicators that there are syntax errors in the code. When they disappear from one window, they often simultaneously disappear from the others.

Tool Bar: The workbench tool bar appears under the menu bar that includes the drop-down menus labeled **File**, **Edit**, **Source**, etc. The tool buttons here act as short cuts for common operations in a perspective; we can also invoke these operations with the pull-down menus, but these buttons are faster. The picture above labels the **Debug** and **Run** buttons, as well as the **New Java Project**, **New Java Package**, and **New Java Class** buttons (described in the next sections). The tool buttons on this tool bar change when we change the perspective. We can customize tool bars within a perspective, but we will not cover this topic here.

Starting a New Programming Project

In this section we will discuss how to start a new programming project, and how to manipulate the perspectives and views on the workbench window displaying that project. So, start up Eclipse as described above.

Before starting any project, we should set the preferences for Eclipse to show (and default to) the **Java** perspective. Generally, we can manipulate perspectives in a few interesting ways.

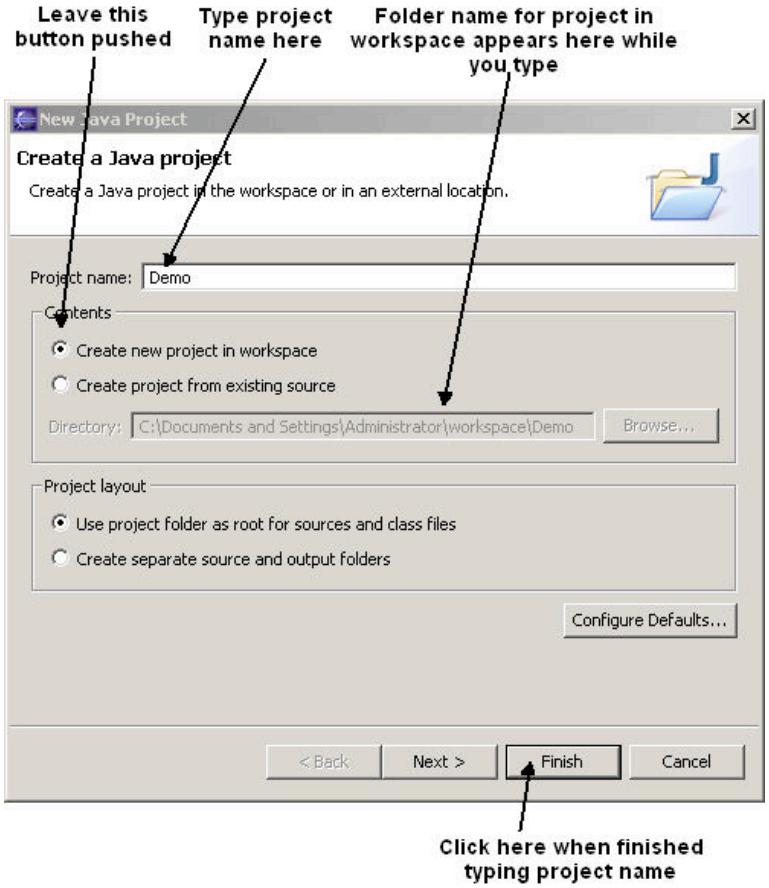
- We can remove a perspective from our view by right-clicking its tab and selecting **Close**.
- We can bring back a closed perspective ***P*** by selecting **Window | Open Perspective | *P*** or by clicking the **Open Perspective** button and selecting ***P***.
- We can drag the left edge of the tab holding the perspective left/right to lengthen/shorten it.
- We can elide the perspective names, just displaying their icons, by right-clicking any perspective and selecting **Show Text** (toggling whether it is shown).
- We can rearrange the order of the perspective tabs by dragging each to its desired location.
- We can dock these tabs at the Top Right, Top Left, and Left of the Ecplipse window by right clicking any perpsective tab and selecting **Dock On**, and then the appropriate location.

So, if the upper-righmost tab in Eclipse shows the **Resource** perspective (the initial default in Eclipse)

- Click the **Open perspective** button appearing before it on the tab and select the **Java** perspective.
- Right-click the **Resource** perspective and close it.

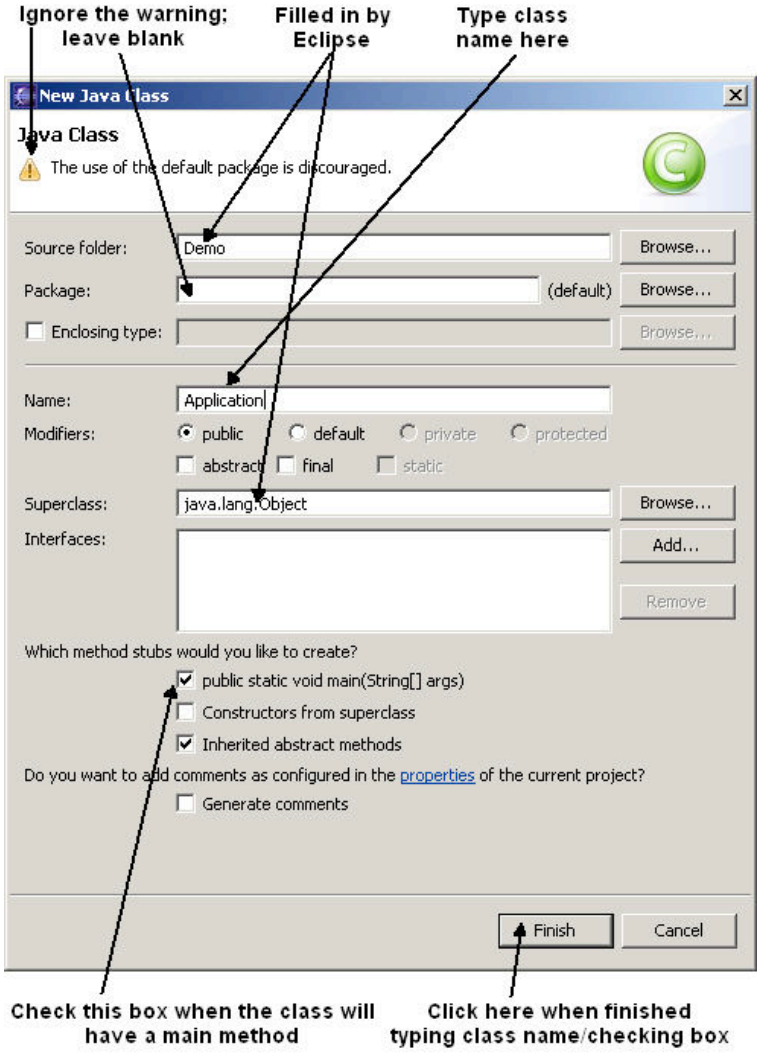
If you want to experiment with any of the other operations on this tab, please do so now; you should be able to undo whatever actions you perform, and ultimately restore and activate the **Java** perspective.

To start a new project, click  the **New Java Project** button on the tool bar for the **Java** perspective. The following **New Project Window** will appear.



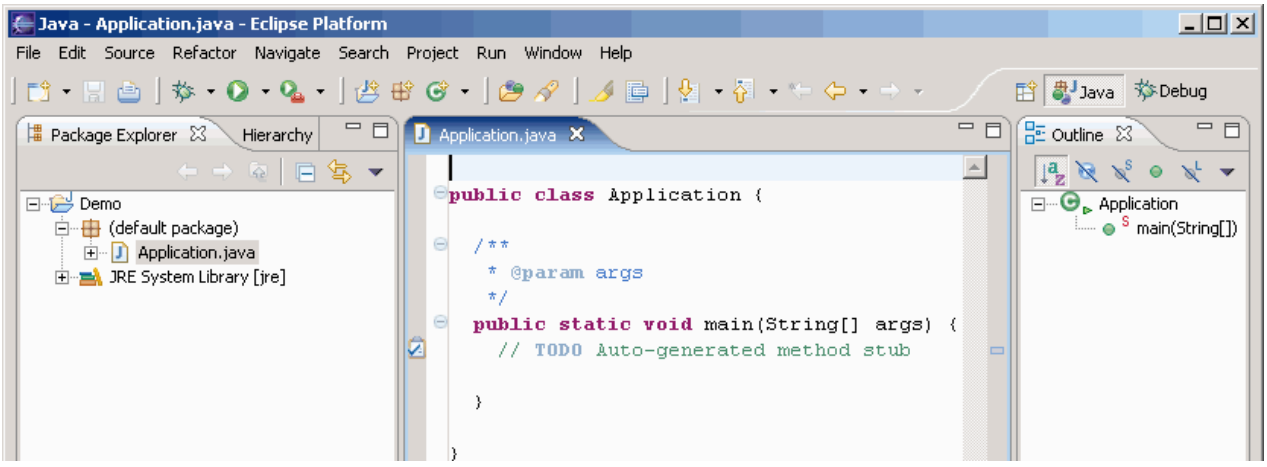
In this window I have created a project whose name is **Demo**; this project is stored in a folder named **Demo**. Eclipse enforces this name convention. For a simple project, just click **Finish** when you are done typing the project name; clicking **Next** leads to another window of options, which we would default anyone and are ultimately accepted clicking **Finish**.

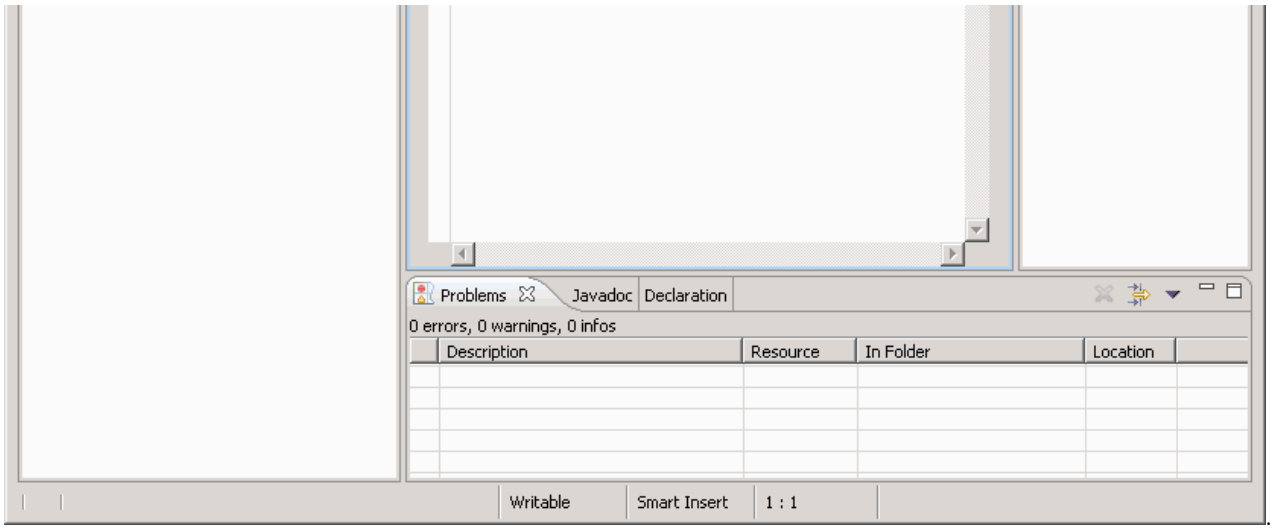
To create a new class inside this project, click  the **New Java Class** button on the tool bar for the **Java** perspective. The following **New Class Window** will appear.



In this window I have created a class whose name is **Application**, inside the anonymous package, inside the project named **Demo**. I have checked the appropriate box, so Eclipse will automatically include a **main** method for the **Application** class. Click **Finish** when you are done typing the name and checking the box.

Eclipse will update to look like the following.





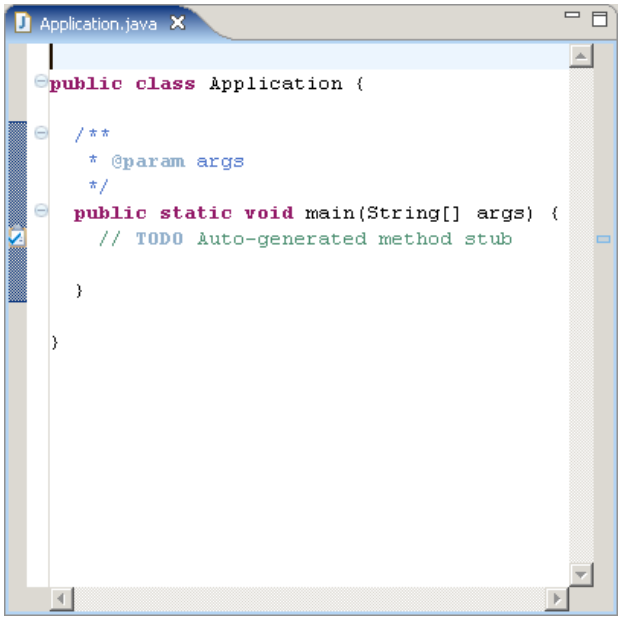
Editing a Class File

We edit classes in editor windows, mostly using standard computer editing commands. Click the **Edit** pull-down menu for a list of commands and their keyboard shorcuts. Also see the **Source** pull-down menu for additional commands involving identation, (re)formatting, commenting, and generating/organizing the members of a class.

Whenever we create a new class, or double click any class in an **Package Explorer**, Eclipse will add a view/tab for editing that class in the editor window (if such a view is not already present). To add a second editor view/tab for a class -so that we can view two parts of it simultaneously- we can right-click its view/tab and select **New Editor**. Within an editor window, we can manipulate views/tabs as described above, including dividing one editor window into two editor windows, vertically or horizontally.

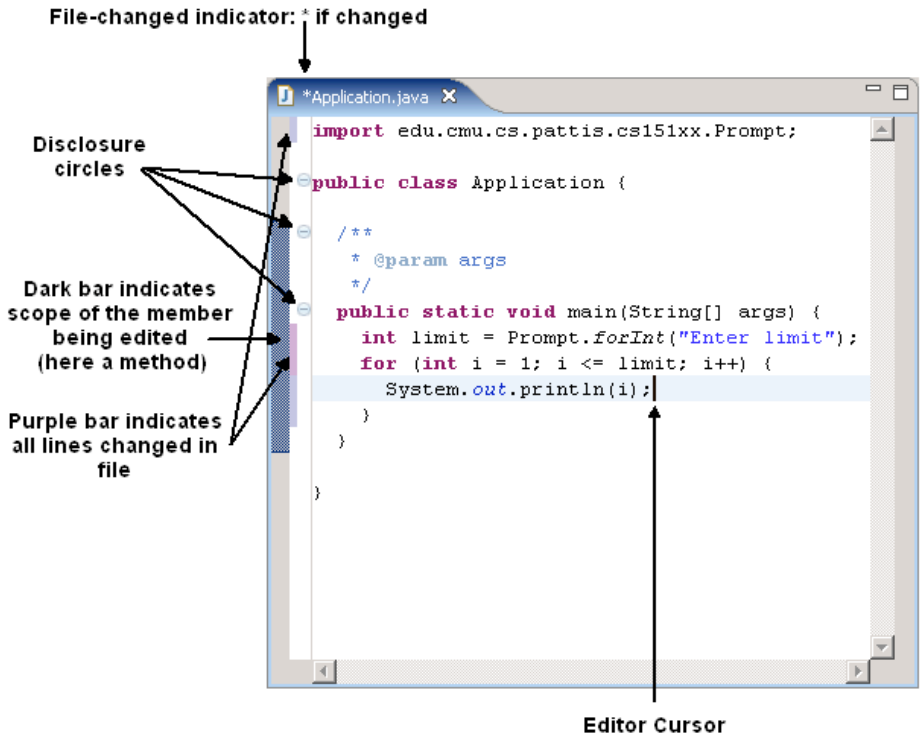
Note that in this editor, a single click positions the cursor; a double click selects a token; dragging the cursor selects multiple characters/lines.

After creating a new class, the editor window looks like as follows. Practice selecting text (characters, tokens, lines) in this window.



Now we can edit the file. Below, we completed the **main** method. Afterwards, the window appears as

follows.

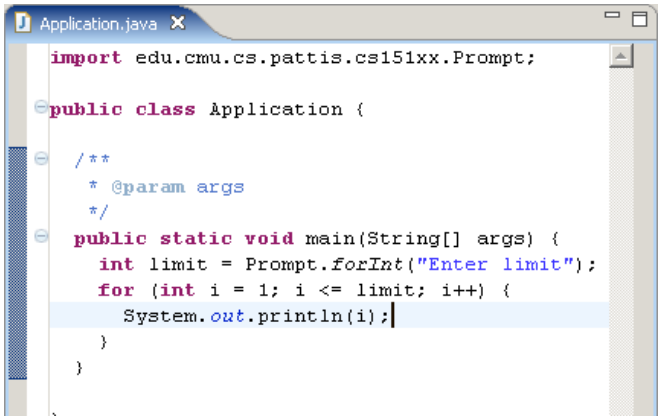


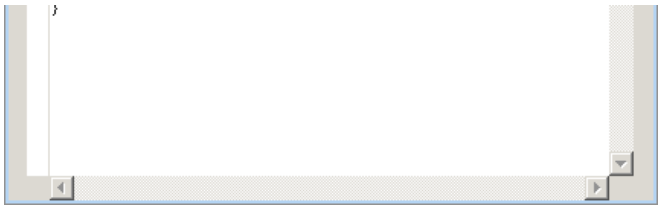
Note the annotations indicating

- the file has changed (and which lines have changed; if the change is one or more deleted lines, a small horizontal line appears where the lines *used to* be)
- the scope of the member being edited (and the cursor in that member)
- the disclosure circles for other members

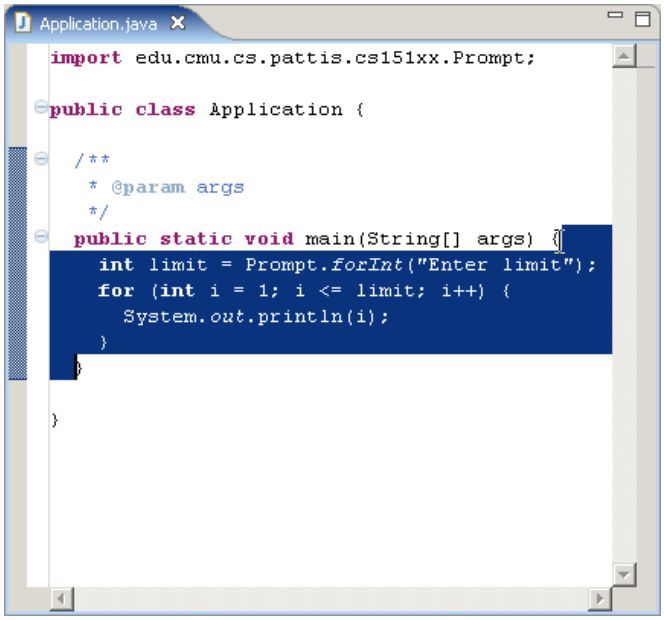
Note the disappearance of the file-changed asterisk and the purple change bars. Often, if we forget to save a file, and perform some othe operation in Eclipse -such as try to run a program- Eclipse will ask us whether to save the file(s) first.

To save the changed contents of an editor view to a file, right-click in the view and select **Save** (this word will not even appear as a selection, unless the file has been changed: see the change indicator). After issuing this command, the file is updated to contain all the information currently in the view for this class. If instead we selected the **Revert File** command, the view would would be restored to the file's contents when it was last saved. So, use **Save** and **Revert File** carefully. If we issued the **Save** command in the window above, it would be updated and appear as follows.

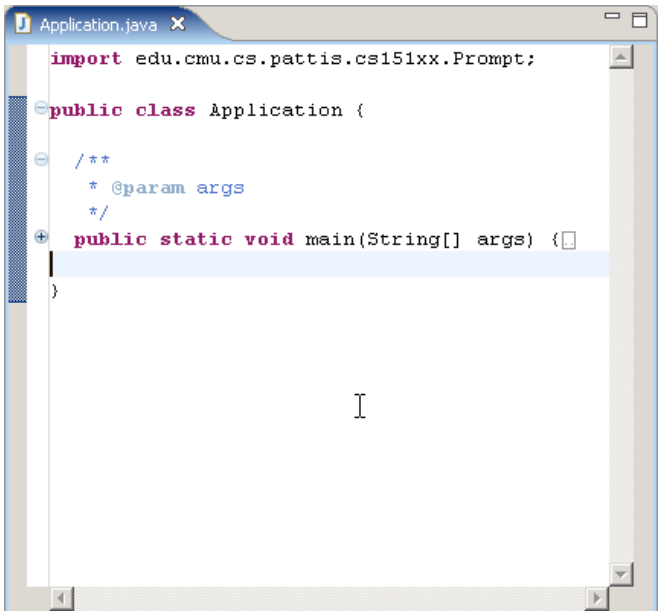




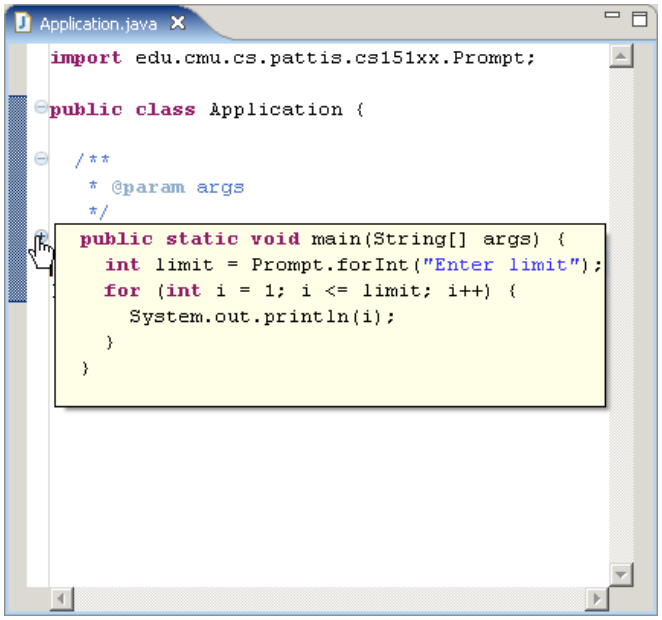
In this editor, whenever we type an opening delimiter -a brace, parenthesis, or quotation mark- the editor automatically supplies the matching closing delimeter, and then repositions the cursor between the two so that we can type the delimited entity. Likewise, whenver we double-click directly after an opening delimiter or directly before a closing delimiter -a brace, parenthesis, or quotation mark- the editor automatically highlihts all information between the matching delimiters. Below I double-clicked just to the right of the opening brace in the **main** method.



We can elide the method we just typed from the view by clicking its disclosure circle: - means it is disclosing (clicking elides); + means it is eliding (clicking discloses).



In fact, if we hover over a disclosure circle that is eliding, the cursor will change to a hand and Eclipse will display the elided material.

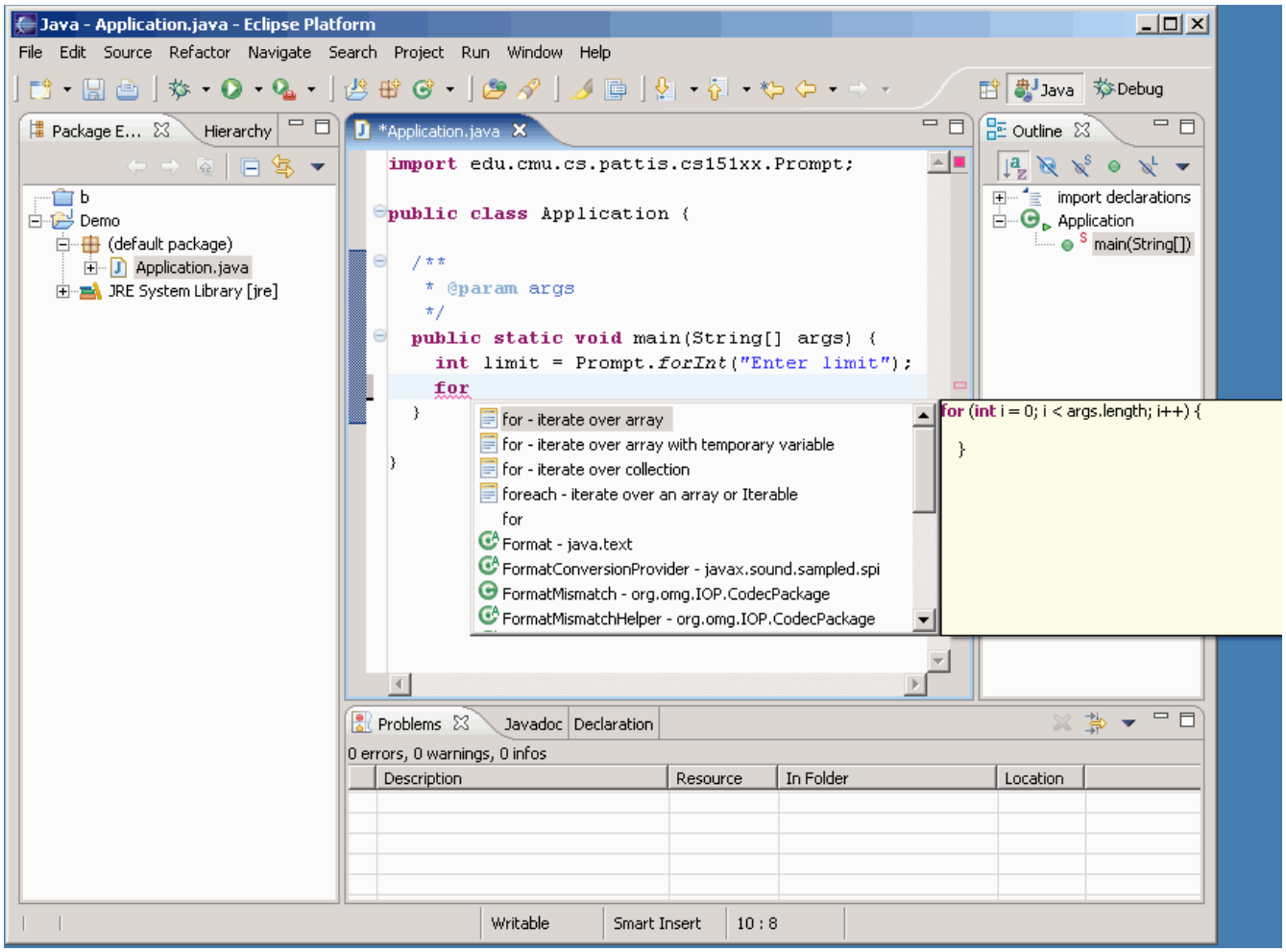


In complicated classes, with many members, we can use the **Outline** view to rapidly examine a selected members in the class: jump to its definition. Although not needed here, we can click among **import declarations**, **Application**, and **main** in the **Outline** view, and observe how the editor view changes, moving the cursor and highlighting the selected tokens. In fact, if we position the cursor inside some code, the **Outline** view updates to highlight its display of the member that we are editing -sometimes even self-disclosing the member, if it is nested in another member. Also observe the changes in the bar indicating the scope of the member being edited.

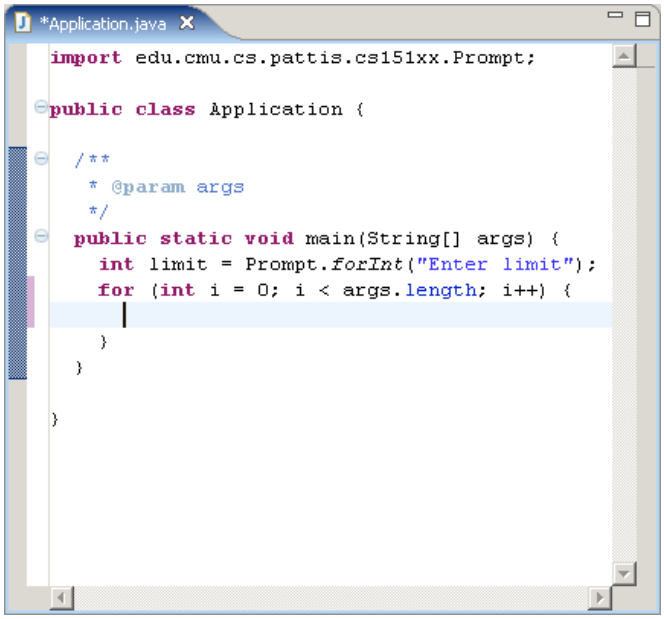
Finally, the **ctrl/space** command in Eclipse generally means "help me complete what I am typing". Two useful examples are:

- If you are in the middle of typing the name of a method call, it can help you complete the name, while illustrating its various prototypes.
- If you are in the middle of typing a keyword, it can complete the keyword. If the keyword starts a control structure, it allows you to choose among various templates for that control structure, and automatically outlines the necessary code when you have made your selection.

Here is an example of typing **ctrl/space** after the keyword **for**.

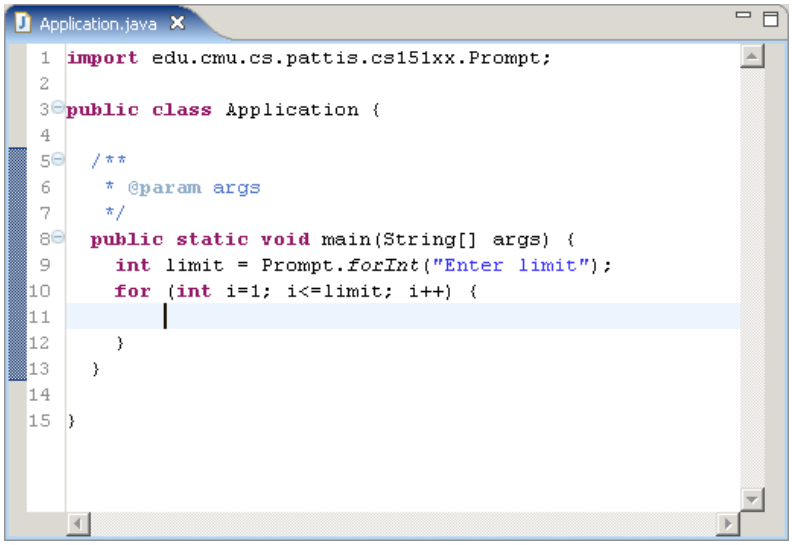


After selecting this first **for** option, the editor view shows the following **for** template, whose outline is now ready to be edited/corrected



One last item. To get line numbers for a file (useful when exceptions are thrown by a running program),

right click in the gray left margin and click **Select Line Numbers** to toggle it. Doing so in the above window leads to it displaying as follows.



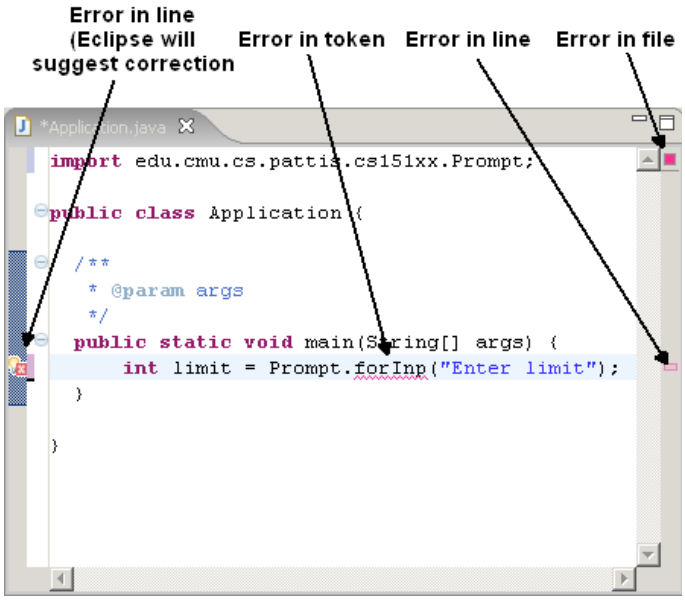
Notice the number are right aligned.

Practice all these operations until you are comfortable entering and navigating code in an editor view.

Correcting Syntax Errors

The process of debugging involves first correcting syntax errors, so the program will compile correctly (and be runnable), and then correcting execution errors, so that the program will run correctly. This section discusses only how to debug syntax errors, using the standard **Java** perspective; a later handout discusses how to debug execution errors, using the **Debug** perspective.

The universal color for a syntax error indicator in Eclipse is red; the universal indicator for a syntax error in Eclipse is a red backgound (often a box, sometimes a circle) containing an X (e.g.,

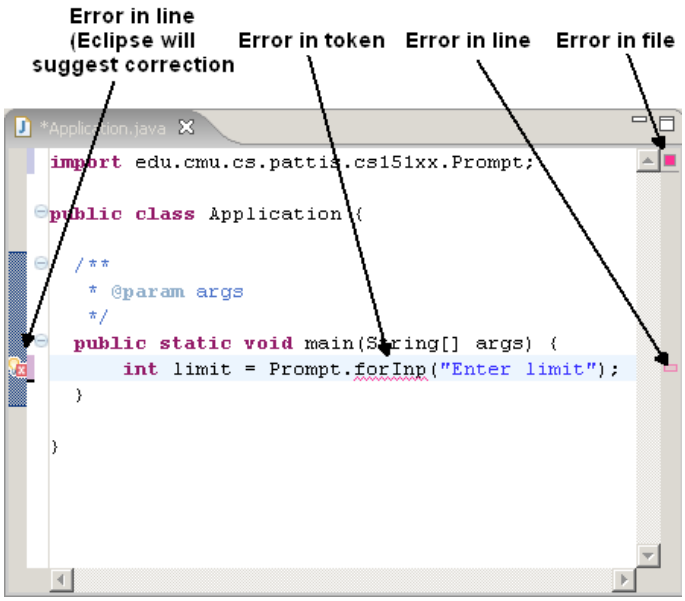


). Often a single syntax error results in multiple red boxes appearing in multiple views. Likewise, the universal color for a warning indicator in Eclipse is yellow. You must stop and fix an error, but you just need to be cautious with a warning (as with driving signs).

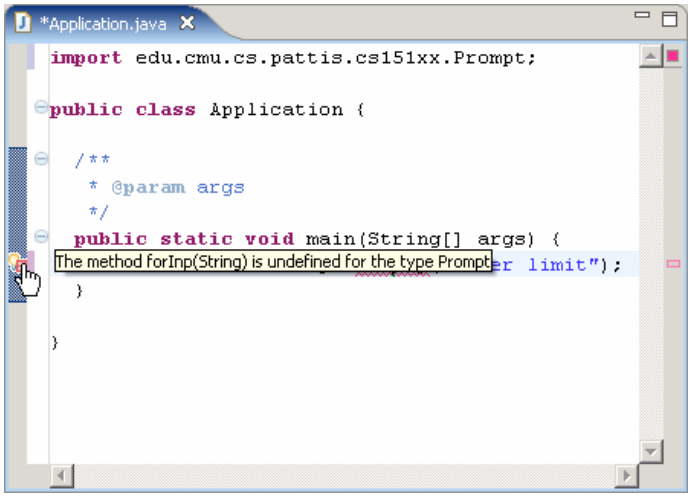
Note that if you click the **Project** pull-down menu, the item **Build Automatically** should be checked: if it isn't, click it, and the next time you click this pull-down menu it should be checked

(clicking toggles it).

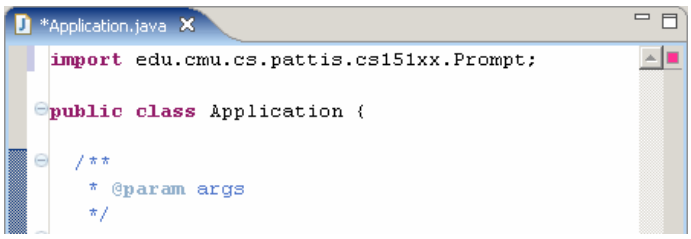
Eclipse tries hard to spot syntax errors while you are typing/editing your code, and help you fix them immediately. For example, after we have included the **import** statement in the first line, and entered the first line in the **main** method, the editor view reports a syntax error because we misspelled **forInt** as **forInp**. Notice that there are 4 error indicators in this editor view. The error indicator in the left margin signals that not only has Eclipse detected a syntax error, but it has a suggested fix for this error.

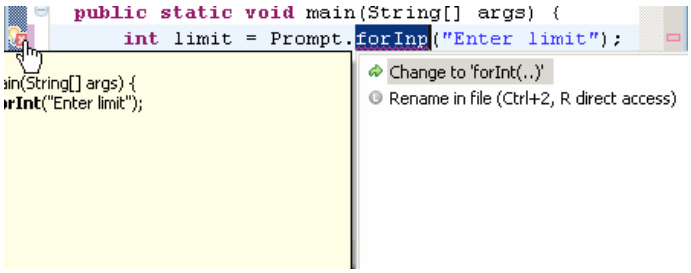


If we hover over any of the 3 error indicators on the line, Eclipse will display a syntax error message for that line; below, we illustate hoving over the error indicator in the left margin.

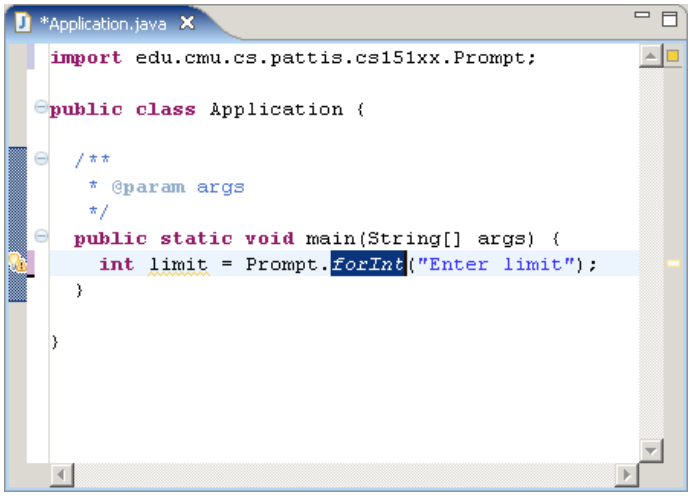


In fact, if we click on this error indicator, Eclipse suggests possible fixes.

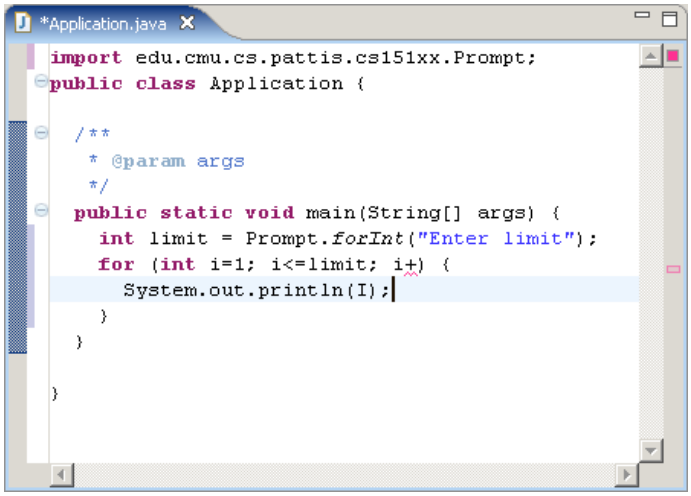




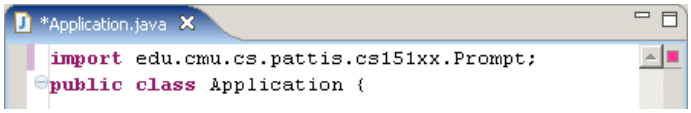
If we then double-click on the first suggestion, Eclipse will correctly fix this error and all 4 error indicators will disappear, but they will each be replaced by an equivalent warning indicator. This is because I have configured one of the preferences in Eclipse to warn whenever I define a local variable (**limit**) whose value is not used subsequently in the method. There is no real error; I just haven't written that code yet.



Now assume that we type the next two lines and make a mistake on each: on the first line we write **i+** instead of **i++**; on the second we write **I** instead of **i**. Then Java will show only a token error on the first line.



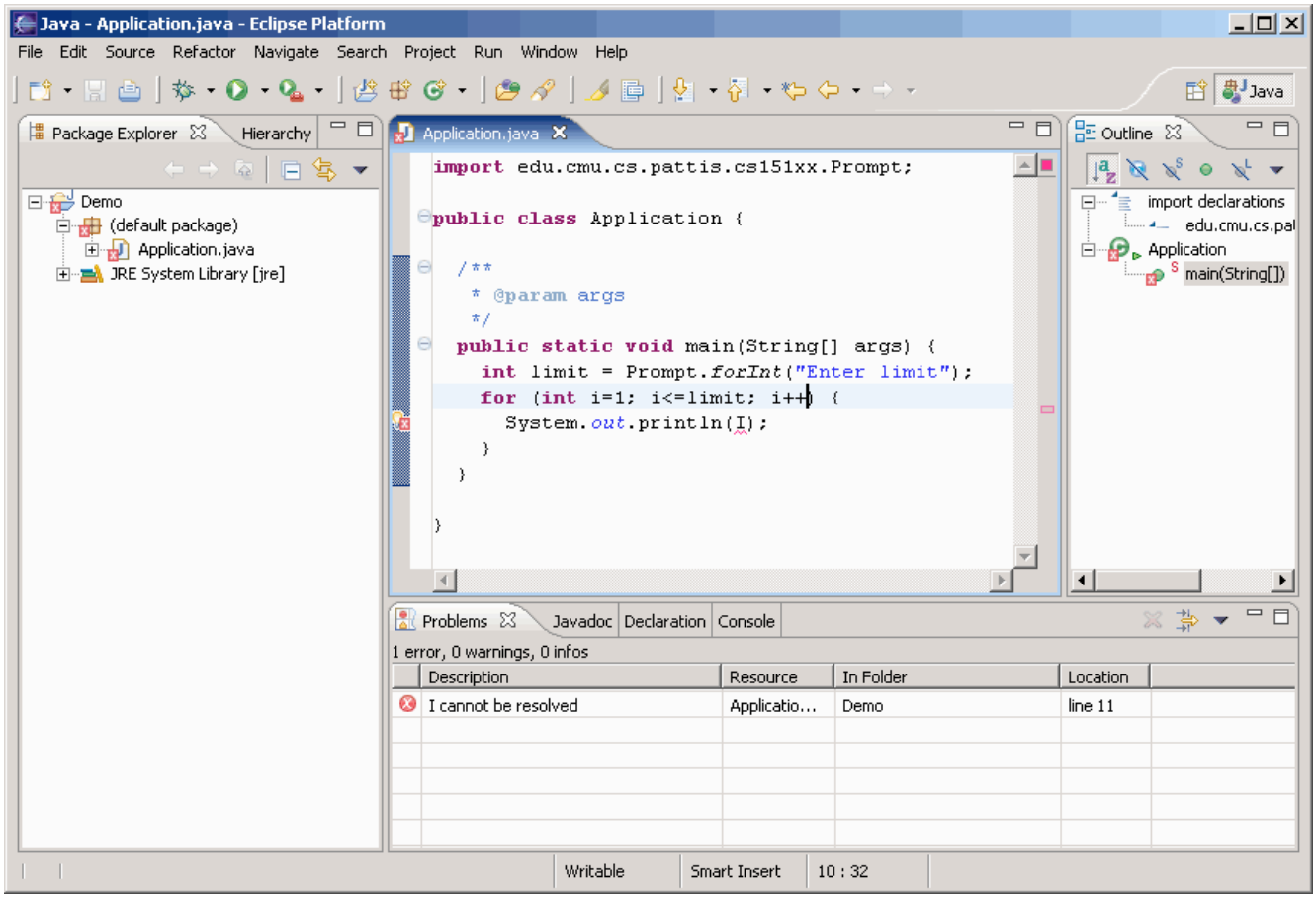
When we fix this first error, then the second error immediatley appears.



```
  /**
   * @param args
   */
  public static void main(String[] args) {
    int limit = Prompt.forInt("Enter limit");
    for (int i=1; i<=limit; i++) {
      System.out.println(I);
    }
  }

}
```

If we save the file now, Eclipses not only saves the file, but it also compiles it. After compiling this class (and still finding one error) all the views are updated to contain error indicators.

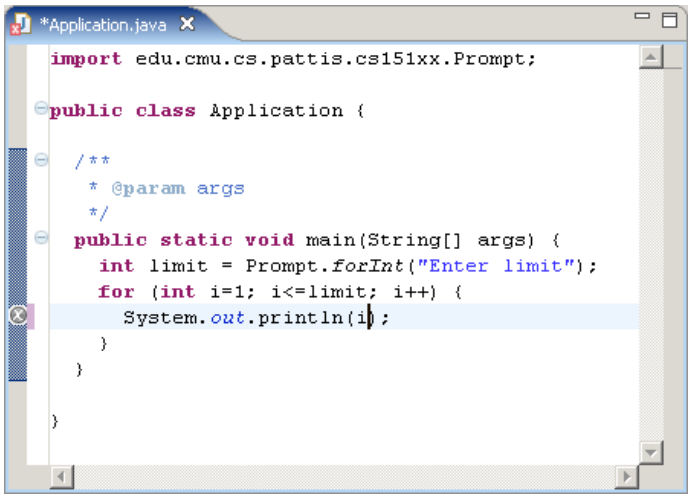


First, note the file-changed indicator (*) and the purple change bars in the margin have disappeared. The result (now showing in all views) is that syntax errors are marked in the **Package Explorer** and **Outline** view (as well as in the tab for the editor view for this file, and finally the **Problem** tab at the bottom includes the error as well.

We can visit and correct each syntax error in the editor view by clicking on its line in the **Problems** view (generally all the syntax errors appear on these lines). We can also resize each of the columns in the **Problems** view. In addition, we can click on any red boxes in the right margin of the editor view; sometimes clicking the disclosure circles to the left of the code will make this operation easier.

If we correct the final error in the editor window, only the editor view changes: its error indicator in the left margin turns from red to gray. All the other error indicators in that view -and in the other views-

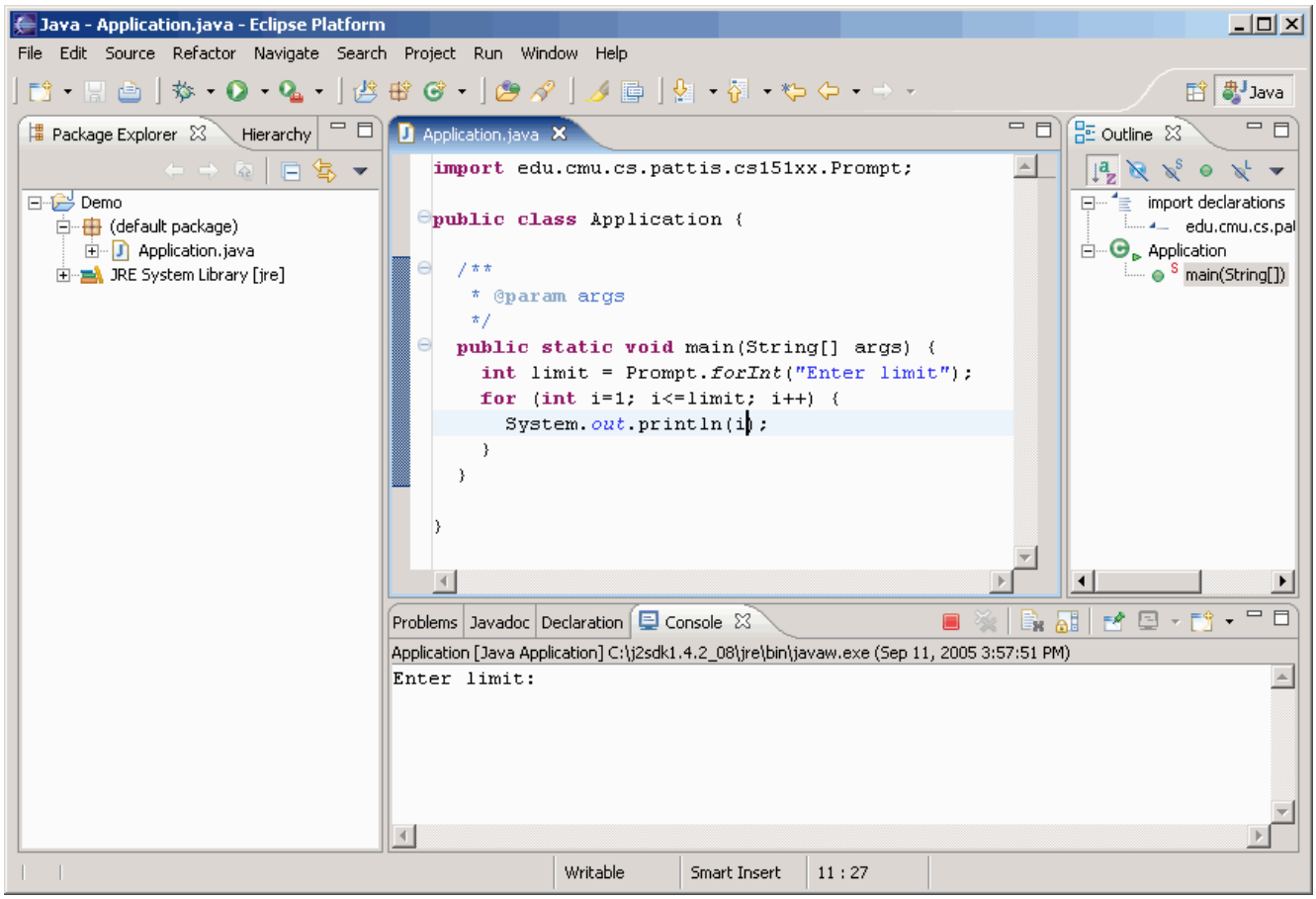
stay the same.



Finally, we can perform two interesting operations here.

- If we save the program, Eclipse will recompile it and all the error indicators will disappear. We can then run the program.
- If we run the program immediately (before saving it), Eclipse will first prompt us to save the file (see the previous section) and once we agree, then it will recompile this file and all the error indicators disappear, and finally it will run the program.

In either case Eclipse will create a console view tab and run the program in it, as illustrated below.



Therefore, there are at least two ways to compile a class.

- Save the file storing the class by the right-click **Save** command in an editor view of that class.
- Run (or just attempt to run) a **main** method in some class (see the previous section for details).

In either case, Eclipse tries to compile all classes in the project that need to be compiled.

The interactions among saving, compiling, and running can sometimes be a bit subtle and confusing. Confusion especially arises concerning the synchonization of the files you are editing and Eclipse's reporting of syntax errors in various views. For example, if we fix an error, its associated error indicatators may disappear from some -but not all views; we must sometimes perform further operations At time, recompiling by saving hasn't done what I've expected, and I've had to recompile by trying to run. There is no substitute for using Eclipse repeated to better understand its operation

Advice for Debugging Syntax Errors

Interpret syntax errors/warnings liberally.

- When the Eclipse detects a problem at a token, often either that token is wrong, or the one before it is wrong (less frequently the incorrect token appears even earlier in the program, and even less frequently later). Check both places.
- Sometimes an error/warning message makes no sense: it says that there is one kind of problem in our code, but we actually have to correct a different kind of problem.
- Problems snowball: one mistake can cause many error/warning messages to appear; by fixing one mistake, many message may disappear.

To debug the syntax errors in a program

- Fix the first problem (or if you are confident, the first few) **Problems** view. If the first error makes no sense, a subsequent error might be causing the problem (snowballing). Always find and fix at least one error; don't spend time fixing more than a few (because recompiling is so quick).
- Recomple the program.
- Repeat this process until there are no syntax errors left.
- To make progress, you must correct at least one error during each (re)compilation. And, fixing only one error at a time ensures that you won't get confused by snowball errors.

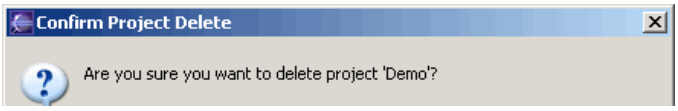
Finishing a Project: Closing/Removing It

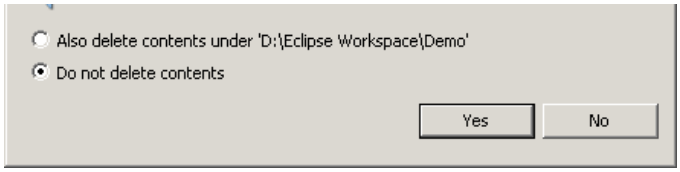
As we have seen before, when we terminate Eclipse, it saves all the preferences and projects that appear in our workbench. When we restart Eclipse, it initializes itself in exactly the same state as when it terminated. Therefore, we can seemlessly start Eclipse, create/work on our projects, stop it, restart it, and continue working on our projects just where we left off.

Now let us explore three options for finishing a project. We need to find out what happens to such projects both on our workbench and in our workspace. The simplest option is to just leave a project alone. It will continue to be present on our workbench and in our workspace. This is a safe option, although our workbench can be come cluttered, and we might accidentally change a project that we meant to leave alone. So, we will explore two other options: closing and deleting.

Closing: First, we can just close the project. Closed projects still appear in the **Package Explorer** (and **Navigator**) view in our workbench (and are still stored in our workspace folder), but with no disclosure square: we cannot examine, run, or modify our code in such projects. We close a project by right-clicking its name and then selecting **Close Project**. One of the few operations that we can perform on a closed project is to reopen it: by right-clicking its name and then selecting **Open Project**. Once this is done, we can explore the contents of the project again. Another operation is removing it from the workbench, which is discussed next.

Deleting: Our second option is actually removing a project from our workbench. In this case, the project disappears altogether from our workbench, as if it were never there. We delete a project by right-clicking its name and then selecting **Delete**; at this point, Eclipse displays the following confirmation window.





If we leave the **Do not delete contents** button pressed, Eclipse removes the project from its workbench, but the folders and files containing all its resources remain intact, whether inside or outside the workspace. We can leave this information where it is, or copy/move it elsewhere; at a later time, if we want to recreate the project, it will be easy to do if we still still have all this information. But, if we press the **Also delete contents under ....** button, then Eclipse not only removes the project from its workbench, but also deletes the folders and files containing all its resources, whether inside or outside the workspace.

Obviously this latter choice is very dangerous, and I recommend NEVER using this button, which always defaults to **Do not delete contents**. If need be, we can use the **Do not delete contents** button, and after we are done, we can delete the project folder (whether inside or outside the workspace, or archive it by moving it to any other directory on our computer.