

# Beginning Eclipse Tutorial

## Eclipse 3.2 Edition

### Table of Contents:

Introduction .....	2
Download the JDK .....	4
Install the JDK.....	7
Set environment variables .....	10
Download Eclipse .....	13
Validate Eclipse download.....	16
Install Eclipse.....	18
Set Eclipse preferences .....	21
Create an Eclipse project.....	29
Create Java packages .....	32
Create Java classes.....	35
Explore a class.....	38
Declare Person instance variables.....	40
Create a constructor .....	44
Declare Address instance variables.....	48
Object association.....	50
Write application code.....	53
Run the application .....	60
Examine application output.....	65
Override toString() .....	67
Refactoring .....	72
Debugging .....	75
Searching.....	80
Conclusion .....	84

### Change Log:

2006/07/16 – Released.

2006/09/24 – Updated. Added Debugging section.

2007/04/02 – Updated for Java SE 6. Added Searching section.

2007/07/02 – Updated for changes to Eclipse download page.

## **Introduction.**

Welcome to this tutorial! If you are new to Java™ and the Eclipse™ IDE (Integrated Development Environment), this tutorial is for you.

This tutorial will not only introduce you to Eclipse and basic Java, but will be a pre-requisite tutorial for other releases in our series. In this tutorial you will learn how to download the software you need to get a Java project started, how to install that software, how to configure the Eclipse IDE, and how to develop and run a small application within Eclipse. Our primary focus will be on Eclipse navigation, but you will learn some Java along the way.

Keep in mind that this tutorial will use the **Windows® XP** operating system, so as you go please make any necessary adjustments for your own OS.

For this tutorial we must download the Java SE JDK (Java Standard Edition Java Developer's Kit) and Eclipse. Be aware that the JDK and Eclipse downloads are rather large (approximately 56 MB for the JDK and 78 MB – 140 MB for Eclipse, depending on the package) so a fast Internet connection is desirable. Furthermore you may want to temporarily shut off pop-up ad blocking software in your browser in order to facilitate an uninterrupted download process.

If you don't have a fast internet connection, you might want to ask a friend who has a fast connection to download the software for you and give it to you on a portable drive.

Before we move on to the first task, please be reminded of the following:

*"This tutorial, which may be composed of several parts as well as supplementary resources, may not be reproduced or distributed in any form, in whole or in part.*

*This tutorial, which may be composed of several parts as well as supplementary resources, is sold as-is, without warranty or conditions of any kind, either expressed or implied. We make no guarantee of the fitness of the techniques, configuration, code, or any other aspect of the tutorial, for any use to which these may be put. Neither the author, nor ArcTech Software LLC, nor any of its agents, will be held liable for damages, caused or alleged, directly or indirectly, by the use of any material in this tutorial.*

*Eclipse™ is a trademark of Eclipse Foundation, Inc.*

*Microsoft® and Windows® are registered trademarks of Microsoft Corporation in the United States and other countries.*

*Sun, Sun Microsystems, the Sun Logo, Java, J2SE, JDK, J2EE, and Java EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.*

*All other brand and product names may be trademarks or registered trademarks of their respective holders.*

*When we are aware that a word or phrase designates a trademark, we have made a reasonable attempt to capitalize the first letter of each such word."*

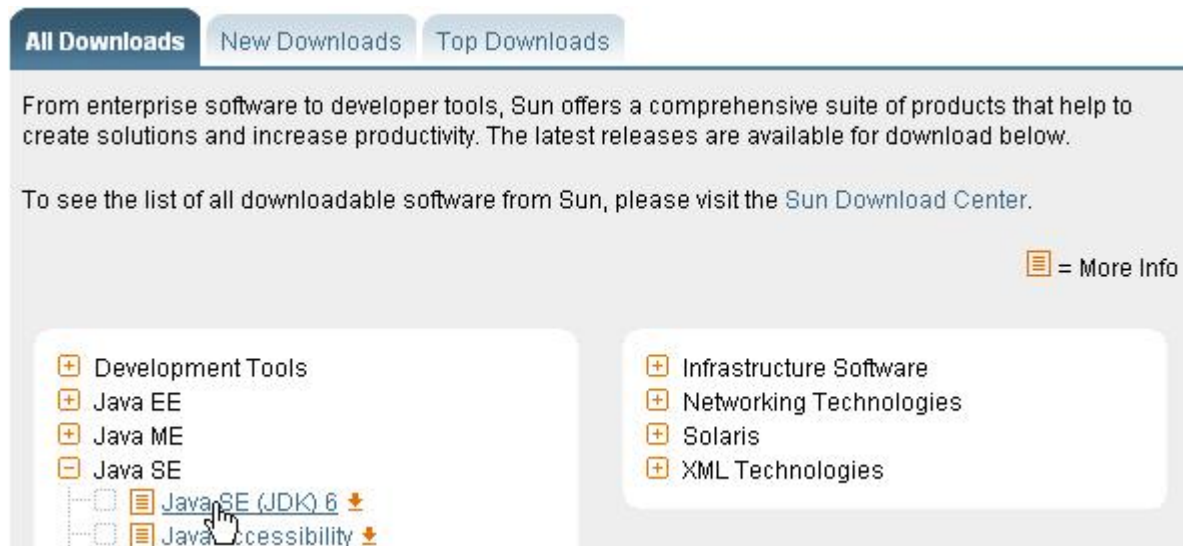
## Download the JDK.

For our first step in this tutorial, we will download the Java SE JDK (Java Standard Edition, Java Developer's Kit.) If you already have a recent Java SE installation on your machine, this step is optional, but please pay close attention to subsequent tasks to make sure that you don't miss any required configurations.

The JDK contains the core Java API to allow you to write Java applications. Navigate your browser to the Java downloads page at the following URL:

<http://developers.sun.com/downloads/>

At the time of this writing, the latest Java SE option is "Java 6" (note that this is subject to change) so click the *link* to make this selection. We will point out that recent Eclipse versions have been tested on reference platforms that use Java 5 (or 1.4.2), and consequently the current Eclipse Europa release recommends a Java 5 JRE for its IDE. This tutorial uses Java 6 because it is the featured and latest release of Java, but previous releases are still available. However all project teams should make a well-reasoned choice of a Java version based on their specific project context and requirements.



On the next page click the button to download the JDK. Note that at the time of this writing, the current release is JDK 6 Update 1, but again this is subject to change.

### JDK 6u1

The Java SE Development Kit (JDK) includes the Java Runtime Environment (JRE) and command-line development tools that are useful for developing applets and applications.

» More info about Java SE 6u1 ...

[Installation Instructions](#) | [ReadMe](#) | [ReleaseNotes](#) | [Sun License](#) | [Third Party Licenses](#)

» Download

Click the “**Accept License Agreement**” radio button (then wait for the page to reload), and click the download that is appropriate for your operating system. Notice that in this tutorial we choose “Windows Offline Installation, Multi-language.”

**Required:** You must accept the license agreement to download the product.

☒ **Accept** License Agreement | [Review License Agreement](#)  
☐ **Decline** License Agreement

**Windows Platform - Java(TM) SE Development Kit 6 Update 1**

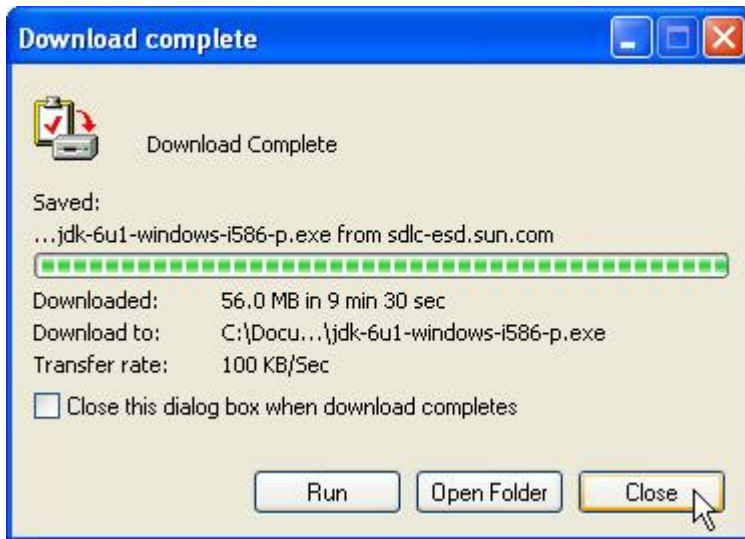
[Windows Offline Installation, Multi-language](#)

[Windows Online Installation, Multi-language](#)

In the “File Download” box, click “**Save**”, and in the subsequent “Save As” box navigate to a folder where you would like to keep the download, and click “**Save**” again.



Wait for the download to complete, and only then close the File Download dialog.



## Install the JDK.

Now that you have successfully downloaded the Java SE JDK, it is time to install it.

To install the JDK we previously downloaded, navigate to the folder into which you saved the download. If you followed the steps above you should have a file named something like this (depending on the current version):

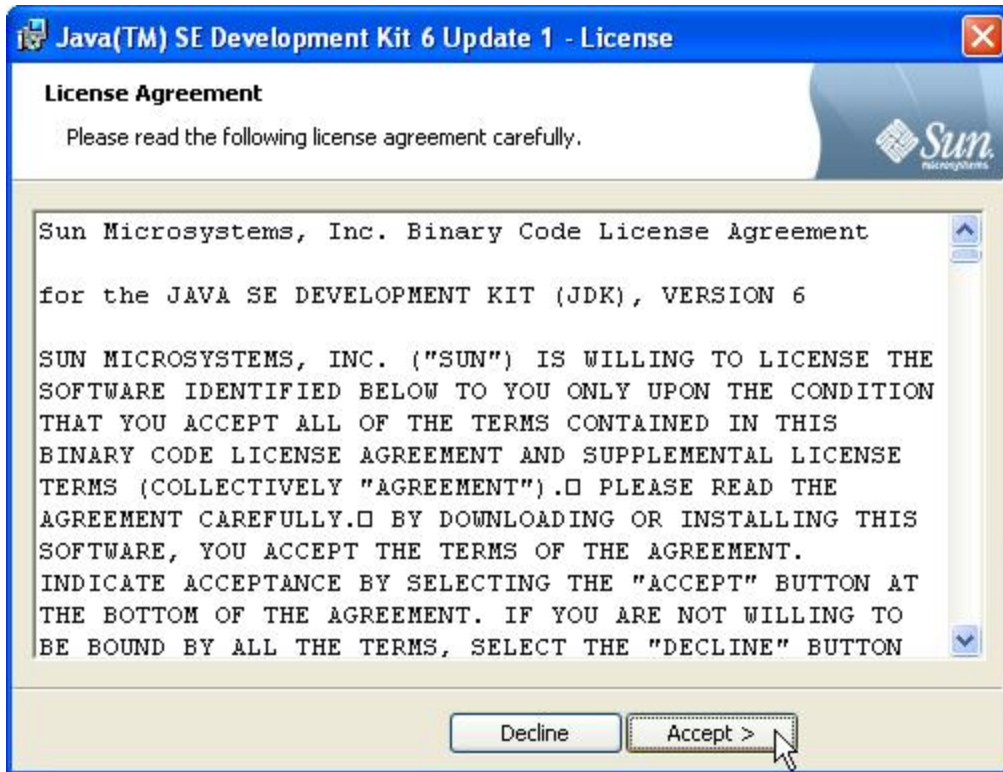
jdk-6u1-windows-i586-p.exe

If you so desire, you may want to consider scanning the file for viruses with your anti-virus software.

Close any other open programs then double-click this “exe” (executable) file. In the “Open File – Security Warning” dialog box note that the publisher is “Sun Microsystems, Inc” and click “**Run**” to execute the installation.

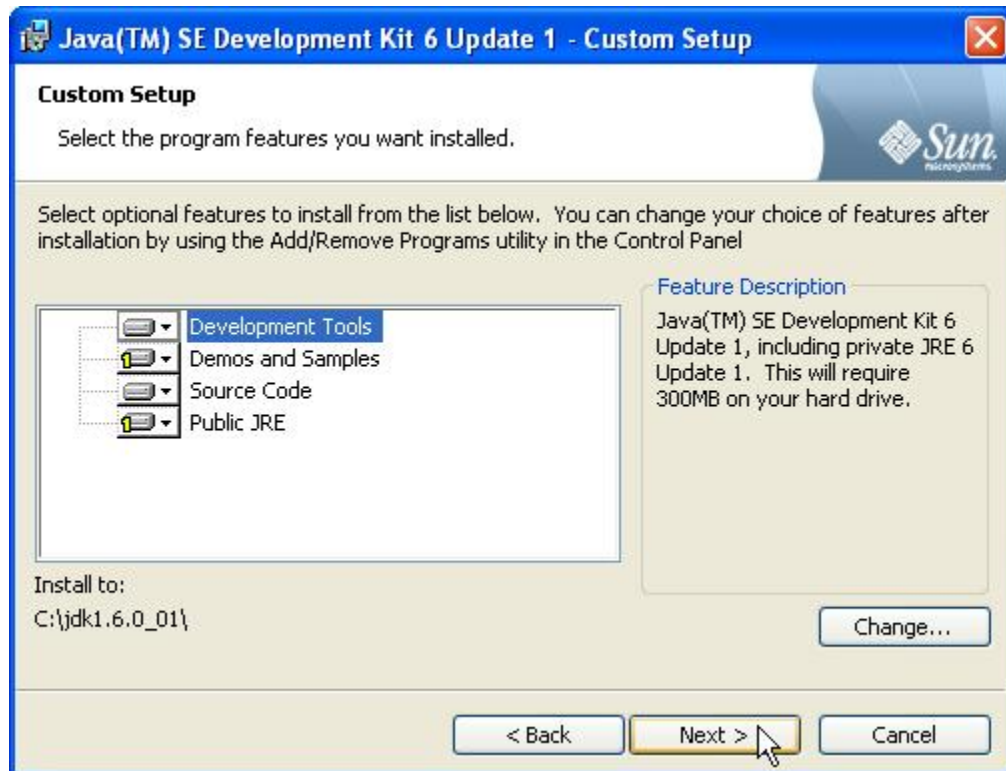


Wait for the installation wizard to initialize and in the “License” dialog box read the license if you so desire, and click “**Accept**”.





Now you come to the “Custom Setup” dialog box.



Follow these instructions, in this order, so that we install only what we need for this tutorial:

- Change the installation directory if you desire. Note for future reference that we have changed the directory to **C:\jdk1.6.0\_01**.
- Click the drop-down arrow beside **“Public JRE”** and select **“Don’t install this feature now.”**
- Click the drop-down arrow beside **“Source Code”** and select **“This feature will be installed on local hard drive.”**
- Click the drop-down arrow beside **“Demos”** and select **“Don’t install this feature now.”**
- Click the drop-down arrow beside **“Development Tools”** and select **“This feature will be installed on local hard drive.”**
- Click **“Next>”**, and wait for the installation process to complete. In the “Complete” dialog box, click **“Finish”**.

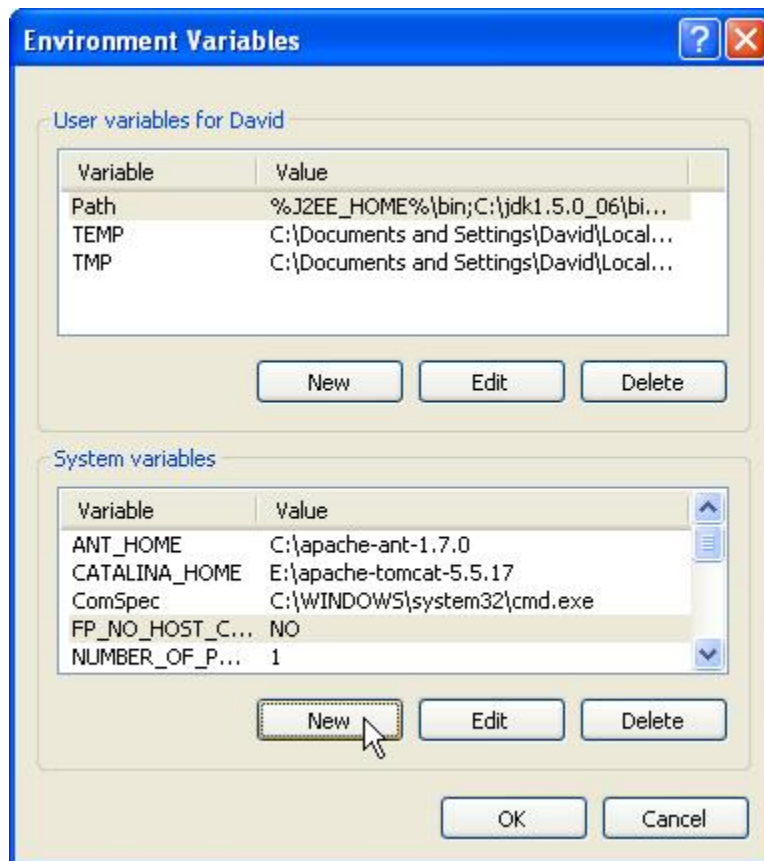
## Set environment variables.

We've made some good progress so far, downloading and installing the J2SE JDK. Now we must configure Java on your computer, which involves setting the **JAVA\_HOME** environment variable, and the **PATH** to the Java executables.

Modify the following instructions for your particular OS.

**JAVA\_HOME:** To set the JAVA\_HOME environment variable, go to Start → Control Panel → System → Advanced tab → and click the Environment Variables button.

In the “Environment Variables” dialog click “New” under either “User variables” to make the environment variable available to the current user only, or click “New” under “System variables” to make the environment variable available to all users of the system.



In the “New Variable” dialog box enter a “Variable name” of JAVA\_HOME and a “Variable value” of the full path to your Java SE root installation folder (in this example it is **C:\jdk1.6.0\_01**, which is the folder just above the “jre” folder.) Click the “**OK**” button in the “New Variable” dialog box.



**PATH:** Next we will add Java to the PATH variable so that your system can find the Java executables. In the “Environment Variables” dialog, select your PATH variable (user or system as desired), click **Edit**, and add the following to your path (don’t forget the semi-colon between PATH entries):

**;%JAVA\_HOME%\bin**

It is usually a good idea to copy your PATH string to a text document before you modify it - if you make a mistake you can paste back in your old path.



Click the “**OK**” button in the “Edit Variable” dialog box.

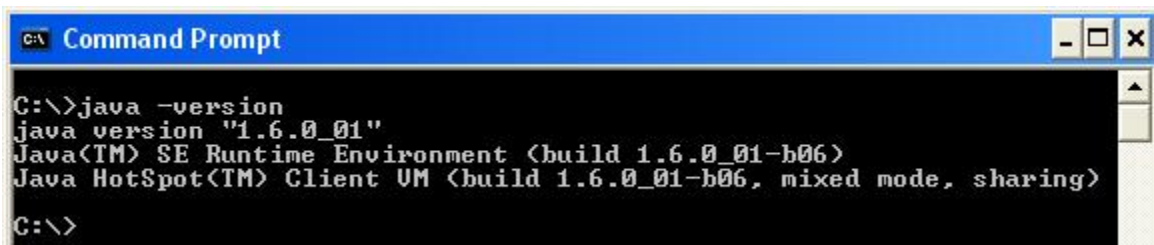
Now that we have set JAVA\_HOME and edited the PATH, in the “Environment Variables” dialog box click “**OK**”.

Click the “**OK**” button to close the “System Properties” dialog box.

To verify that this task is successfully completed, open a command prompt (in Windows this is Start → Run... → type **cmd** in the "Open:" text box → click OK.) At the command line type the following:

```
java -version
```

You should see Java respond with the version that you just installed (assuming that other versions are not ahead in the PATH.)

A screenshot of a Windows Command Prompt window. The title bar is blue and says "Command Prompt". The window has standard Windows window controls (minimize, maximize, close) on the right. The command prompt shows the following text:

```
C:\>java -version
java version "1.6.0_01"
Java(TM) SE Runtime Environment (build 1.6.0_01-b06)
Java HotSpot(TM) Client VM (build 1.6.0_01-b06, mixed mode, sharing)
C:\>
```

## Download Eclipse.

Now that we have the JDK installation and configuration completed, let's move on to the **Eclipse installation**.

For this task you must first download the Eclipse IDE. We spelled out the software download process in some detail for the JDK. For Eclipse, we'll just point you in the right direction and let you download it and save it to your hard drive.






The main Eclipse web site is here:

<http://www.eclipse.org/>

The Eclipse downloads home page is here:

<http://www.eclipse.org/downloads/>

### Eclipse Packages

	<b>Eclipse IDE for Java Developers</b> - Windows (78 MB) The essential tools for any Java developer, including a Java IDE, a CVS client, XML Editor and Mylyn. <a href="#">Find out more...</a>	Windows Linux MacOSX
	<b>Eclipse IDE for Java EE Developers</b> - Windows (125 MB) Tools for Java developers creating JEE and Web applications, including a Java IDE, tools for JEE and JSF, Mylyn and others. <a href="#">Find out more...</a>	Windows Linux MacOSX
	<b>Eclipse IDE for C/C++ Developers</b> - Windows (62 MB) An IDE for C/C++ developers. <a href="#">Find out more...</a>	Windows Linux MacOSX
	<b>Eclipse for RCP/Plug-in Developers</b> - Windows (153 MB) A complete set of tools for developers who want to create Eclipse plug-ins or Rich Client Applications. It includes a complete SDK, developer tools and source code. <a href="#">Find out more...</a>	Windows Linux MacOSX
	<b>Eclipse Classic</b> - Windows (140 MB) The classic Eclipse download: the Eclipse Platform, Java Development Tools, and Plug-in Development Environment, including source and both user and programmer documentation. <a href="#">Find out more...</a>	Windows Linux MacOSX

In late June, 2007, the Eclipse Foundation simultaneously released software from 21 open source software projects. This combined release is named "Europa". From this release you can choose products that can be integrated together to meet your particular development needs.

Please note that this tutorial uses **Eclipse 3.2.2** in a package which is now called "Eclipse Classic", and which in the Europa release is now at version 3.3. You may try a package focused on Java if you wish, but this tutorial does use what is now called "Eclipse Classic" in version **3.2.2** (and in a moment we'll show you where you can still find this version.)

One point about Eclipse Classic is that at the time of this writing, this download appears to be the only one that offers MD5 checksums for checking the integrity of the download (we'll cover integrity checking in a subsequent task.) Eclipse downloads are available on many "mirrors" (download servers) across the world, so because of this broad availability it is important to check the download integrity. Hopefully soon the other downloads will include MD5 checksums, but for now this may influence which download you choose. Sometimes you have to search for the MD5 checksums. In the case of Eclipse Classic, at the time of this writing click its "[Find out more...](#)" link to arrive at the page below.

What You Need

You will need a **Java Runtime Environment** (JRE) to run Eclipse Classic. A Java 5 JRE is recommended.

- **Eclipse Classic** (Windows™ users, see this **note**)
  - **Other downloads for 3.3**
  - **All versions**
  - **Archived Platform Builds**



From the page shown in the image above, note the [All versions](#) link, which we will return to in a moment. For now, click the "[Other downloads for 3.3](#)" link to see all Eclipse Classic downloads:

Eclipse SDK					
Status	Platform	Download	Size	File	Checksum
✓	Windows (Supported Versions)	eclipse-SDK-3.3-win32.zip	141 MB	eclipse-SDK-3.3-win32.zip	(md5) (sha1)
✓	Windows (x86/WPF) **early access** (Supported Versions)	eclipse-SDK-3.3-win32-wpf.zip	139 MB	eclipse-SDK-3.3-win32-wpf.zip	(md5) (sha1)
✓	Linux (x86/GTK 2) (Supported Versions)	eclipse-SDK-3.3-linux-gtk.tar.gz	138 MB	eclipse-SDK-3.3-linux-gtk.tar.gz	(md5) (sha1)

The reason we show you the "[Other downloads...](#)" page is that it not only gives you the Eclipse Classic download links for all supported platforms, it also gives you a link to a corresponding **md5 file** that will allow you to validate the integrity of your download.

**Note that at the time of this writing, you can still get Eclipse 3.2.2 (used in this tutorial), and its md5 checksum, by clicking the [All versions](#) link shown in the first image on this page.**

So choose a download for your OS platform. Download the software **and the corresponding md5 file** into a directory on your hard drive. Note that the Eclipse download is not an executable (exe) but rather a ZIP or TAR.GZ file depending on your platform.

This task will likely take you some time, depending on your internet connection speed, since the download is quite large!

If you look carefully on the Eclipse downloads page, you see one of the following links:

[Click for instructions on how to verify the integrity of your downloads.](#)

Click [here](#) for instructions on how to verify the integrity of your downloads.

We will verify the integrity of the Eclipse download in the next task.

*Please note that from this point onward, this tutorial uses what is now called Eclipse Classic, in version 3.2.2. Therefore some images may show slightly different screens than those in the latest Eclipse version.*



## Validate Eclipse download.

Now that you have downloaded a version of Eclipse that is appropriate for your OS platform, and the corresponding md5 file, we need to validate the integrity of the download, to give us some assurance that parties unknown have not tampered with it.

I have a practice of checking the downloaded file for viruses using my anti-virus software, which is capable of traversing ZIP files. If you are so inclined to do the same, by all means do so.

If you look carefully on the Eclipse downloads page, you see one of the following links:

[Click for instructions on how to verify the integrity of your downloads.](#)

Click [here](#) for instructions on how to verify the integrity of your downloads.

Currently each link is to a page that contains basic instructions for using a utility program to validate a download against its md5 checksum. That same page contains the following link to another web site that offers a Windows version of such a utility program. **This is the link we are interested in for this task:**

<http://www.etree.org/md5com.html>

Now keep in mind that these links are subject to change, and if you are on an OS platform other than Windows, you will have to find a corresponding md5 utility program. The link to the Windows utility looks like this:



For Windows users, download the *md5sum.exe* program *into the same folder that you downloaded Eclipse and the corresponding md5 file.*

Note that you could place *md5sum.exe* into its own folder, such as **md5**, and add the complete location (e.g. c:\md5) to your PATH variable. In that case it is not necessary that the md5 utility and the download that you are checking reside in the same folder – you can run the md5 utility from any location.



It is fairly simple to run the Windows md5 checksum utility: Open a command prompt (in Windows this is Start → Run... → type **cmd** in the "Open:" text box → click OK.)

Change the working directory to the folder of your Eclipse, md5, and md5sum.exe downloads (note the following example):

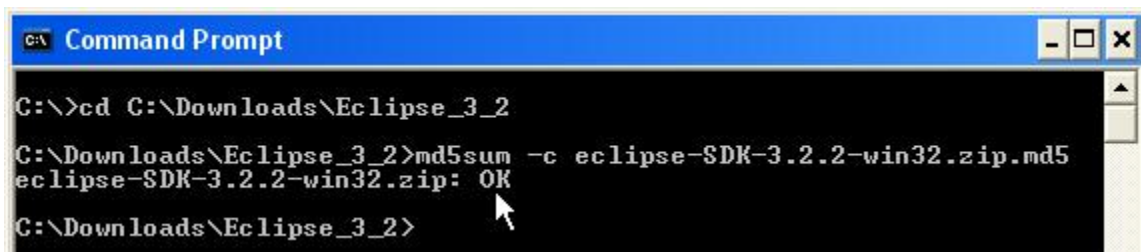
```
cd C:\Downloads\Eclipse_3_2
```

Now execute the following command, which includes your downloaded md5 filename:

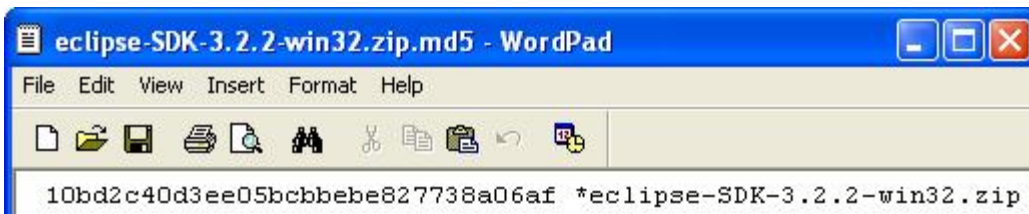
```
C:\Downloads\Eclipse_3_2>md5sum -c eclipse-SDK-3.2.2-win32.zip.md5
```

If the Eclipse download has not been tampered with, you should get a message similar to this:

```
eclipse-SDK-3.2.2-win32.zip: OK
```



The setup for this verification (already performed for you) is that the md5 file is created in which one line contains the md5 checksum, a space, an asterisk, and the name of the download to test.



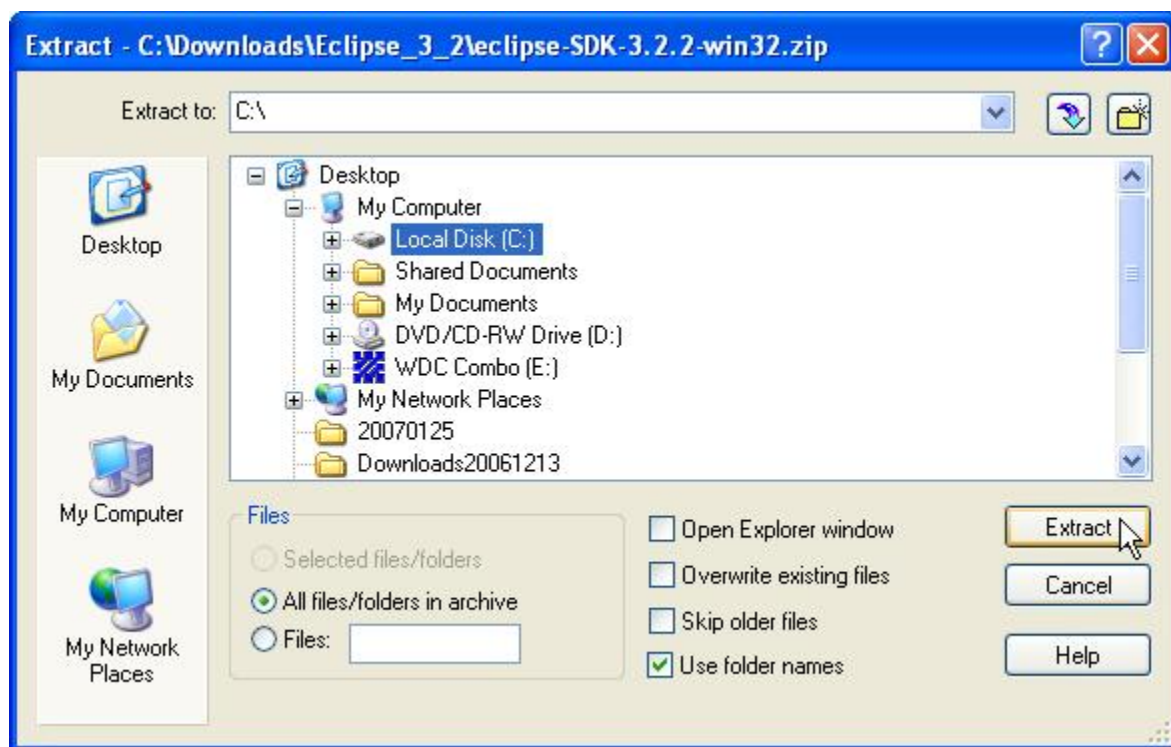
## **Install Eclipse.**

Now it is time to install and configure Eclipse. Later we will write and execute a Java program in the Eclipse IDE.

Since for this tutorial we downloaded Eclipse in a .ZIP archive, we will open the archive in WinZip and extract its contents onto the hard drive (if you do not yet have WinZip, you can purchase this utility at <http://www.winzip.com/>).

Double-click the Eclipse .zip archive (or right-click and choose “Open with WinZip”).

In WinZip execute “Actions...Extract” and in the “Extract” dialog box select “All files/folders in archive”, select the root of your hard drive, check the “Use folder names” box, and click the “Extract” button. Notice that after the archive is extracted, the Eclipse application will be located in a folder called “C:\eclipse” on your hard drive.

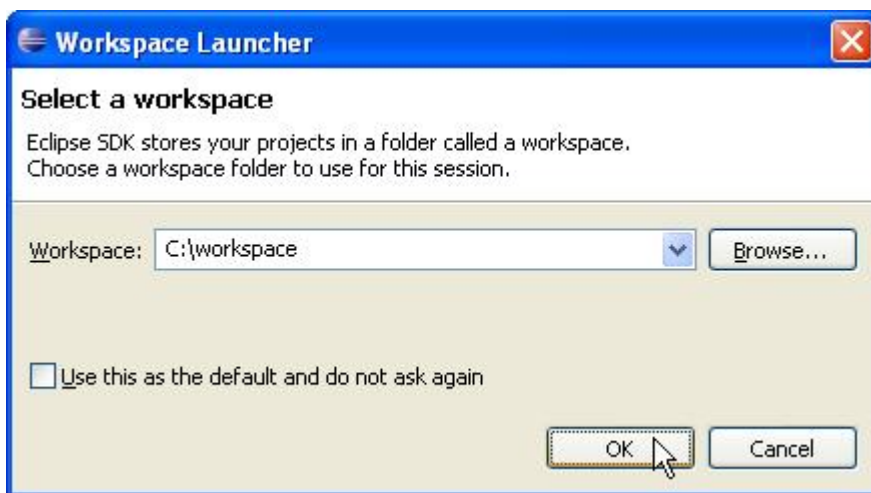


The Eclipse executable (**eclipse.exe**) is in the root eclipse folder – you can right-click and drag this exe to your Desktop and create a shortcut to Eclipse on your Desktop.

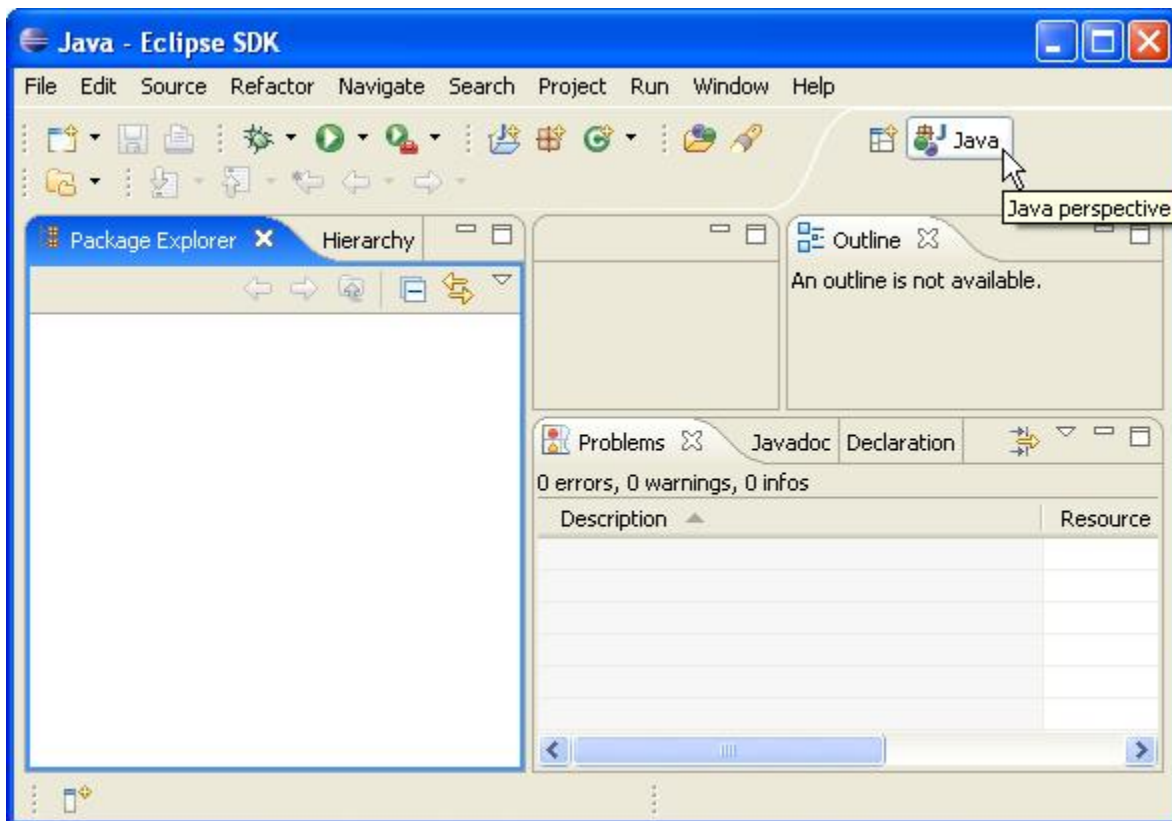


Double-click the Eclipse executable (or your shortcut) to open Eclipse. Eclipse displays the “Workspace Launcher” dialog and asks you to select a workspace directory. An Eclipse workspace stores your Eclipse configuration. Change to a directory in which you would like the workspace to reside, or accept Eclipse’s workspace recommendation and in the “Workspace Launcher” dialog click the “OK” button. *Warning: **Never** check the “Use this as the default and do not ask again” box.*

Eclipse creates the new workspace and displays its Welcome screen.



Close the Welcome screen (click the “X” on the Welcome tab) and the Eclipse IDE is ready to go in the **Java Perspective**. In Eclipse, a **Perspective** is a set of related **Views** (windows) that enable a development specialist to perform specific tasks. You can see the various Perspectives offered out-of-the box with the menu option *Window, Open Perspective*. In this tutorial we will stay in the Java Perspective, which offers Views and editors for creating and executing Java applications.

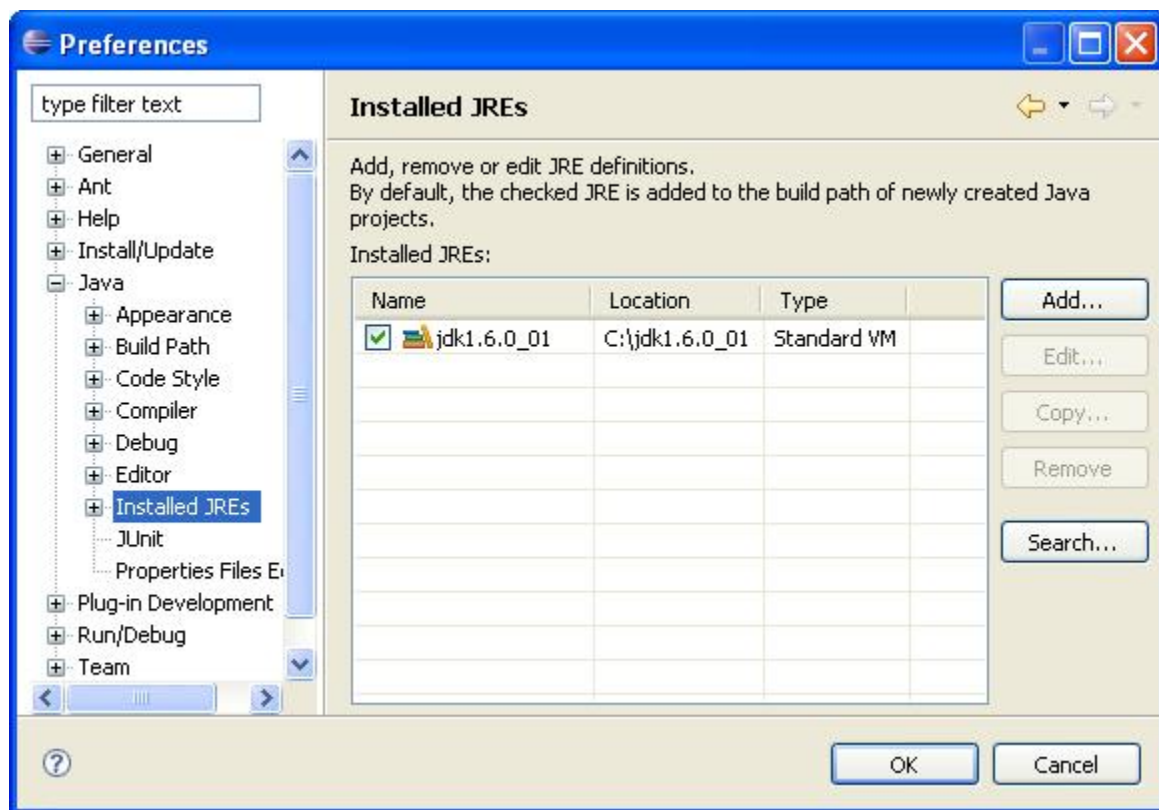


## Set Eclipse preferences.

Now that we have Eclipse up and running, we will survey some Eclipse configuration options in the important “Preferences” dialog box.

From the Eclipse menu bar select **Window, Preferences...** to open the “Preferences” dialog box. The Preferences dialog is one that you will become very familiar with as you continue to work with Eclipse. In this dialog you can make various workspace configurations and set your own development preferences in the categories listed at the left of the dialog.

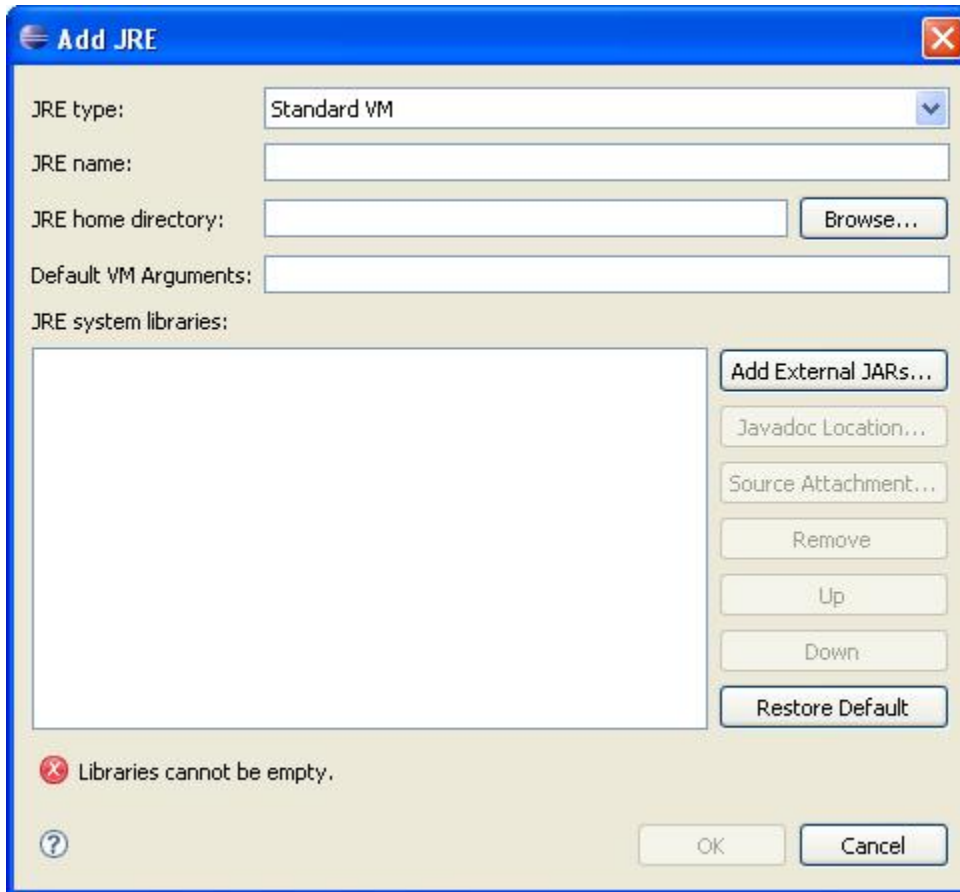
Note that in the image below we have expanded the “Java” category and have selected the “Installed JREs” option.



Notice in the previous image that the JDK that we previously downloaded and installed is already providing the default JRE (Java Run Time) for Eclipse – Eclipse is smart enough to figure that out on its own during its initial start-up.

If you **did not** download the JDK earlier in this tutorial, and your existing JRE **does not** appear in the “Installed JREs” preference, you must add it yourself. Click the “Add” button and fill in the following information:

- A name (which you make up) to identify this JRE.
- The *JRE home directory* (this is the directory immediately above the JDK “jre” directory). Eclipse will display the library JAR files for the JRE, and will even attach the source code if *src.zip* is in the home directory.



Click “OK” in the “Add JRE” dialog box, and back in the “Installed JREs” preference check the box beside your desired JRE and click “OK”.

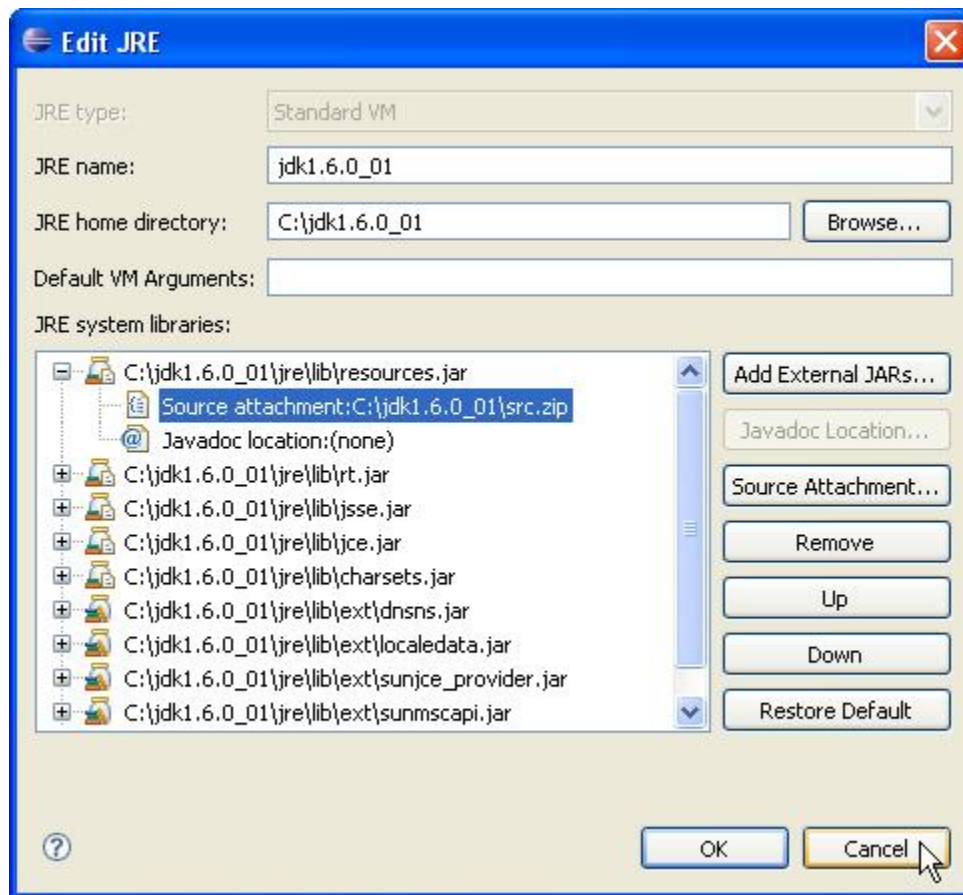
For Eclipse help on adding a JRE, take the menu option *Help, Help Contents* and look at the results under the following search string: *Adding a new JRE definition*

Once again, note that if you followed the steps in this tutorial, Eclipse has found the JRE that we previously downloaded and installed, and Eclipse has made that the default JRE in the “Installed JREs” preference.

In the “Installed JREs” preference, select our default JRE and click the “**Edit**” button.

Notice the information that Eclipse has provided. Eclipse even found the Java source code that we installed with the JDK, as you can see from the highlighted configuration below.

Click “**Cancel**” to exit the “Edit JRE” dialog box and return to the “Installed JREs” preference.



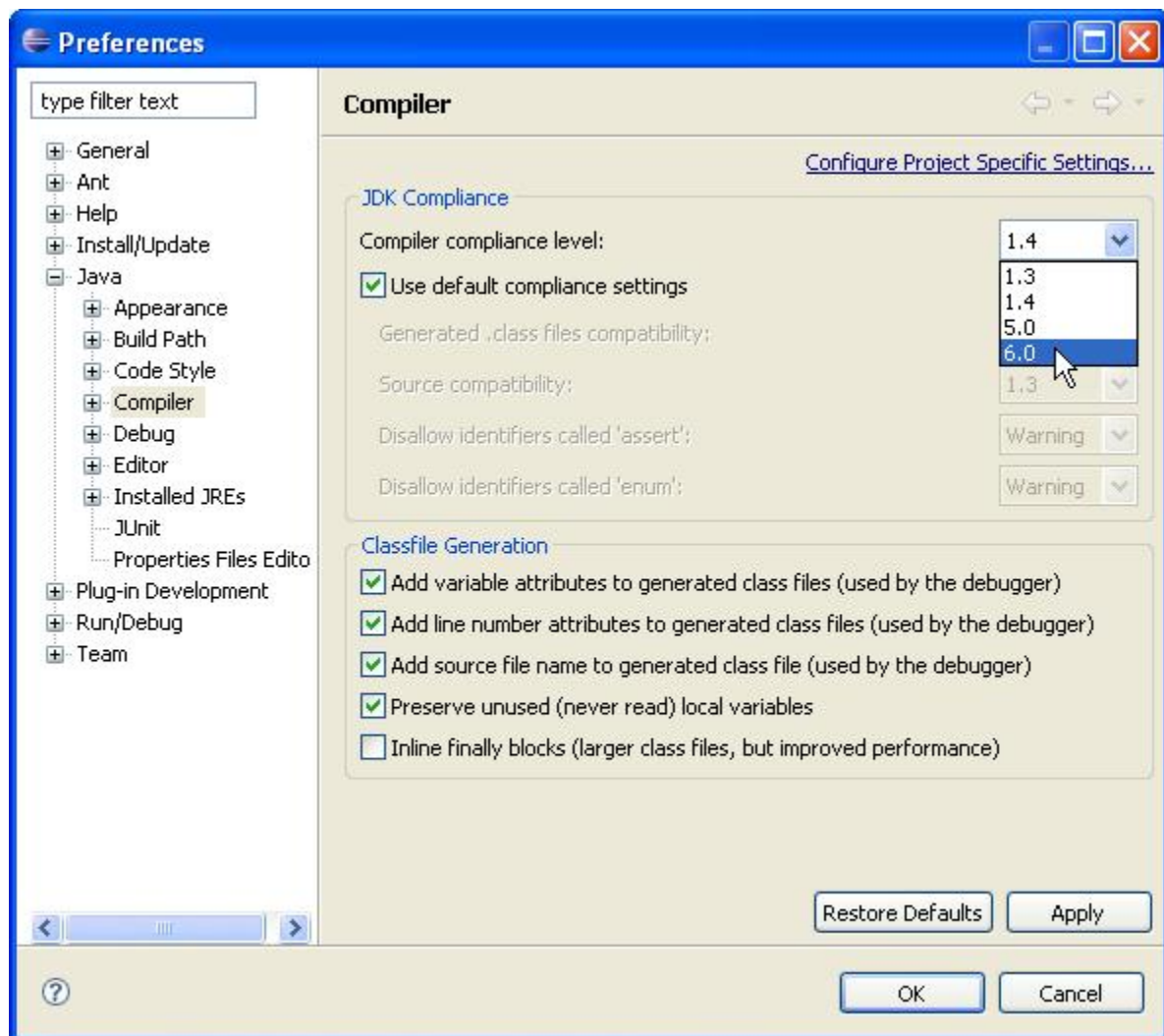


Back in the “Preferences” dialog box, under the “Java” category select the “Compiler” option. Set the “*Compiler compliance level*” to correspond to the JDK version you are using. In this tutorial we will set the “*Compiler compliance level*” to 6.0 since we are using the JDK 6. You can read more about this release here:

<http://java.sun.com/javase/6/>

We do think that this article on Java SE 5 features is still worth a look:

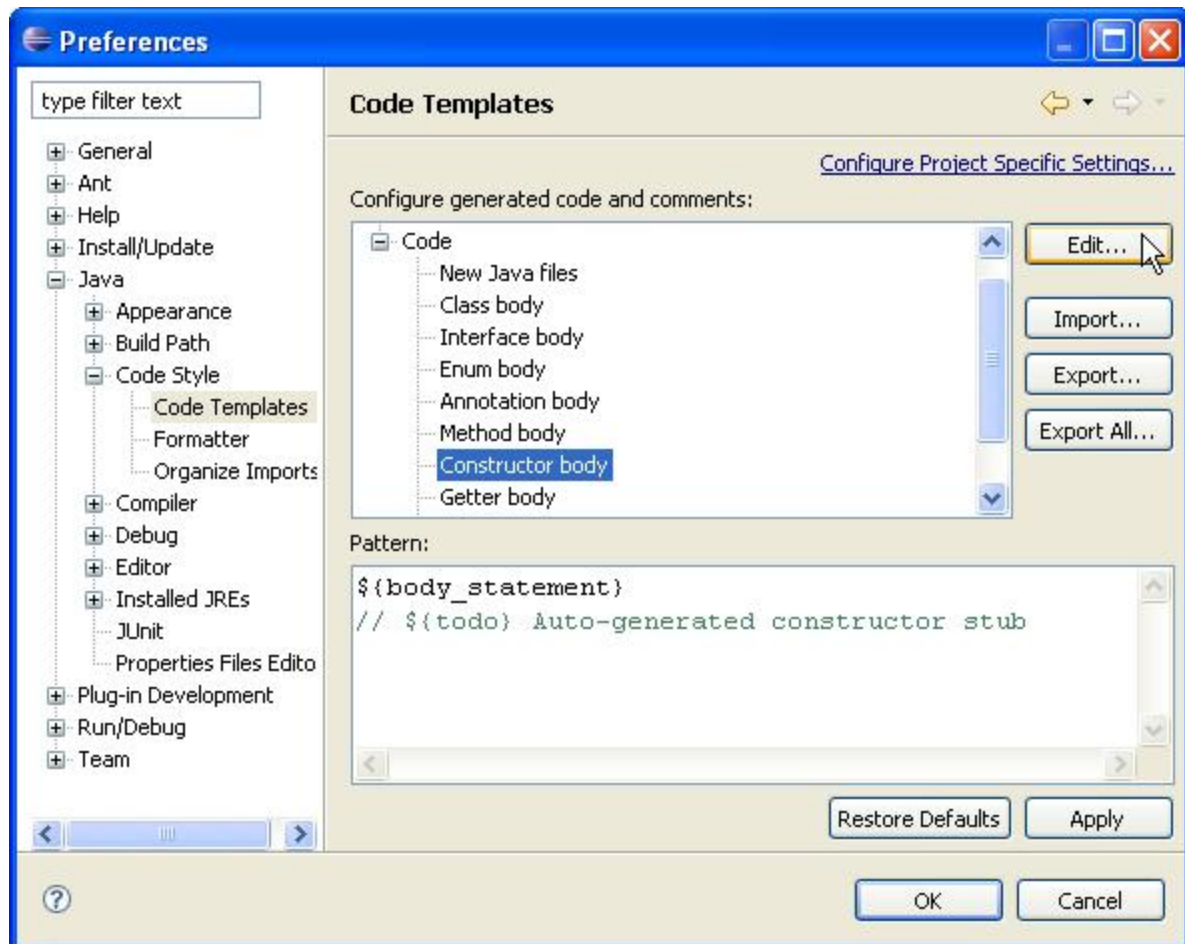
<http://java.sun.com/developer/technicalArticles/releases/j2se15/>





Now, in the “Preferences” dialog, expand the *Java...Code Style* tree and select “Code Templates”. When Eclipse generates code, it uses the templates in this preference.

In the example below, when Eclipse generates a class constructor it will include a “// TODO” **task tag** in the code that will also show up in the Eclipse *Task* view (Window → Show View → Other → General → Tasks) to remind you that you may want to revisit the constructor later.



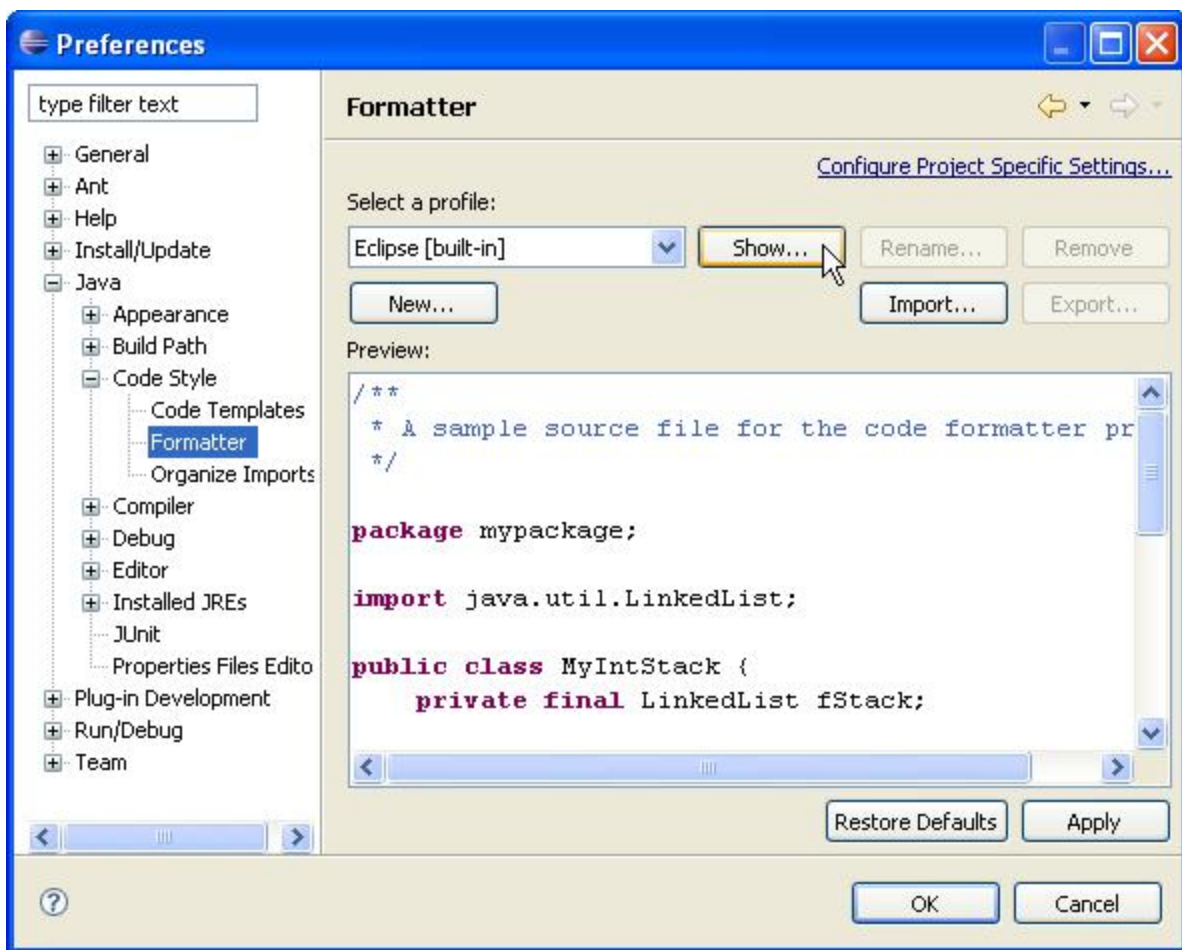
If you don't wish to generate such auto-reminders, you can select the "Constructor body" template and click "Edit..." to open the "Edit Template" dialog box. In the "Edit Template" dialog box you can delete (or change) the undesired text and click "OK". Click "OK" back in the "Code Templates" preference.

You can modify other templates in a similar way. As you use Eclipse to generate code and comments, you will see the usefulness of these code template preferences.



There are workspace preferences for formatting source code. Formatting configurations are grouped into profiles. In the “Eclipse [built-in]” profile, click the “Show...” button and notice the vast array of options. Under the *Comments* tab, I uncheck the *New line after @param tags* box (which affects Javadoc formatting.) Under the *Line Wrapping* tab I prefer a *Maximum line width* value longer than the default of 80 characters. If you make formatting changes and click “Apply” or “OK”, you will be prompted to enter a new profile name. For future formatting configuration changes you will “Edit” this profile.

When your team has uniformly configured formatting preference, developers can right-click anywhere inside an Eclipse source code editor and take the context option **Source, Format** (or with the editor active type **Ctrl+Shift+F**).

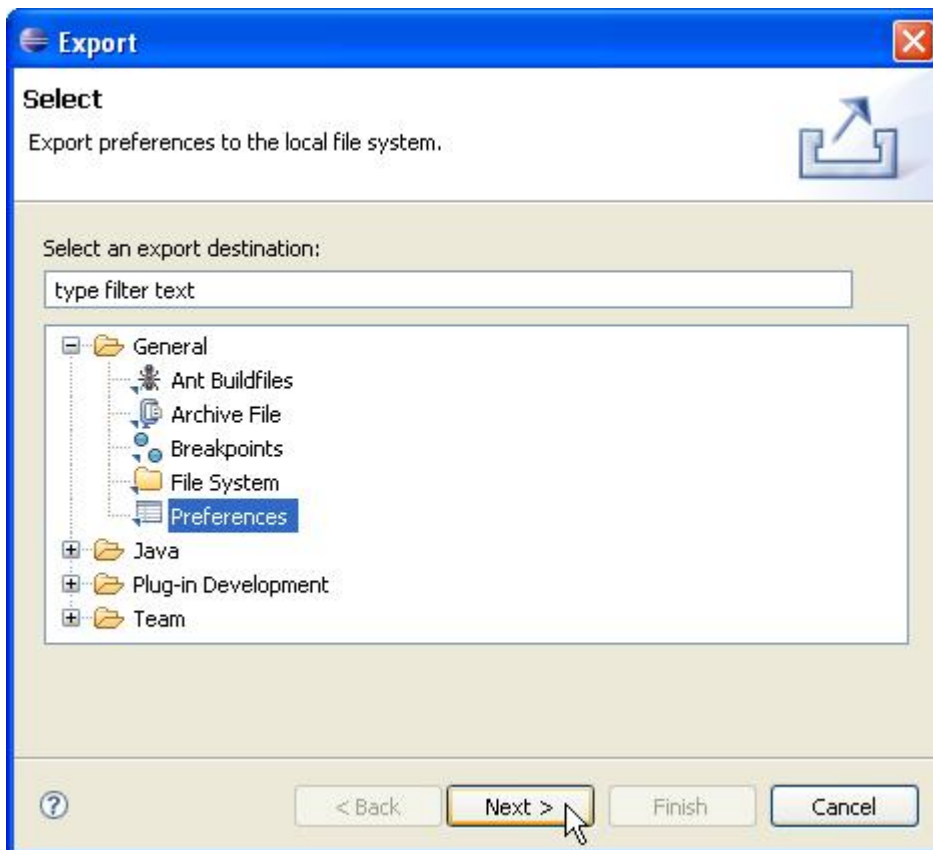


At this time we are finished setting and exploring Eclipse preferences, so please click “OK” in the “Preferences” dialog box. If you get a “Compiler Settings Changed” dialog box, click “Yes”.



Clearly the Eclipse “Preferences” dialog has a variety of options available to you, and if you need to change the behavior of Eclipse take a close look in this dialog and you may very well find the configuration that you want.

Once your preferences have been established, you can export them in case you want to import them into a workspace later. From the Eclipse menu choose *File, Export..., General, Preferences*.

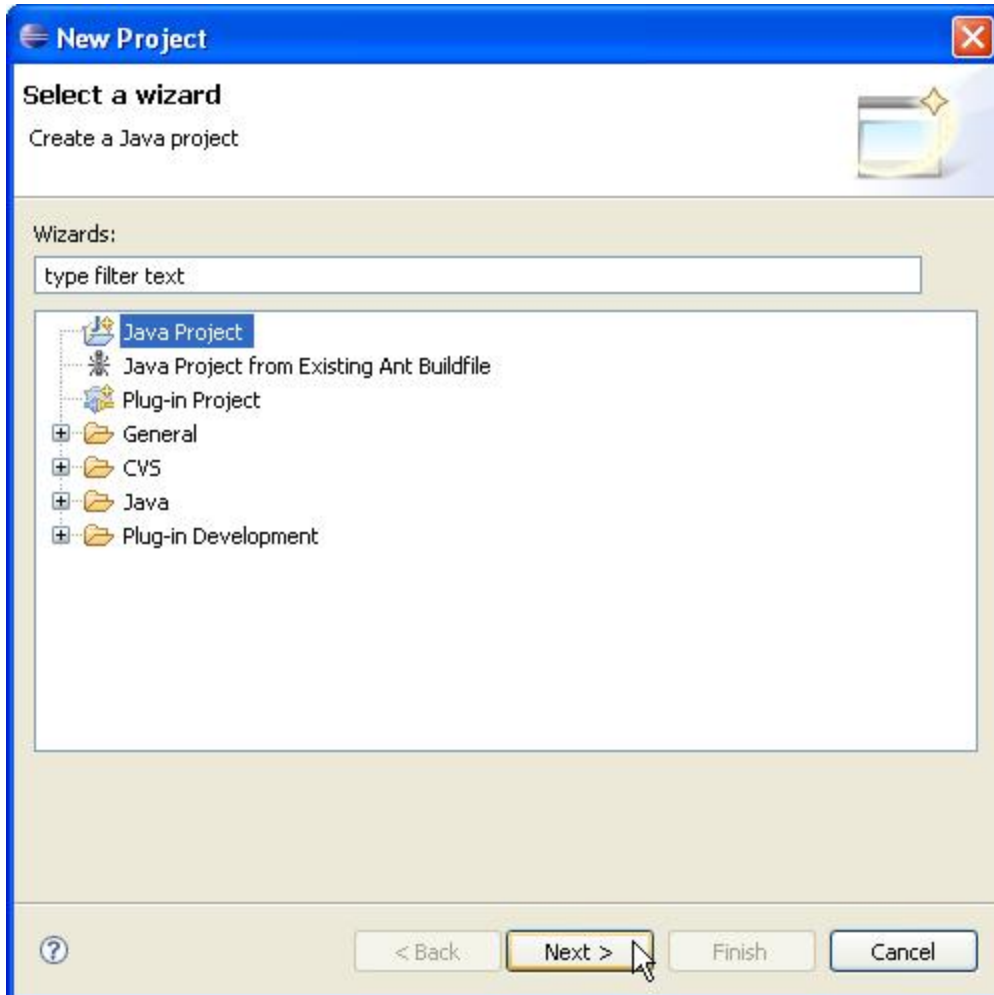


## Create an Eclipse project.

Now we will use Eclipse to write a small Java application. Our first step is to create a new Eclipse **Project**.

From the Eclipse menu bar select *File, New, Project* to start the “New Project” wizard.

Select “Java Project” and click “Next”.

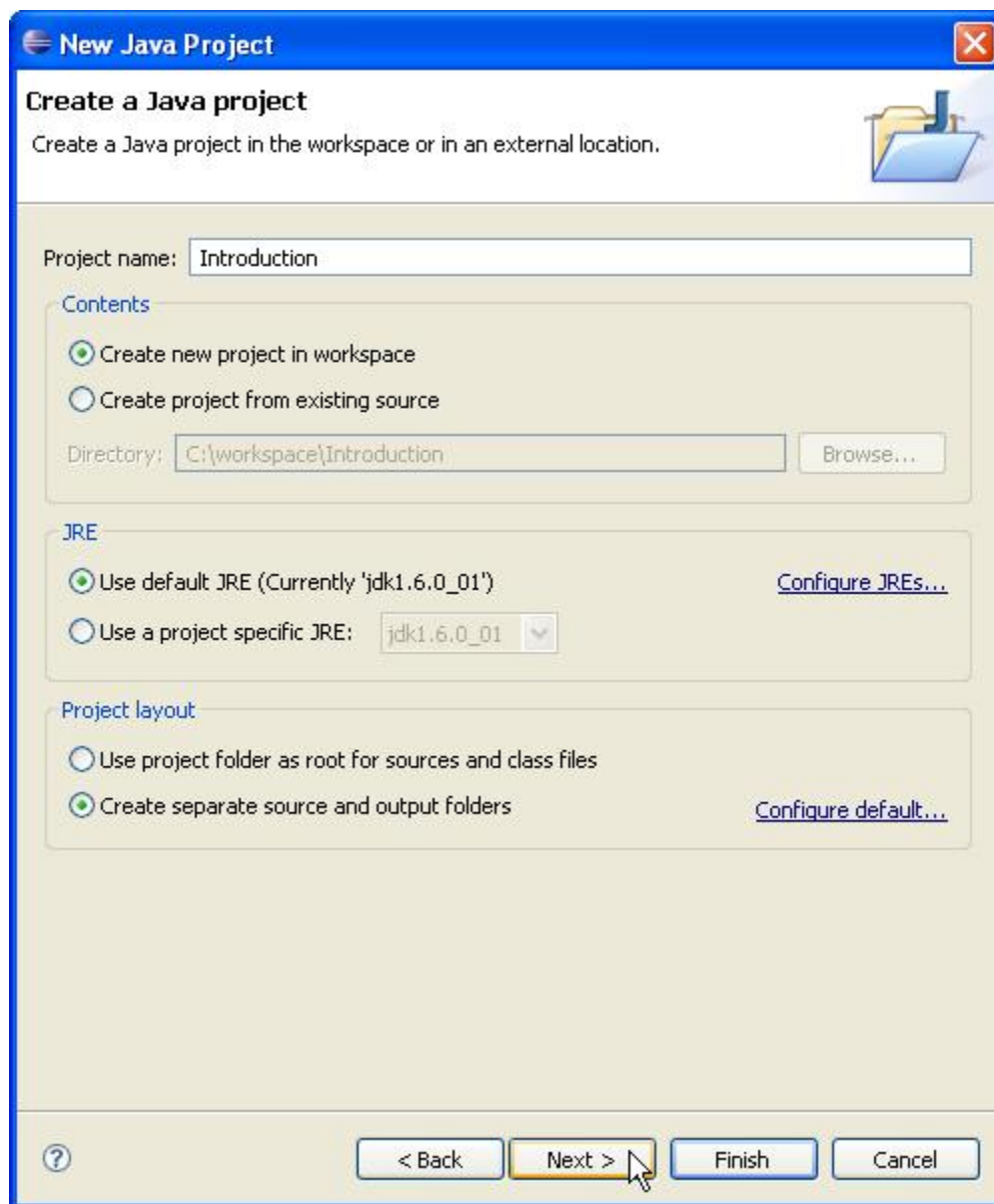


In the “New Java Project” dialog box give the project a name of “Introduction”.

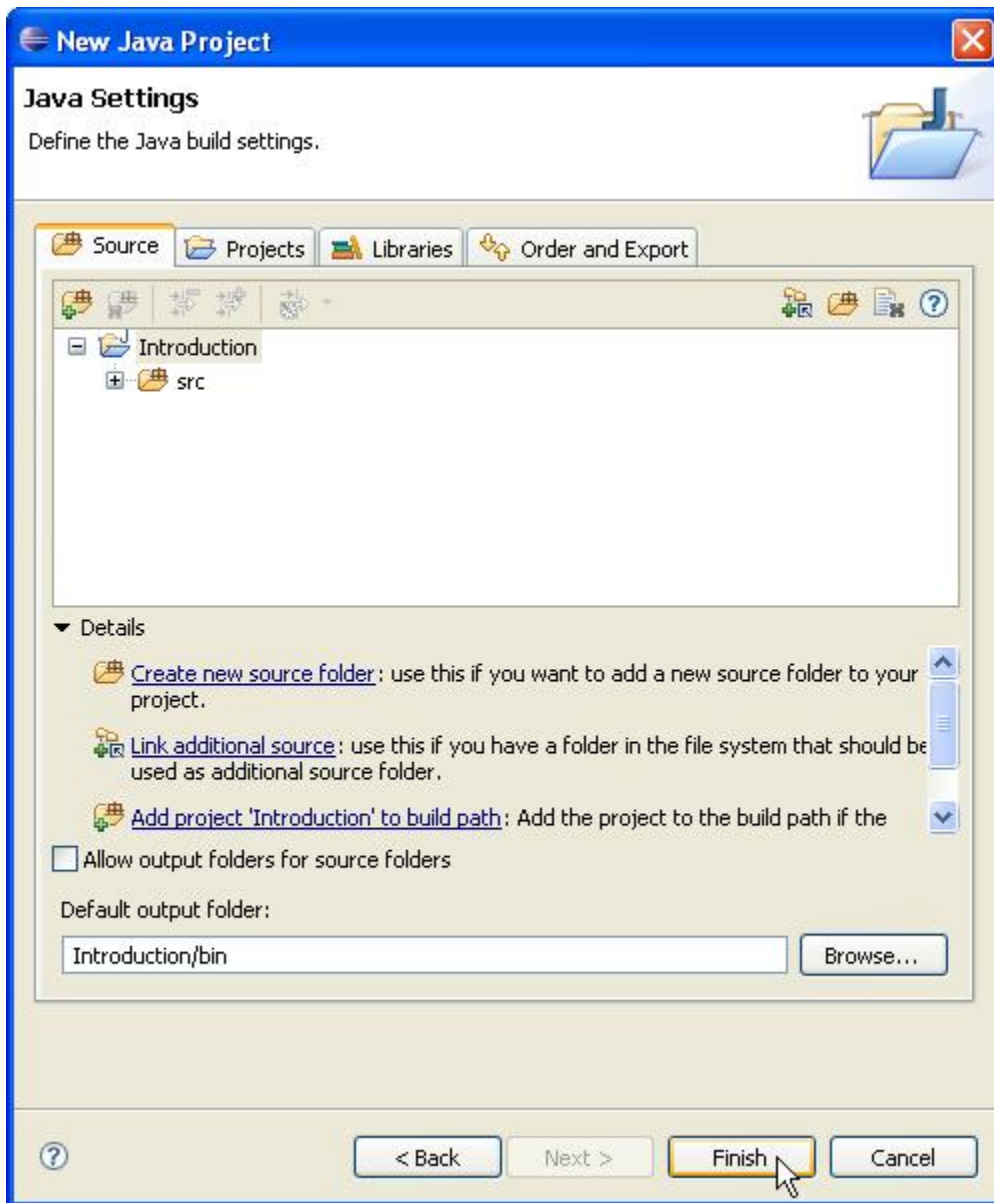
Leave the “Create new project in workspace” radio button selected to create the project in the current workspace (after this step you will see a project folder called “Introduction” under your “workspace” directory.)

Keep the “Use default JRE” selection.

Click the “Create separate source and output folders” radio button and then click “**Next**”.



Notice in the next screen of the “New Java Project” wizard that Eclipse by default will store your source (.java) files in the project’s “src” directory, and your compiled Java class (.class) files in the project’s “bin” directory. Click the “**Finish**” button to accept these defaults and create the new project. You now have a new project showing in your Java perspective!





## Create Java packages.

Now let's build a simple Java application in our new "Introduction" Eclipse Java project. The primary purpose of this project will be to introduce Eclipse features, but in the process we will cover some basic Java as well.

This project will consist of three Java classes: **Person**, **Address**, and **Application**. Each Person has an Address, and the Application class will print out Person and Address properties to the Eclipse console.

Java application code is organized into structures called **packages**, which then correspond to folders in the file system. Let's create two packages to organize the code of our application.

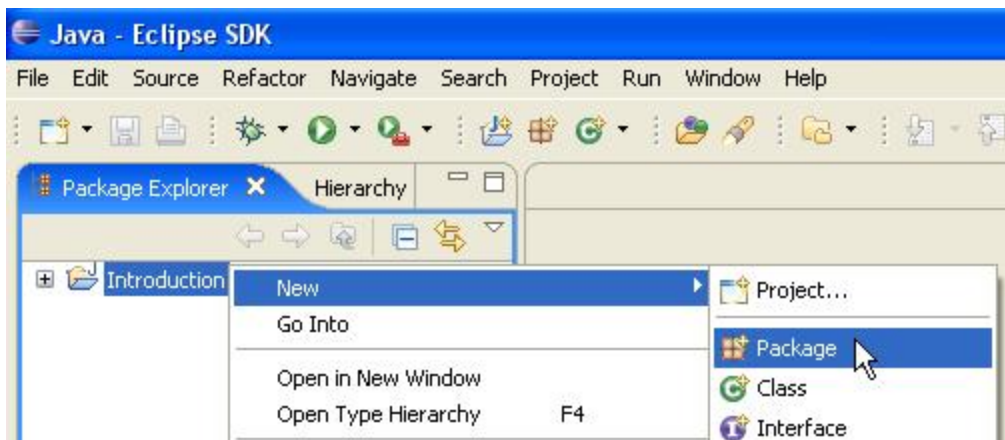
**com.as.tutorial.model** - for the Person and Address classes

**com.as.tutorial.main** - for the Application class.

Note that we are choosing a package naming convention of the form *com.company.application.module*, although this form is followed loosely because our context is that of a tutorial, not an ultimately production application.

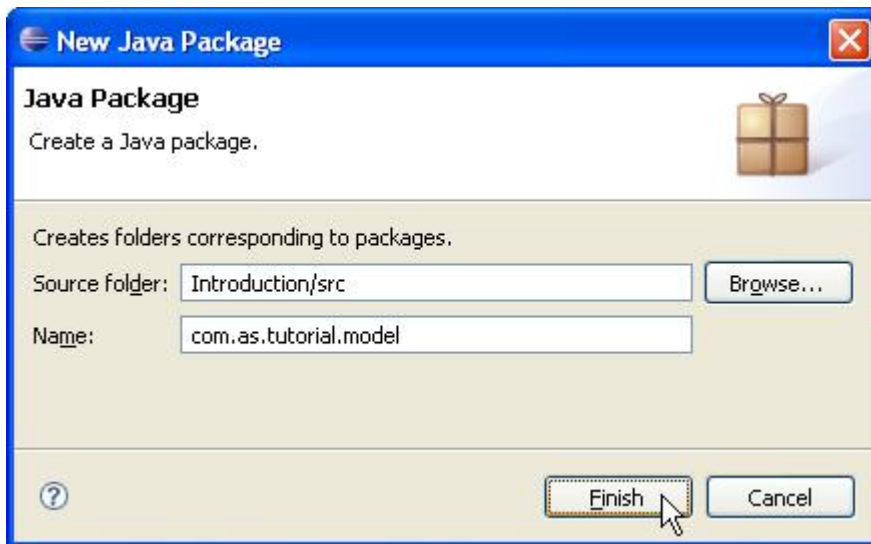
In Eclipse, make sure your new project is visible in the Package Explorer. The "Package Explorer" is one of the views available in the Eclipse Java Perspective, and if necessary within the Java Perspective you can open the Package Explorer by taking the menu option Window → Show View → Package Explorer.

From the project context menu (i.e. right-click) select New, Package.





First create the package **“com.as.tutorial.model”** in the Introduction/src directory. Note that in Java, the individual package elements are separated by a period (.).



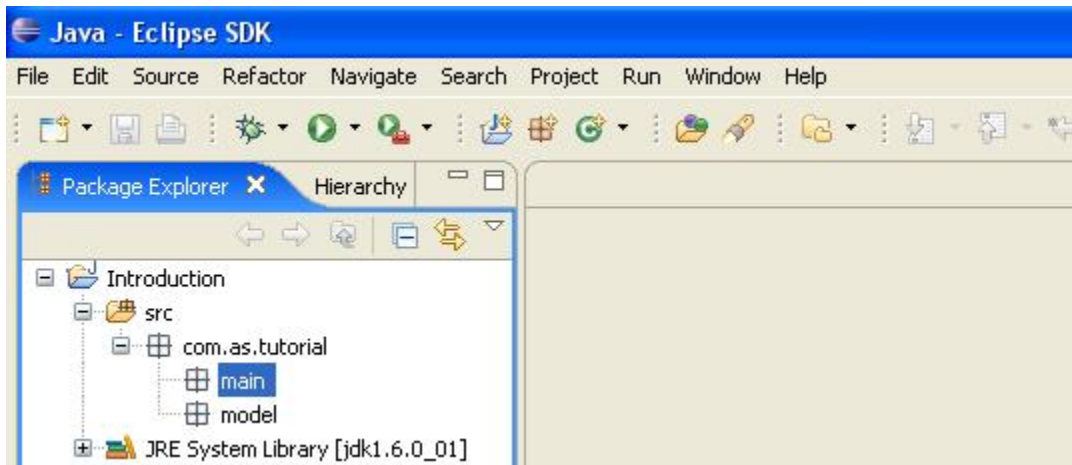
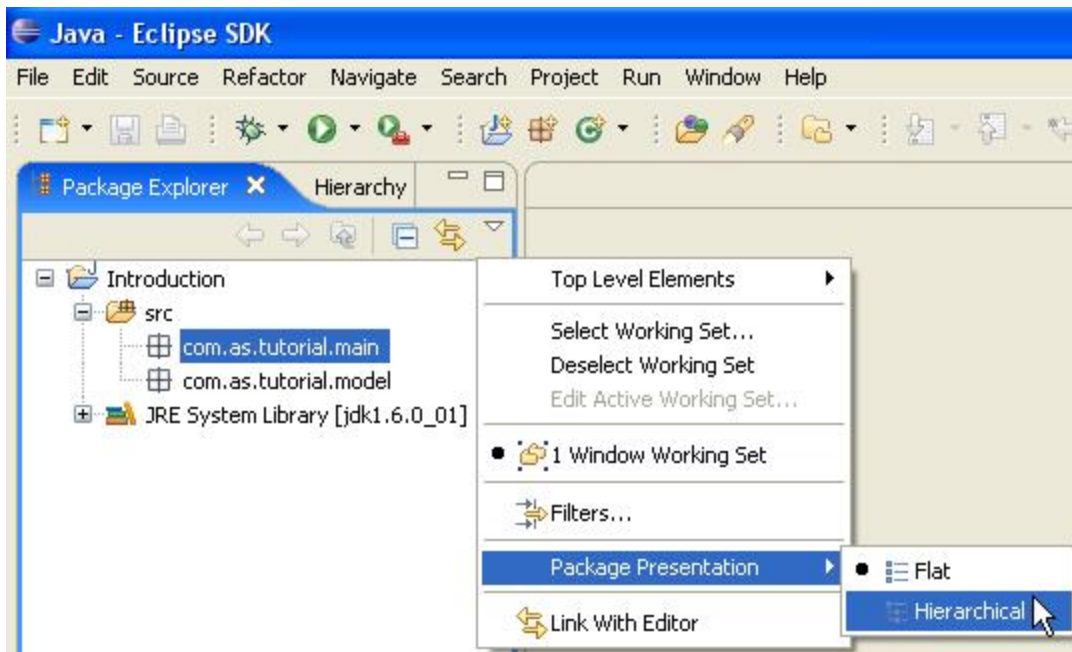
In a similar manner create the package **“com.as.tutorial.main”** in the Introduction/src directory.

When you are finished, your project should look like this:



We will soon create our Java classes in these packages.

If you prefer, you can display your packages in the Package Explorer in a more intuitive “hierarchical” structure. Just click the down-arrow at the right of the Package Explorer View and take the option *Package Presentation, Hierarchical*.



## Create Java classes.

Now we will create the classes of our application. Classes are used to instantiate objects that encapsulate data (private instance variables) and behavior (methods.)

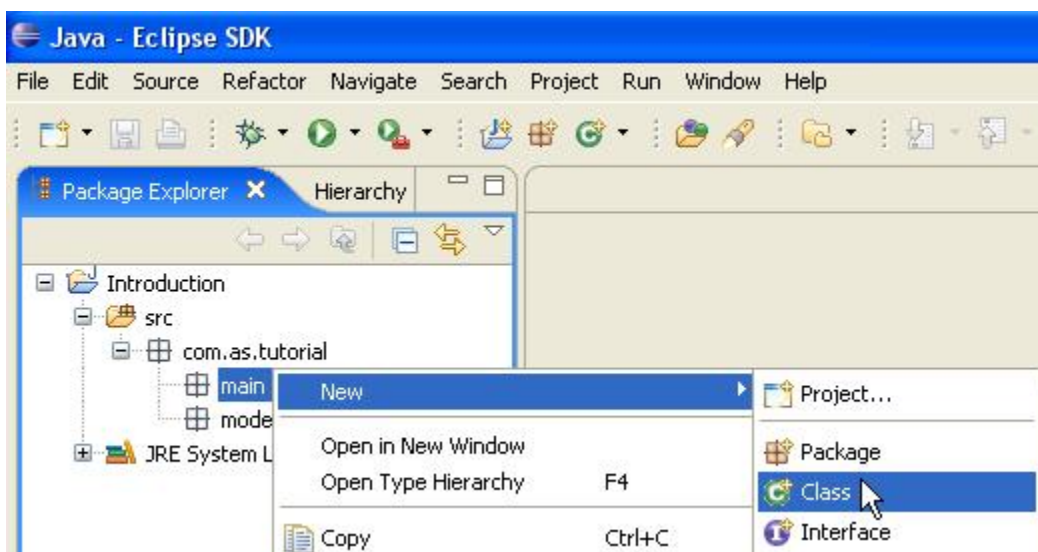
We will have three classes in our application. They are as follows:

**Person:** This is a Java Bean that will define the properties of a *person*.

**Address:** This is a Java Bean that will define the properties of an *address*.

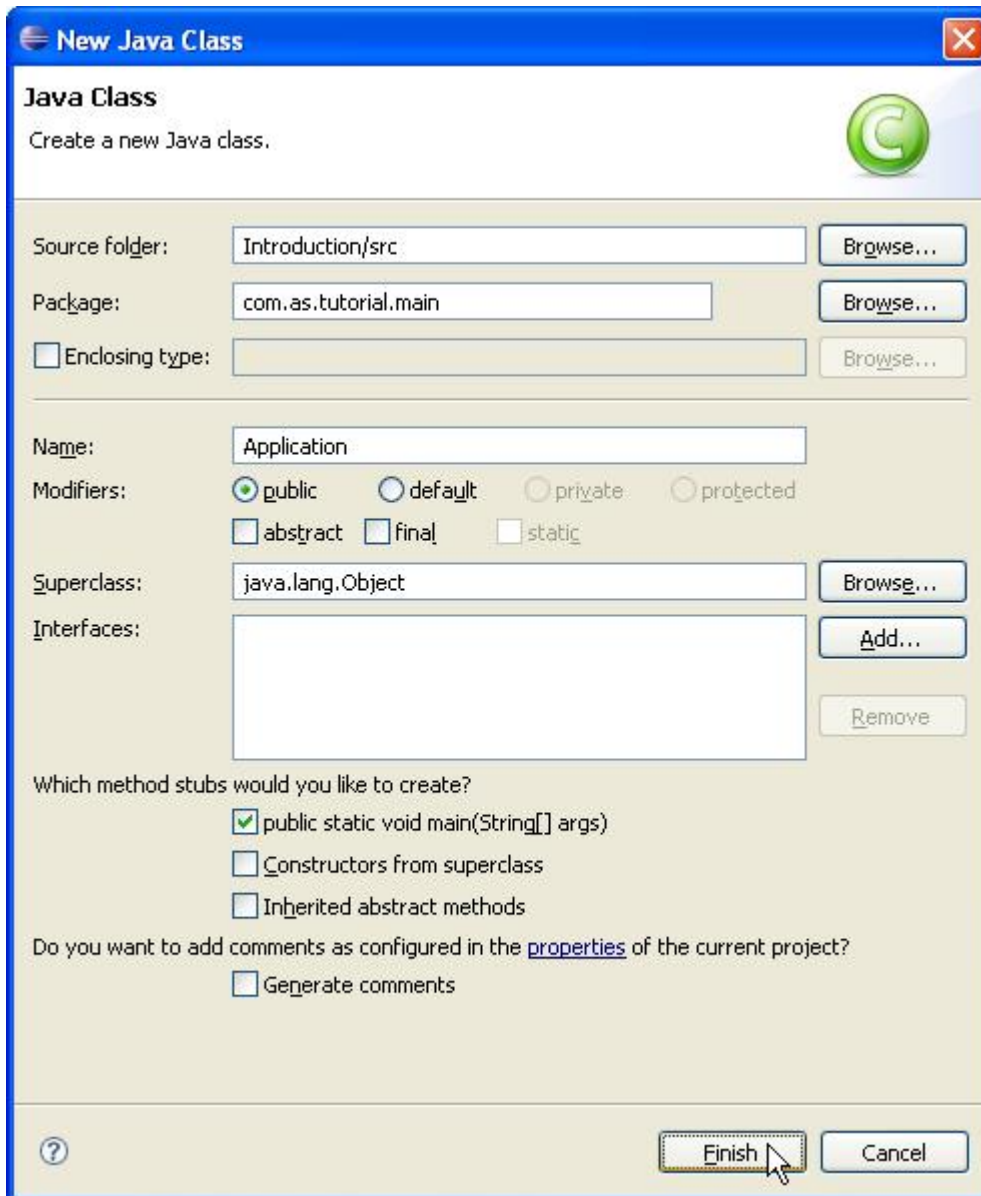
**Application:** This is our executable application class, which will simply print Person and Address properties to the Eclipse console.

Let's start by creating the Application class, the executable class in our application. Java classes are named with a leading capital letter, and each inner word of the name begins with a capital letter. Right-click the **com.as.tutorial.main** package and from the context menu select New, Class.



In the “New Java Class” dialog accept the defaults except for the following. Enter a class “**Name**” of **Application**, and of the three “**method stub**” check boxes only check the “**public static void main(String[] args)**” check box.

Make sure that your “New Java Class” dialog looks like that in the image below, and click “Finish” to create the Application class.



In a similar manner, create the other two classes according to the following specifications. *Please follow these specifications carefully.*

## Person

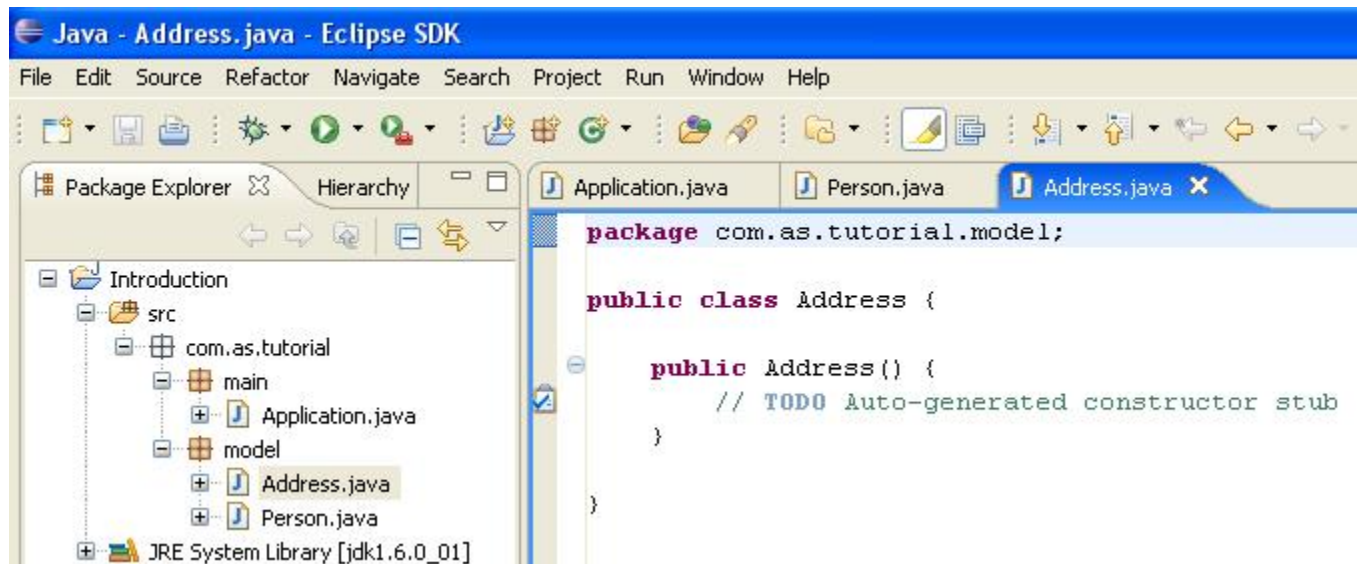
- Package: com.as.tutorial.model
- Name: Person
- Modifiers: Public
- Superclass: java.lang.Object
- Method stub: **only check** “Constructors from superclass”

## Address

- Package: com.as.tutorial.model
- Name: Address
- Modifiers: Public
- Superclass: java.lang.Object
- Method stub: **only check** “Constructors from superclass”

In some applications the model objects, such as persistent classes, implement the `java.io.Serializable` interface, but we will not implement that interface in this tutorial.

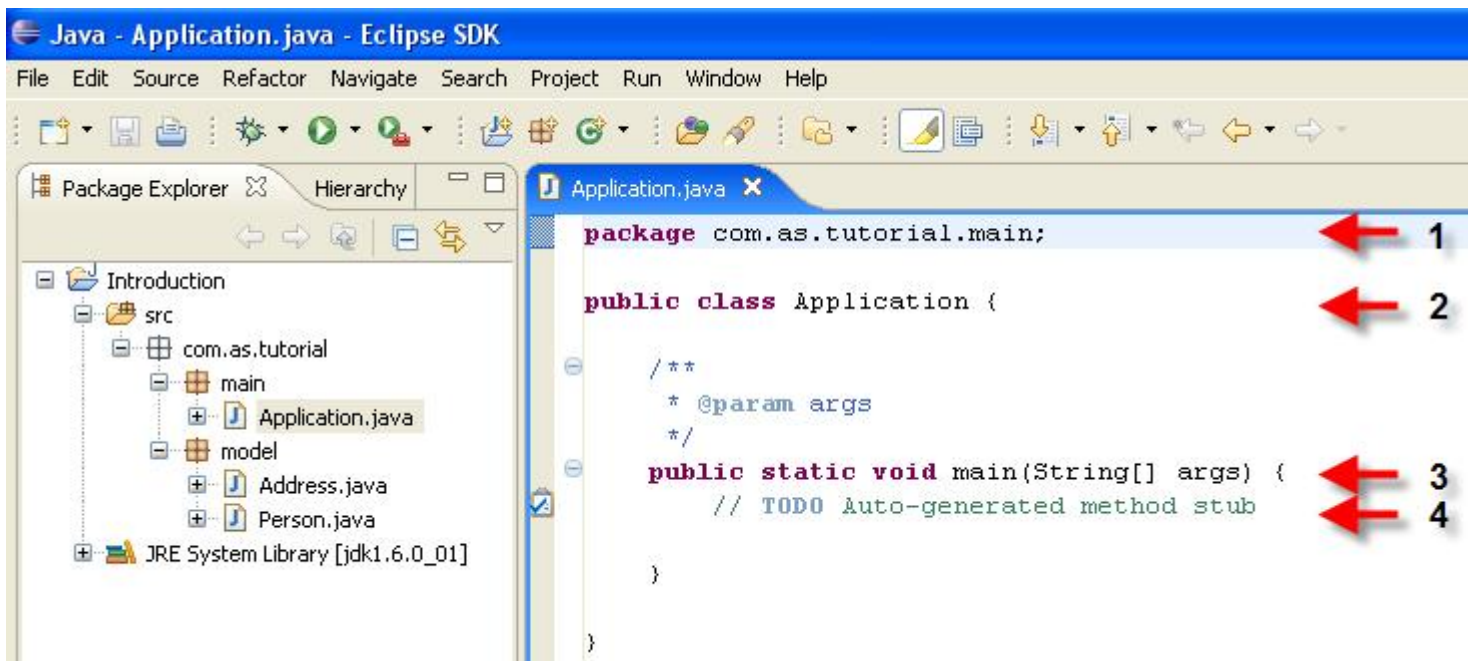
When you are finished, your Package Explorer should look like that in the image below.



## Explore a class.

Now that our classes are created, we will examine the Application class a little closer.

In the Package Explorer, double-click the Application class to open it in the Java editor.





Refer to the image above as you review the following notes about the class:

**Note 1:** The first line of the file is the “**package**” statement which specifies the name of the package in which the class is contained.

**Note 2:** A class declaration will often include the “**extends**” keyword following the class name. The “extends” keyword is followed by the name of a class from which the given class will **inherit**, that is, from which the given class will obtain its default behavior. In our class declarations there is no “extends” keyword. Remember the “New Java Class” dialog that we used to create our classes? In each case we left the “Superclass” as the default of **java.lang.Object**. If class A does not specifically “extend” another class B, by implication class A extends **java.lang.Object**, the highest-level class in the Java inheritance hierarchy. Therefore all three of our classes extend “**Object**”.

We could have written the class declaration like this:

```
public class Application extends Object
```

However, “extends Object” is implied when our class declaration is just as we currently have it:

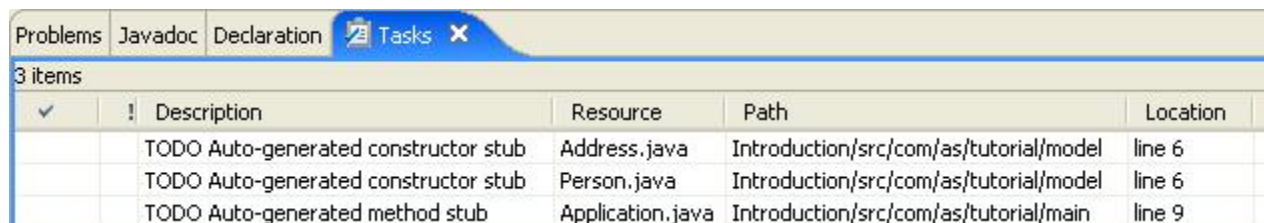
```
public class Application
```

**Note 3:** The signature “**public static void main(String[] args)**” declares a special kind of **method** called a “**main**” method, which is the point of entry for a Java application. Later we will put code in this method.

**Note 4:** We also have the following Eclipse-generated line in our “main” method:

```
// TODO Auto-generated method stub
```

The reason we have this line is that we did not change the Preferences for generated “Method body” code (Window → Preferences → Java → Code Style → Code Templates → Code → Method body) to omit this reminder. You can see the message “Auto-generated method stub” in the “Task” View by opening this View if it is not already open (Window → Show View → Other → General → Tasks).



✓	!	Description	Resource	Path	Location
		TODO Auto-generated constructor stub	Address.java	Introduction/src/com/as/tutorial/model	line 6
		TODO Auto-generated constructor stub	Person.java	Introduction/src/com/as/tutorial/model	line 6
		TODO Auto-generated method stub	Application.java	Introduction/src/com/as/tutorial/main	line 9

## Declare Person instance variables.

Next we will add some data members to our Person and Address classes. In Java these are called *instance variables*, *member variables*, or *fields*.

For Person, we will add the following instance variables:

- firstName
- lastName
- birthYear

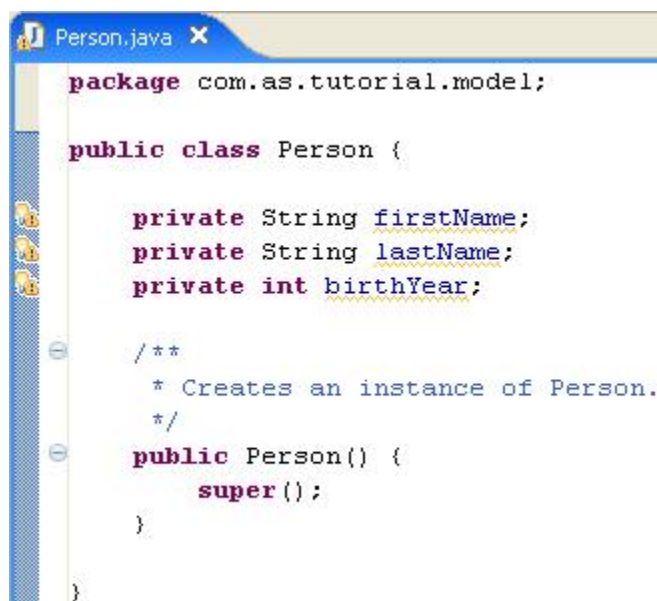
For Address, in a subsequent task we will add the following instance variables:

- addressLine1
- city
- state
- zip

Each variable will be of Java type **java.lang.String**, except for birthYear which will be of the Java primitive type **int**.

In the Package Explorer, double-click the Person class to open it in the Java editor. In the editor type the three variables firstName, lastName, and birthYear as **private instance variables** of the Person class. Each instance of the class will have its own copy of each variable.

Delete the constructor TODO and add a *super()*; superclass constructor call. Add Javadoc if you wish. Press **Ctrl+S** to save your changes. Ignore the yellow warnings – we will fix those later.



```
Person.java x
package com.as.tutorial.model;

public class Person {

    private String firstName;
    private String lastName;
    private int birthYear;

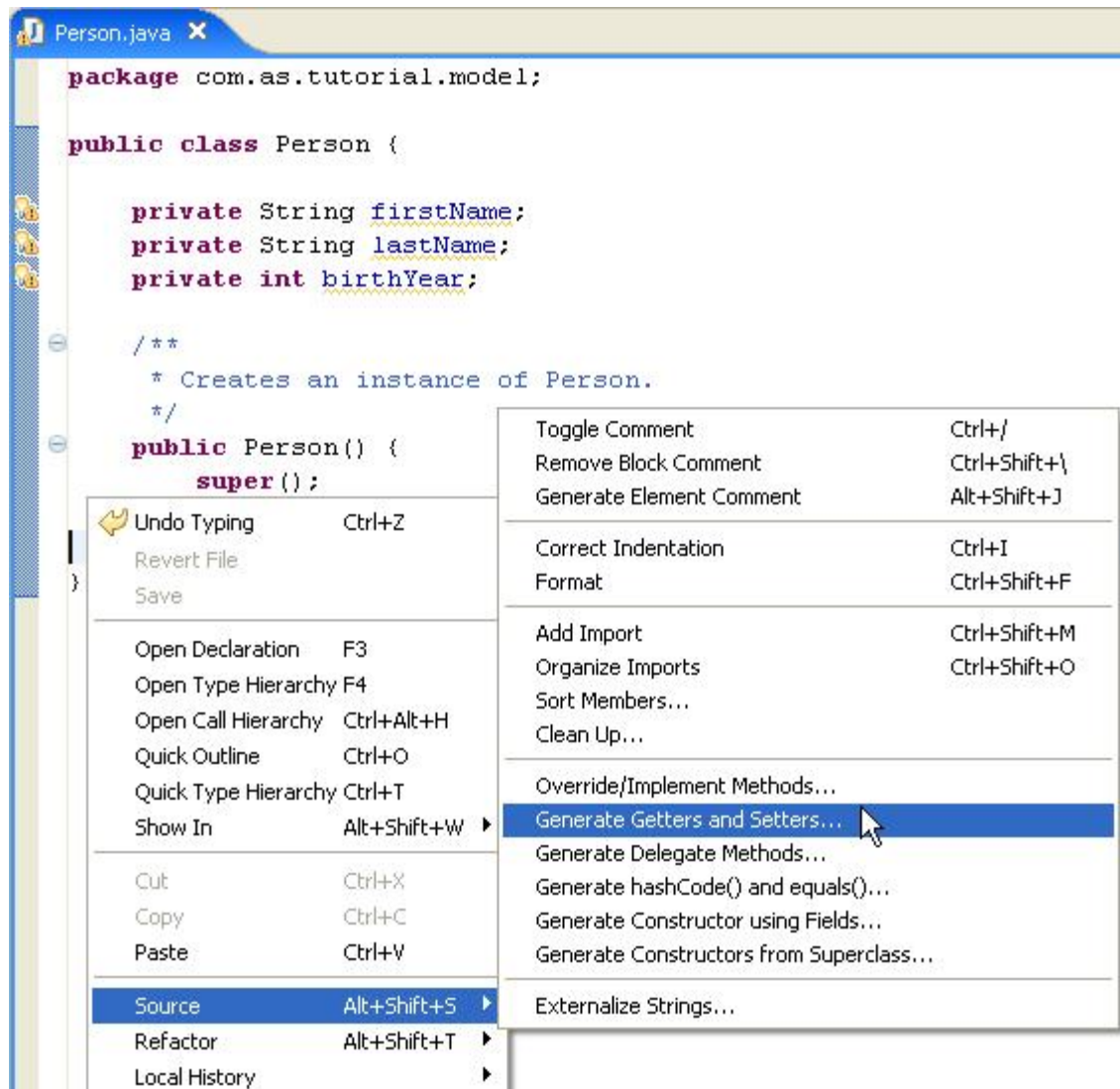
    /**
     * Creates an instance of Person.
     */
    public Person() {
        super();
    }

}
```

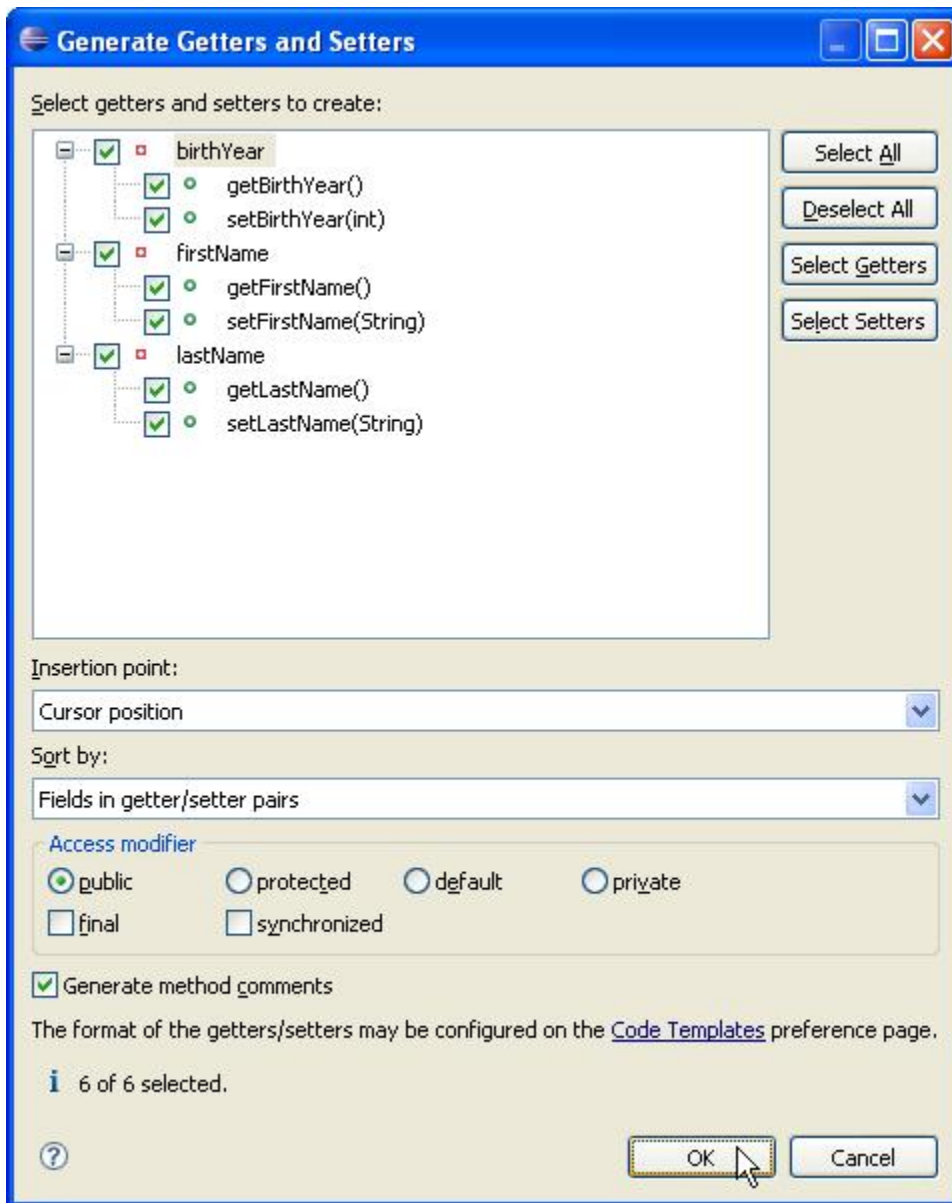


Java classes should generally abide by the Object-Oriented principle of **encapsulation**. Encapsulation means that instance variables (such as `firstName`, `lastName`, and `birthYear`) are private, that is, not directly accessible by other classes. Access to an instance variable is only made available by public **getter** and **setter** methods.

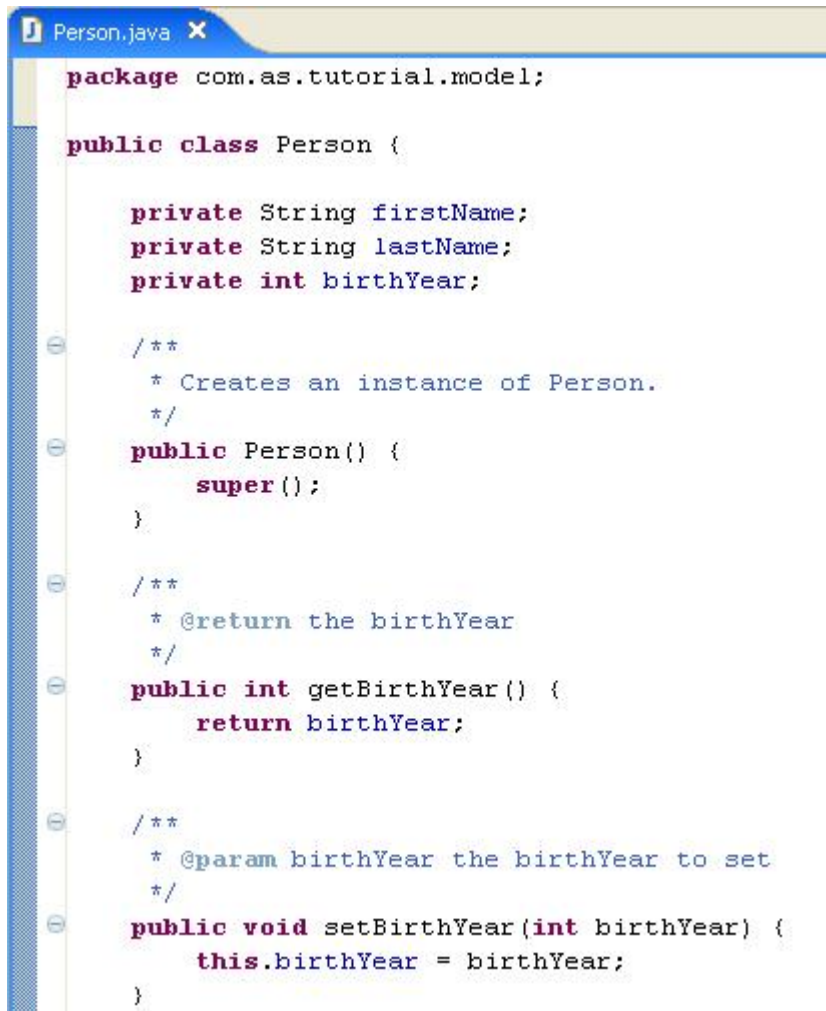
Let's use Eclipse to generate these getter and setter methods for us. In the `Person.java` editor, click once just **above** the `Person` class closing brace `}`. Then right-click and choose the option *Source, Generate Getters and Setters....*



In the “Generate Getters and Setters” dialog, expand each field if you wish. Click the “**Select All**” button so that all methods are selected, check the “**Generate method comments**” box, and click “**OK**”.



Notice that Eclipse generated the getter and setter methods for each field at the current cursor position. The “**public**” access modifier means that these methods are accessible from all other classes in the application (whose instances have a reference to a “Person” instance.) The key here is that the instance variables themselves are private. The Person class grants access to these variables only as it sees fit. For example, a class may very well omit some or all “setter” methods, to prevent its data from being modified. Press **Ctrl+S** to save the Person class.



```
package com.as.tutorial.model;

public class Person {

    private String firstName;
    private String lastName;
    private int birthYear;

    /**
     * Creates an instance of Person.
     */
    public Person() {
        super();
    }

    /**
     * @return the birthYear
     */
    public int getBirthYear() {
        return birthYear;
    }

    /**
     * @param birthYear the birthYear to set
     */
    public void setBirthYear(int birthYear) {
        this.birthYear = birthYear;
    }
}
```

## Create a constructor.

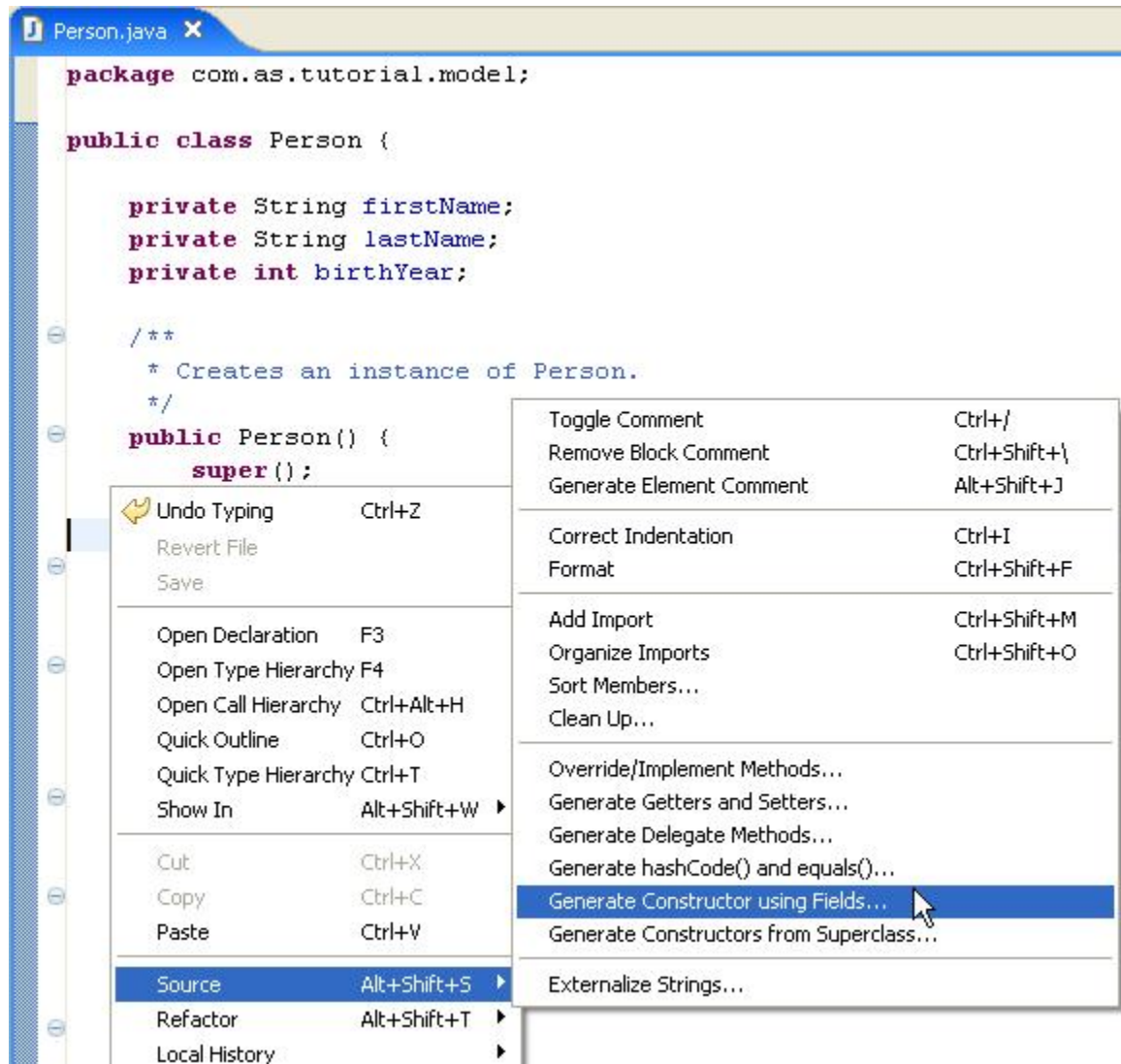
Let's now talk about the Person class **constructor**:

```
public Person() {  
    super();  
}
```

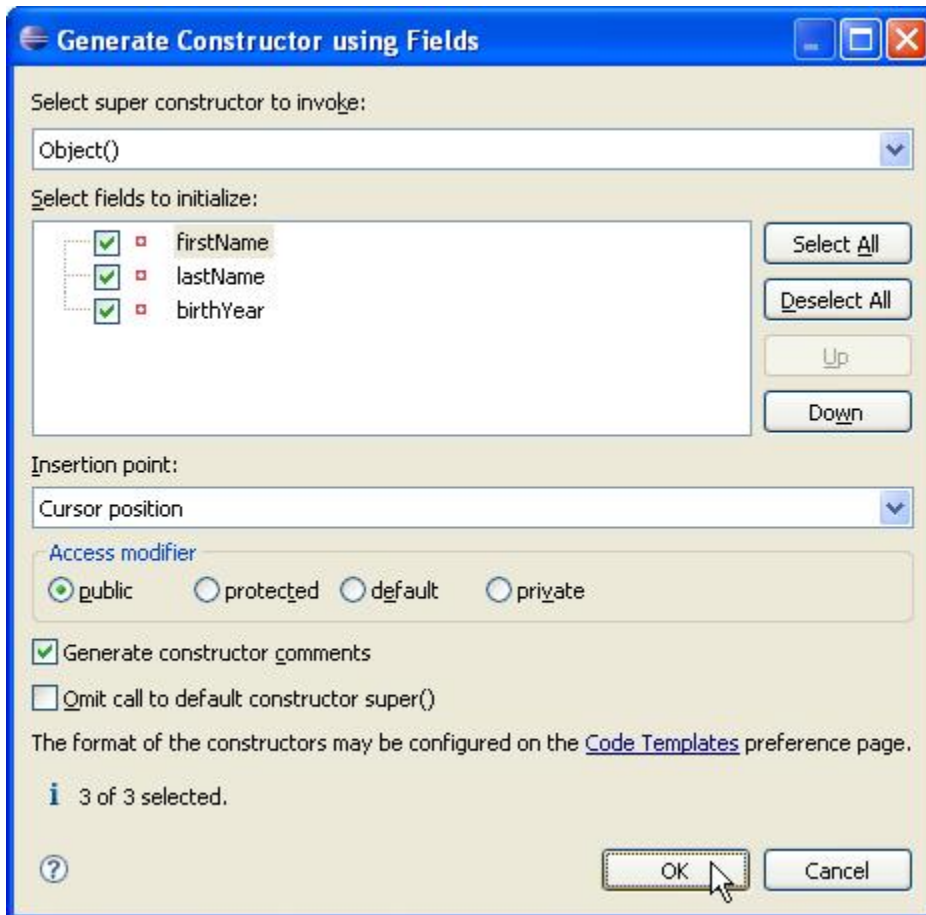
A constructor initializes an instance of a class, when that instance is created at runtime. Note that this constructor offers no **parameters** between the left and right parenthesis in the constructor signature - **public Person()**. Such a constructor is called a “no-arg”, “no-argument”, or “parameterless” constructor. This constructor is the default constructor for the class in the sense that it does not do anything except to invoke (call) the constructor of its super class (parent), which is `java.lang.Object`.

Let's create a Person constructor that initializes its three instance variables `firstName`, `lastName`, and `birthYear`. And, just like the getters and setters, we can ask Eclipse to generate this constructor for us.

In the Person.java editor, use your mouse to click just **below** the closing brace of the no-arg constructor. Then take the option *Source, Generate Constructor using Fields....*

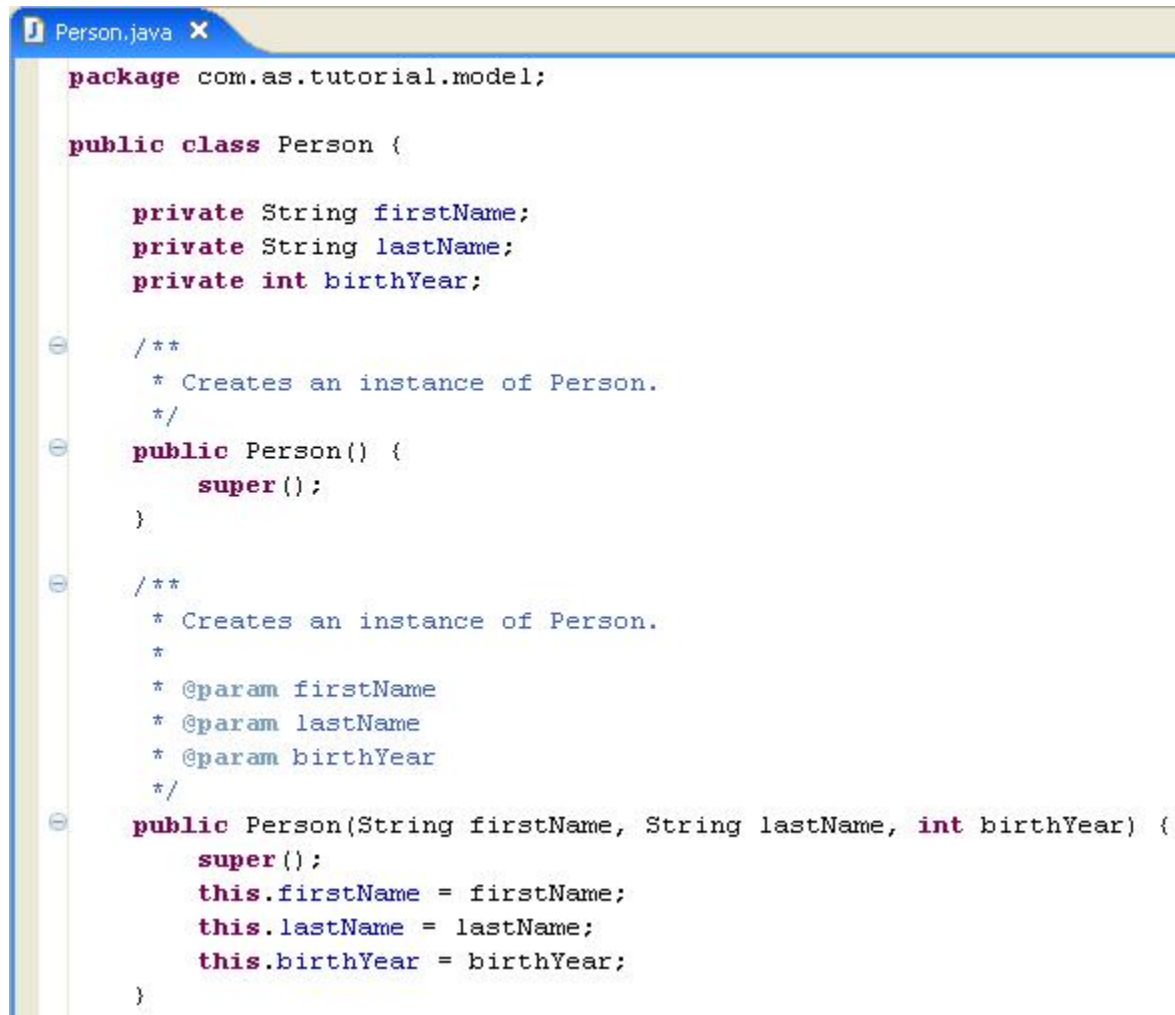


In the “Generate Constructor using Fields” dialog, click the “**Select All**” button so that all fields are selected, check the “**Generate constructor comments**” box, and click “**OK**”.



Notice that Eclipse generated a Person constructor to initialize all three instance variables, at the current cursor position.

Please check that your constructors look like those in the image below, and press **Ctrl+S** to save your changes.



```
Person.java x
package com.as.tutorial.model;

public class Person {

    private String firstName;
    private String lastName;
    private int birthYear;

    /**
     * Creates an instance of Person.
     */
    public Person() {
        super();
    }

    /**
     * Creates an instance of Person.
     *
     * @param firstName
     * @param lastName
     * @param birthYear
     */
    public Person(String firstName, String lastName, int birthYear) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.birthYear = birthYear;
    }
}
```



## **Declare Address instance variables.**

Now it is your turn!

**First**, in the Address class, add the following private instance variables, all of type **String**:

- addressLine1
- city
- state
- zip

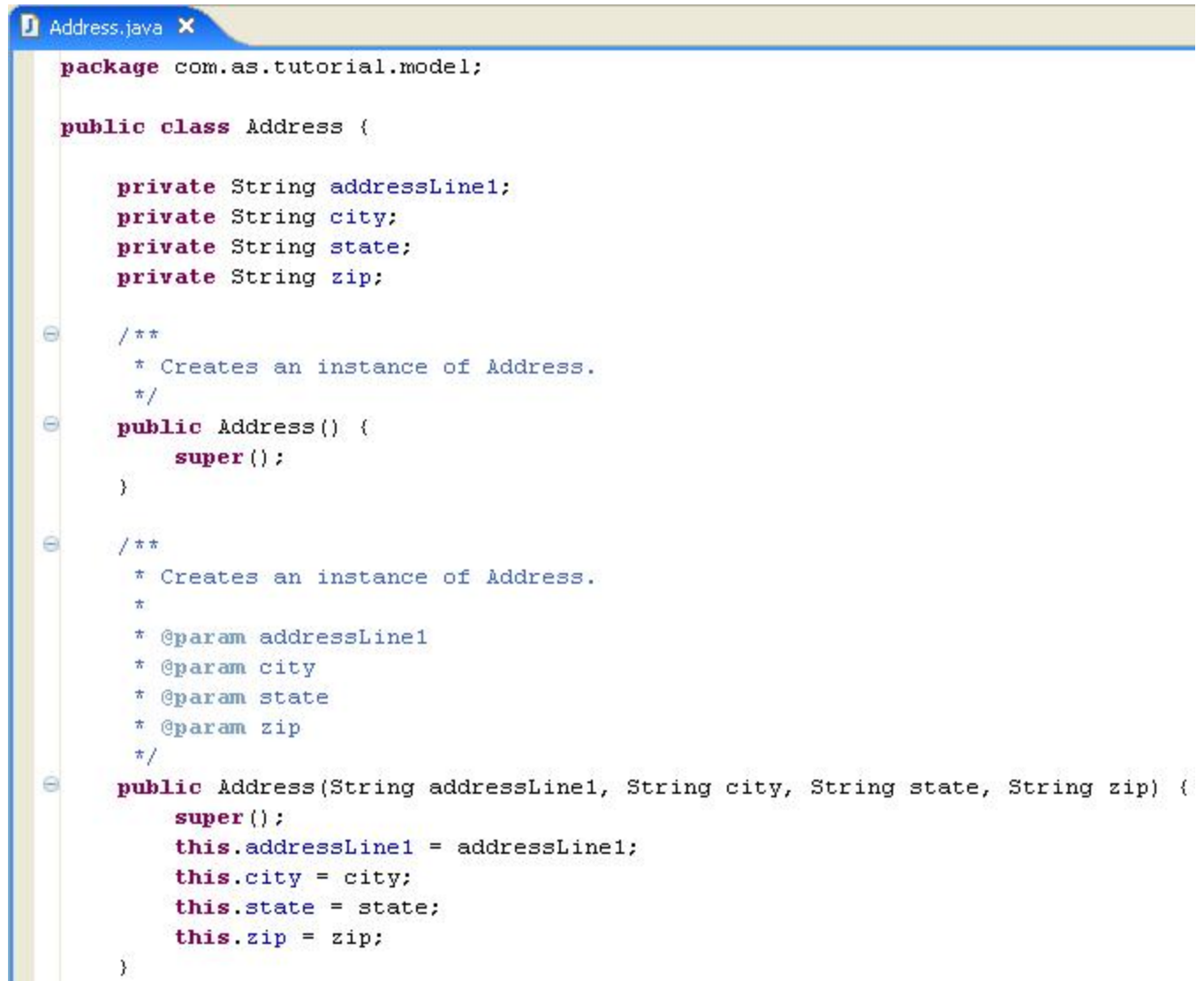
**Second**, use Eclipse to generate getters and setters for these fields.

**Third**, use Eclipse to generate a constructor that initializes all of these fields.

**Fourth**, delete the TODO lines and optionally add constructor Javadoc comments.

**Finally**, press **Ctrl+S** to save your changes to the Address class.

Shown below is a partial look at the completed Address class (getters and setters are not shown.)



```
package com.as.tutorial.model;

public class Address {

    private String addressLine1;
    private String city;
    private String state;
    private String zip;

    /**
     * Creates an instance of Address.
     */
    public Address() {
        super();
    }

    /**
     * Creates an instance of Address.
     *
     * @param addressLine1
     * @param city
     * @param state
     * @param zip
     */
    public Address(String addressLine1, String city, String state, String zip) {
        super();
        this.addressLine1 = addressLine1;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }
}
```

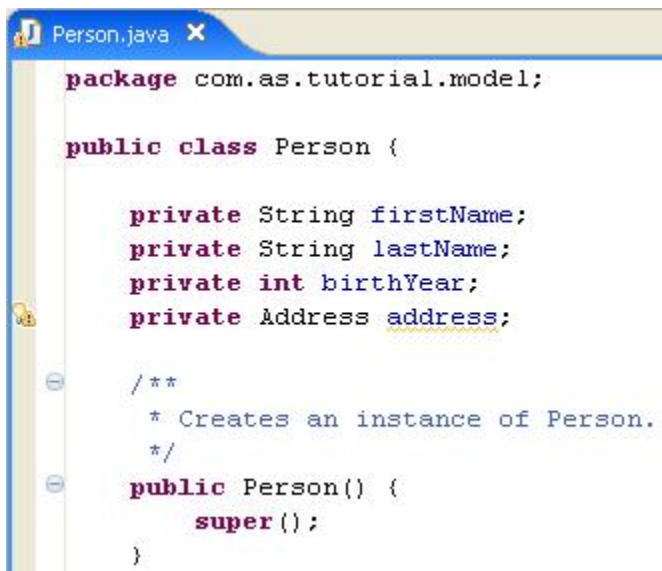
## Object association.

Next we'll assume that in the real world, each **Person** has an **Address**. Let's model that relationship by giving a Person an instance variable of type Address.

Double-click the Person class to open it in an editor, and at the end of the instance variable list simply type the following line.

```
private Address address;
```

Press **Ctrl+S** to save your changes.



```
Person.java x
package com.as.tutorial.model;

public class Person {

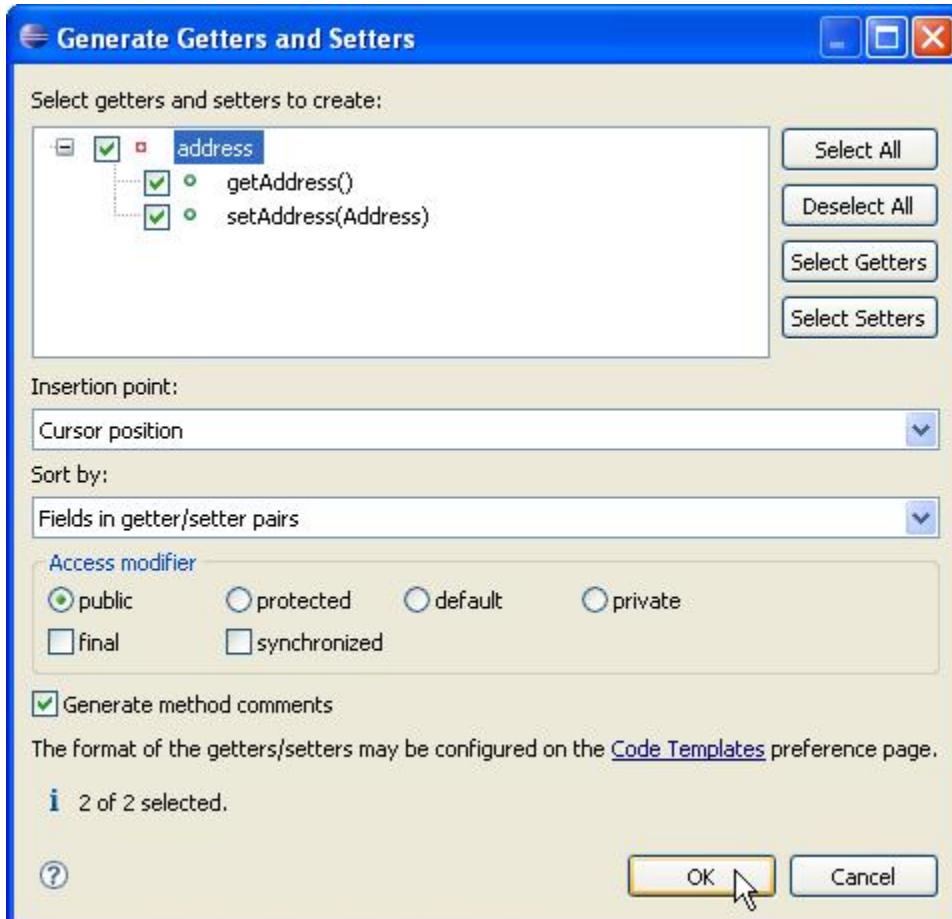
    private String firstName;
    private String lastName;
    private int birthYear;
    private Address address;

    /**
     * Creates an instance of Person.
     */
    public Person() {
        super();
    }
}
```

This is an example of **object association**. In this case we say that **(a) Person has (an) Address**.

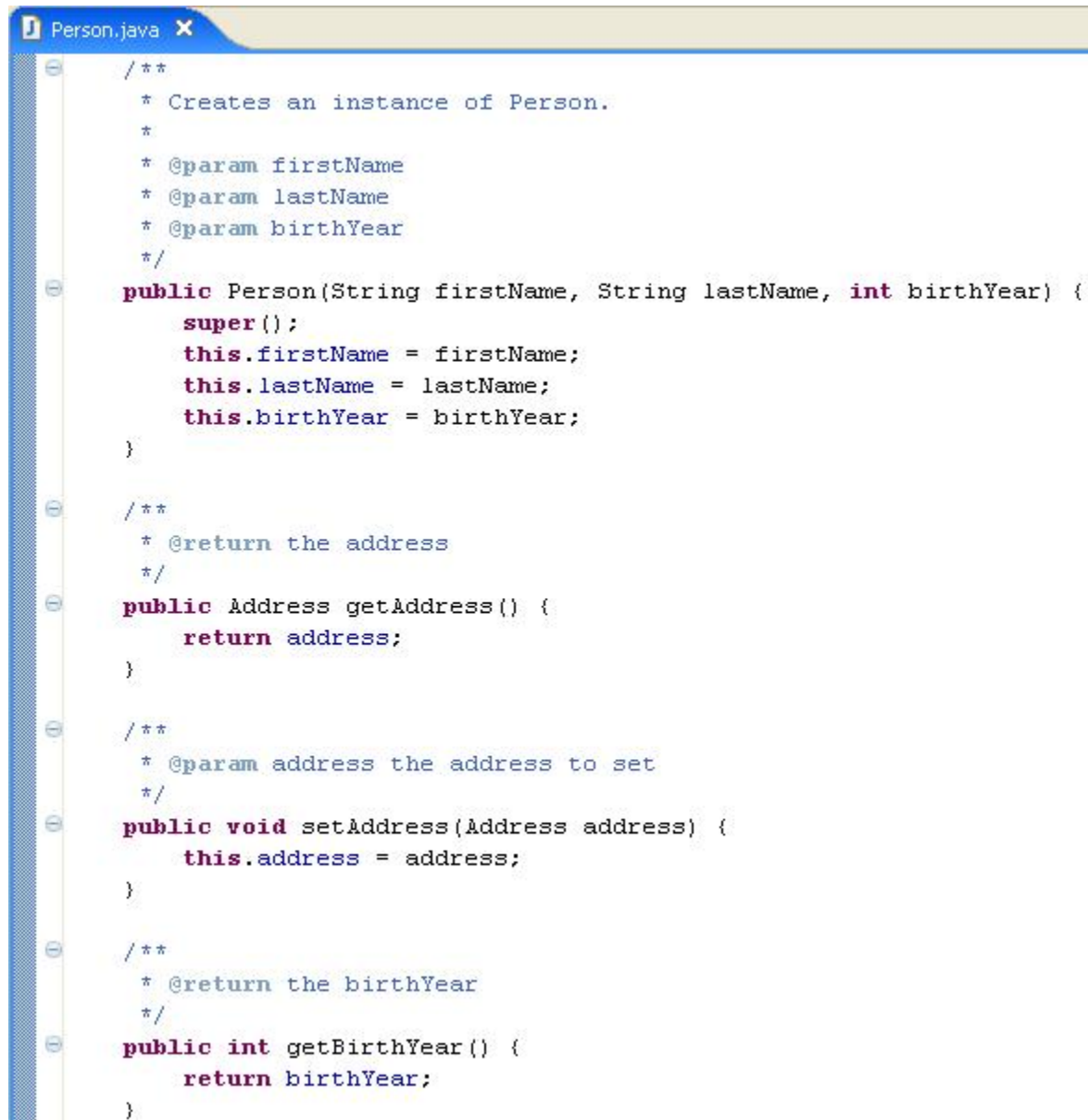
Let's create getter and setter methods for Address. In the Person.java editor, click just above the "getBirthYear()" method. Then right-click and again take the option *Source, Generate Getters and Setters...*

In the "Generate Getters and Setters" dialog, **check** the "address" field, the other defaults in the dialog should be fine, and click "OK".



Now you should have a getter and setter for the private “address” field, just before the “getBirthYear()” method.

Don’t forget to press **Ctrl+S** to save your changes.



```
Person.java X
/**
 * Creates an instance of Person.
 *
 * @param firstName
 * @param lastName
 * @param birthYear
 */
public Person(String firstName, String lastName, int birthYear) {
    super();
    this.firstName = firstName;
    this.lastName = lastName;
    this.birthYear = birthYear;
}

/**
 * @return the address
 */
public Address getAddress() {
    return address;
}

/**
 * @param address the address to set
 */
public void setAddress(Address address) {
    this.address = address;
}

/**
 * @return the birthYear
 */
public int getBirthYear() {
    return birthYear;
}
```

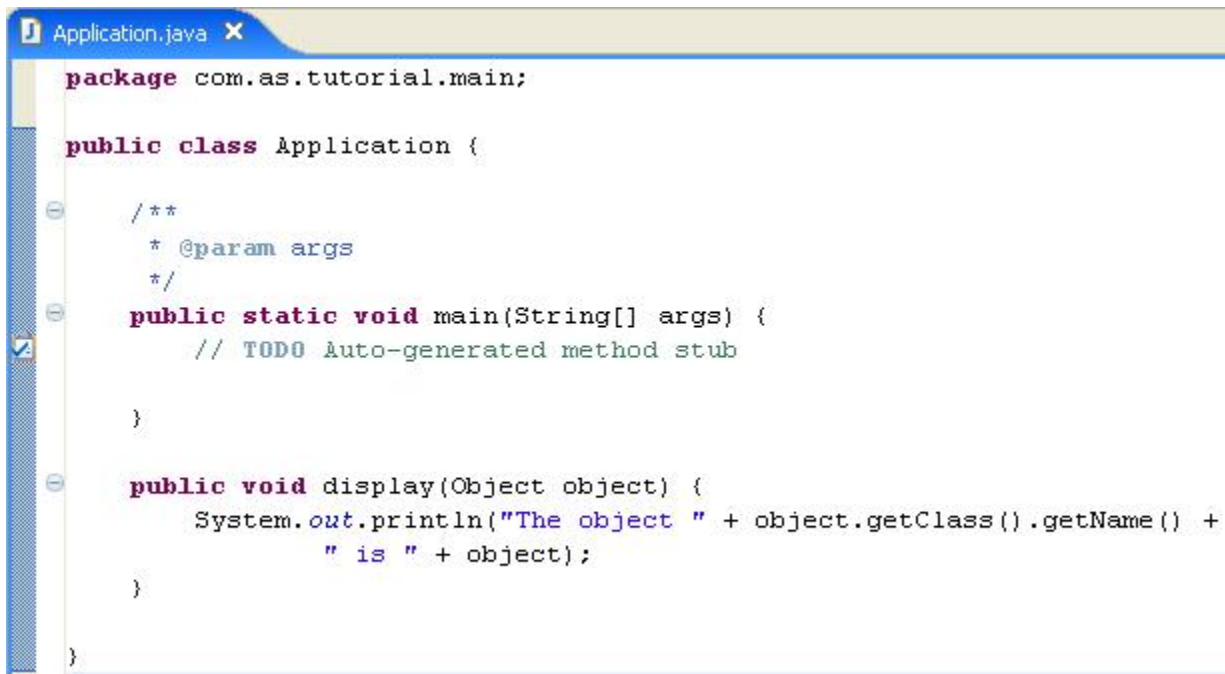
## Write application code.

Now we will go back to our main Application class and write some code that we can execute and that will produce some results!

In the Package Explorer, double-click the Application class to open it in an editor.

Type in the display(...) method exactly as it is shown below, or copy it and paste it in to the editor at the point shown. Press **Ctrl+S** to save the change.

```
public void display(Object object) {  
    System.out.println("The object " + object.getClass().getName() +  
        " is " + object);  
}
```



If you are guessing that our main Application class is going to display objects, like Person and Address, you are correct! Notice the following points about the **method** that we just created:

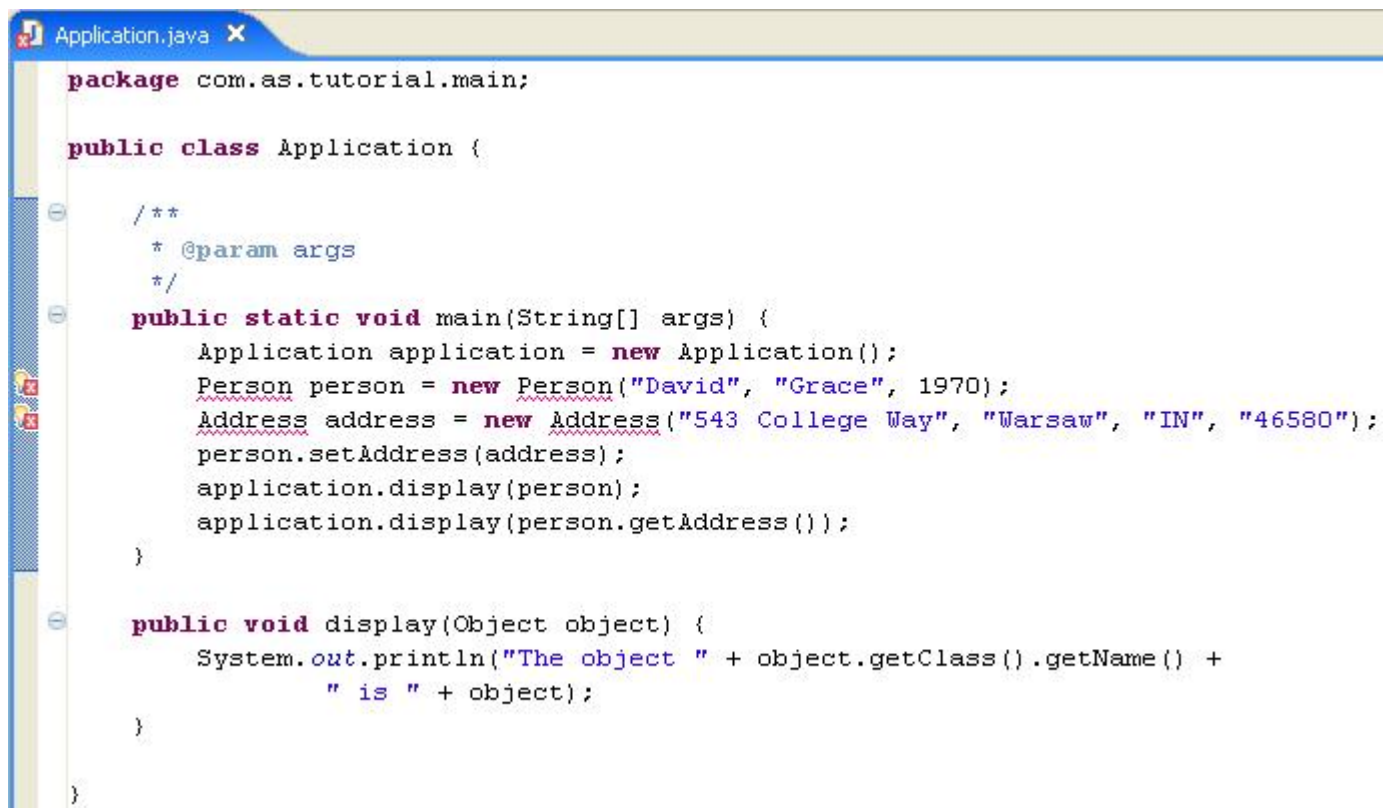
- It is **public**, so that it can be called by any class that has an instance of the Application class.
- It is **void**, so it does not return a value.
- It takes an instance of **java.lang.Object** as its input parameter. Since java.lang.Object is at the top of the Java inheritance hierarchy, in essence this method can accept any object as input (like a Person or an Address).

Now we will write some code in the Application “main” method that will call our display(...) method. Type in the following six lines of code as shown, or copy and paste them in (notice that we have deleted the // TODO). Press **Ctrl+S** to save the change.

```
Application application = new Application();
Person person = new Person("David", "Grace", 1970);
Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
person.setAddress(address);
application.display(person);
application.display(person.getAddress());
```



After you have added the code to the main(...) method, your Application.java editor will look like that in the image below. In a moment we will fix those red errors.



```
Application.java x
package com.as.tutorial.main;

public class Application {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Application application = new Application();
        Person person = new Person("David", "Grace", 1970);
        Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
        person.setAddress(address);
        application.display(person);
        application.display(person.getAddress());
    }

    public void display(Object object) {
        System.out.println("The object " + object.getClass().getName() +
            " is " + object);
    }
}
```

What is the purpose for the lines of code we have written? Let's quickly look at each line.

```
Application application = new Application();
```

- Creates an instance of our Application class, using the default constructor that the compiler generated for us.

```
Person person = new Person("David", "Grace", 1970);
```

- Creates an instance of our Person class, referenced by the local variable *person*.

```
Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
```

- Creates an instance of our Address class, referenced by the local variable *address*.

```
person.setAddress(address);
```

- Gives the "Person" instance a reference to the "Address" instance.

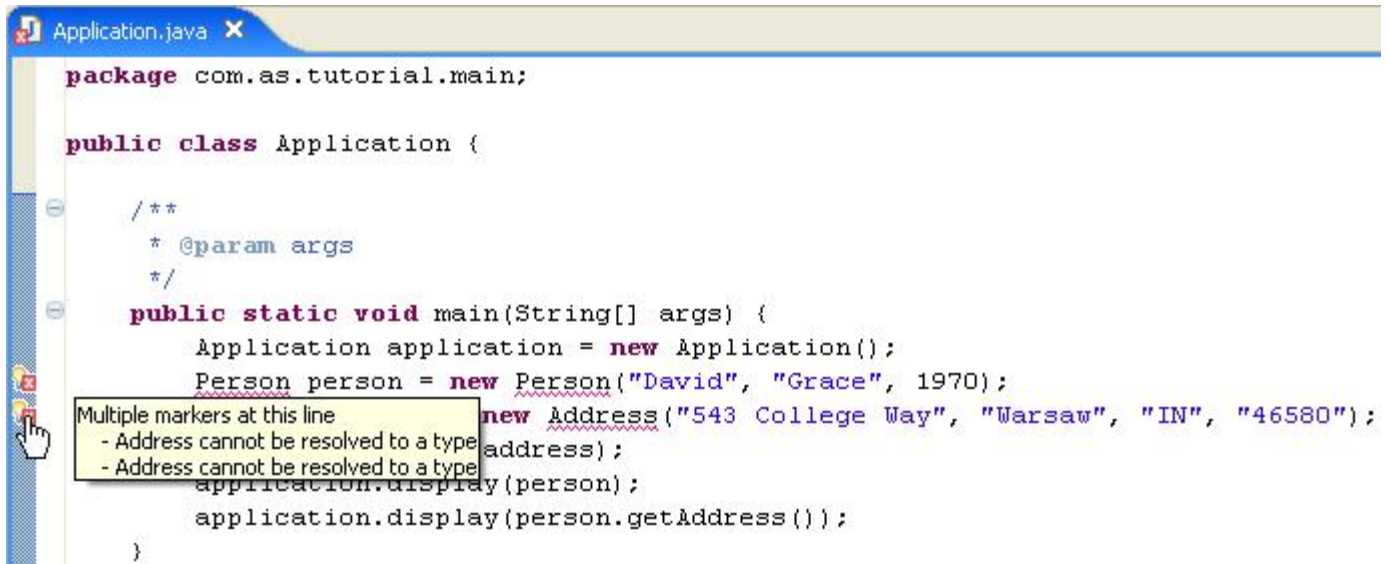
```
application.display(person);
```

- Calls the Application class display(...) method and passes the Person instance.

```
application.display(person.getAddress());
```

- Calls the Application class display(...) method and passes the Person's Address instance.

Now we will fix those errors. You can position your mouse pointer over the error to get a description of the problem:

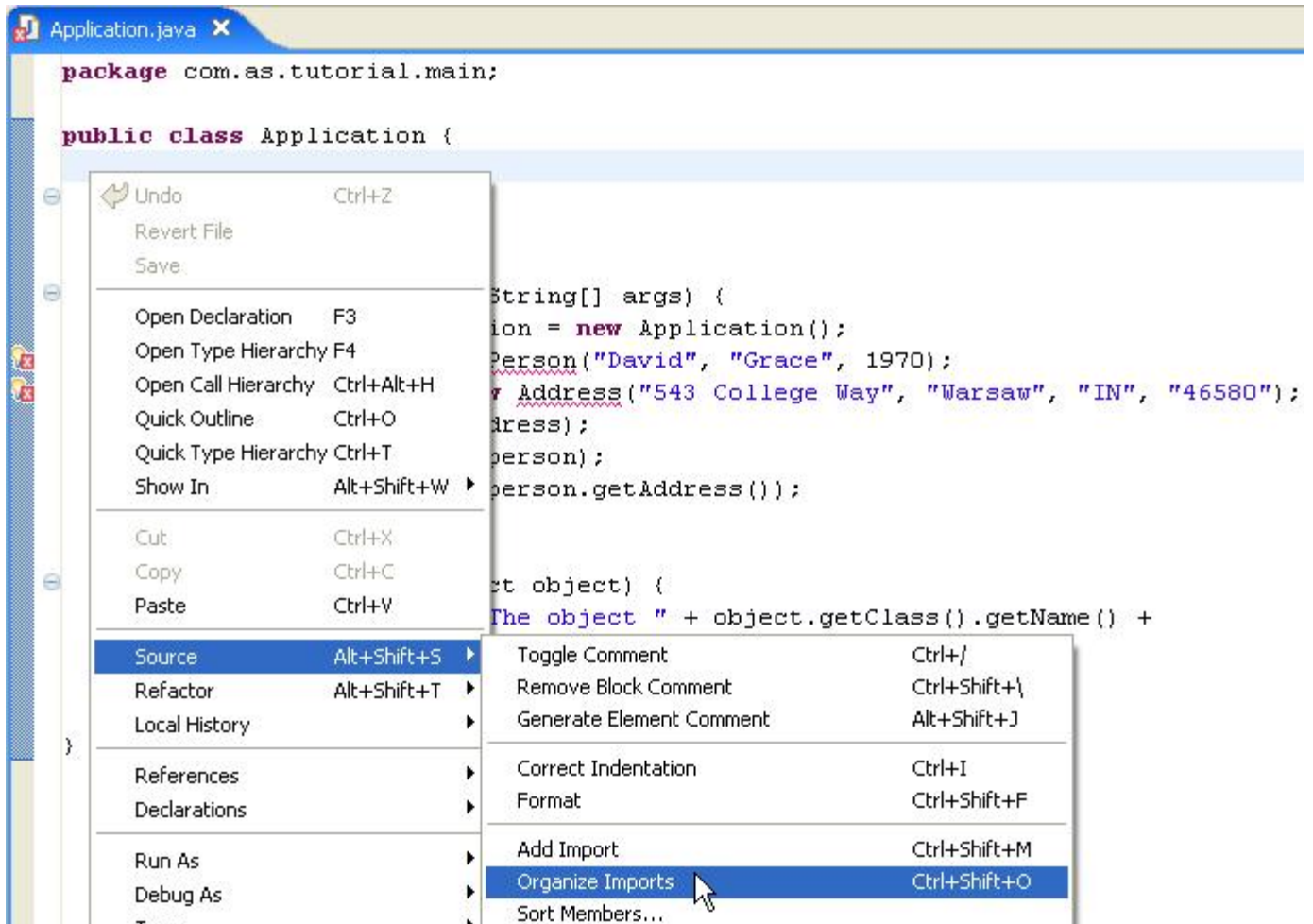


You can also look in the “Problems” view (Window → Show View → Problems). You can double-click a problem line to open the resource in an Eclipse editor, with the problem highlighted.

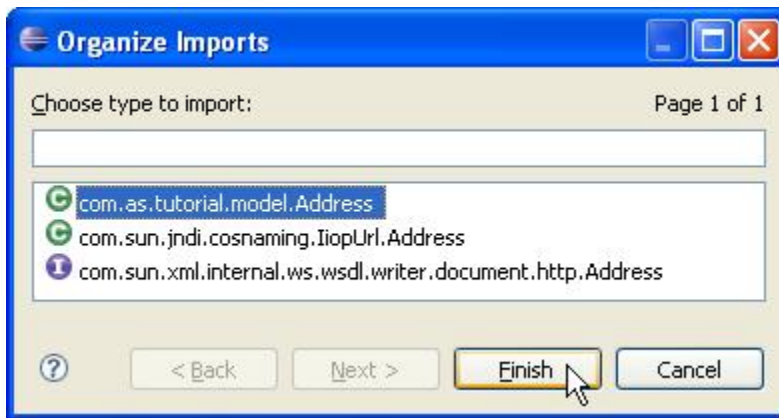
Description	Resource	Path	Location
<b>Errors (4 items)</b>			
Address cannot be resolved to a type	Application.java	Introduction/src/com/as/tutorial/main	line 11
Address cannot be resolved to a type	Application.java	Introduction/src/com/as/tutorial/main	line 11
Person cannot be resolved to a type	Application.java	Introduction/src/com/as/tutorial/main	line 10
Person cannot be resolved to a type	Application.java	Introduction/src/com/as/tutorial/main	line 10

The problem here is that the **Application** class cannot identify the **Person** and **Address** classes, because Person and Address are in a different package than Application!

There are multiple ways to fix this problem. We will show you one technique, which you will probably use often. Click anywhere in the Application.java editor, then right-click and take the option **Source, Organize Imports** (or type **Ctrl+Shift+O**)



In this case, Eclipse cannot figure out which Address class you want, so it displays the “**Organize Imports**” dialog. Select “com.as.tutorial.model.Address” and click “**Finish**”.



Eclipse adds the following two import statements in Application.java, just below the package declaration. Press **Ctrl+S** to save the change, and the problem is fixed!

Note that we have also added a Javadoc comment for the main(...) method.

```
Application.java x
package com.as.tutorial.main;

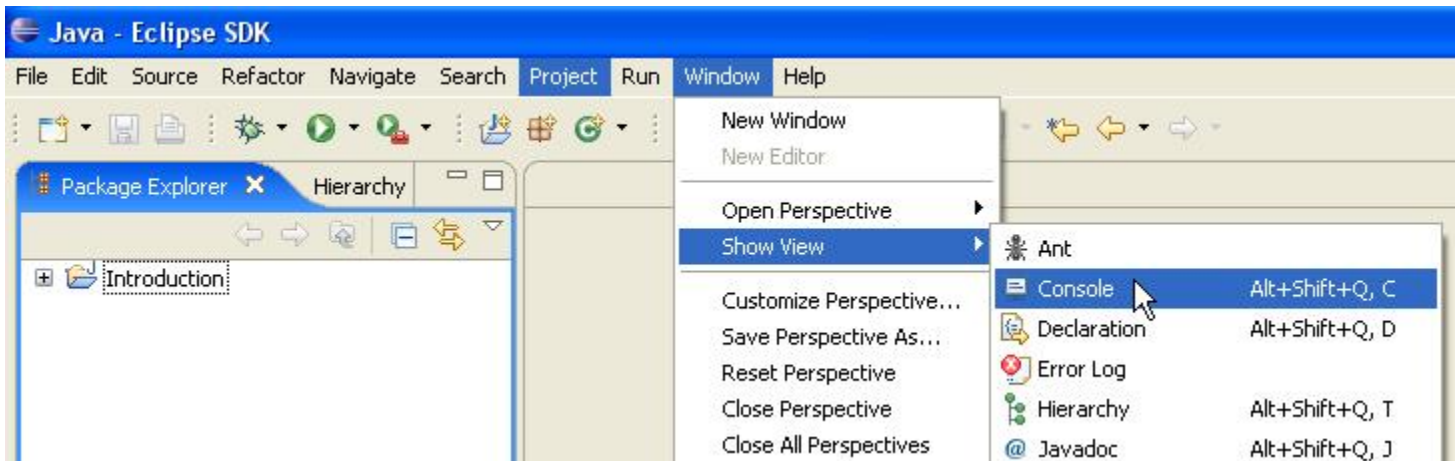
import com.as.tutorial.model.Address;
import com.as.tutorial.model.Person;

public class Application {

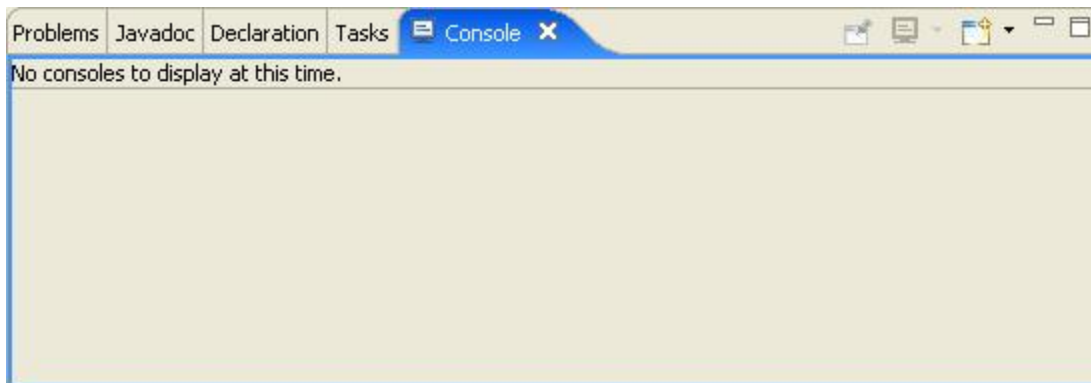
    /**
     * Application entry method.
     *
     * @param args
     */
    public static void main(String[] args) {
        Application application = new Application();
        Person person = new Person("David", "Grace", 1970);
        Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
        person.setAddress(address);
        application.display(person);
        application.display(person.getAddress());
    }
}
```

## Run the application.

Now we will actually execute our code. We need to be able to see the results so **first open the Console View** if it is not already open. (Window → Show View → Console.)



The Console View initially looks like that in the image below.

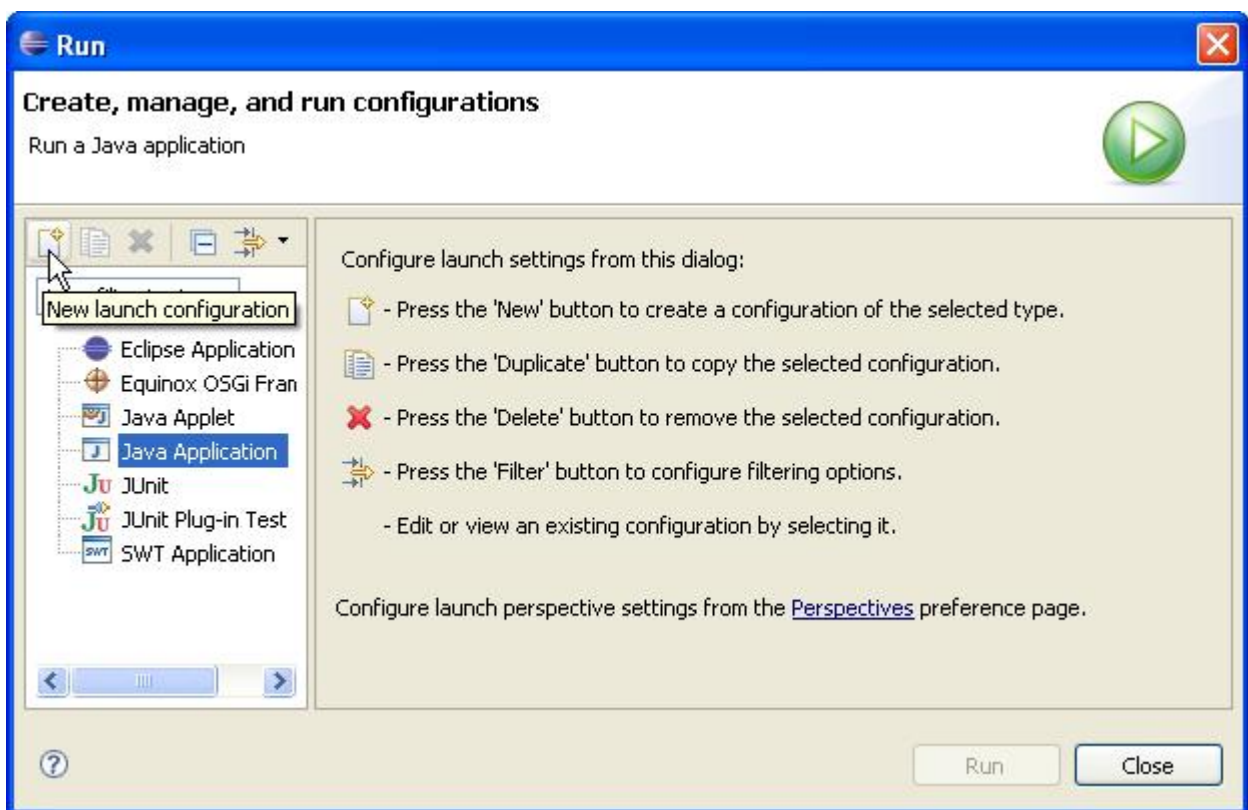




Next, select the “Introduction” project then take the menu option **Run, Run....**



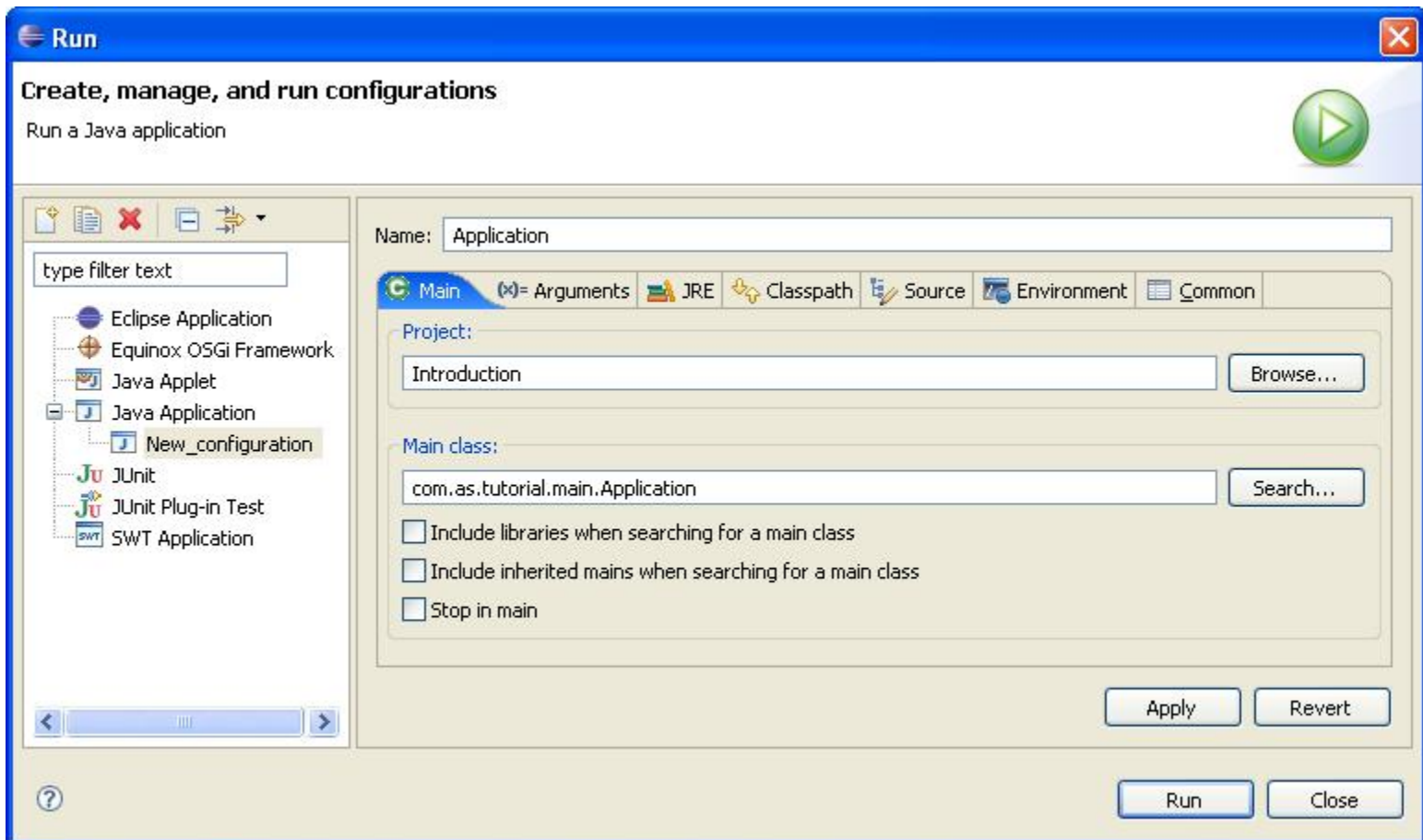
In the “Run” dialog box select the configuration “**Java Application**” and click the “**New launch configuration**” button.





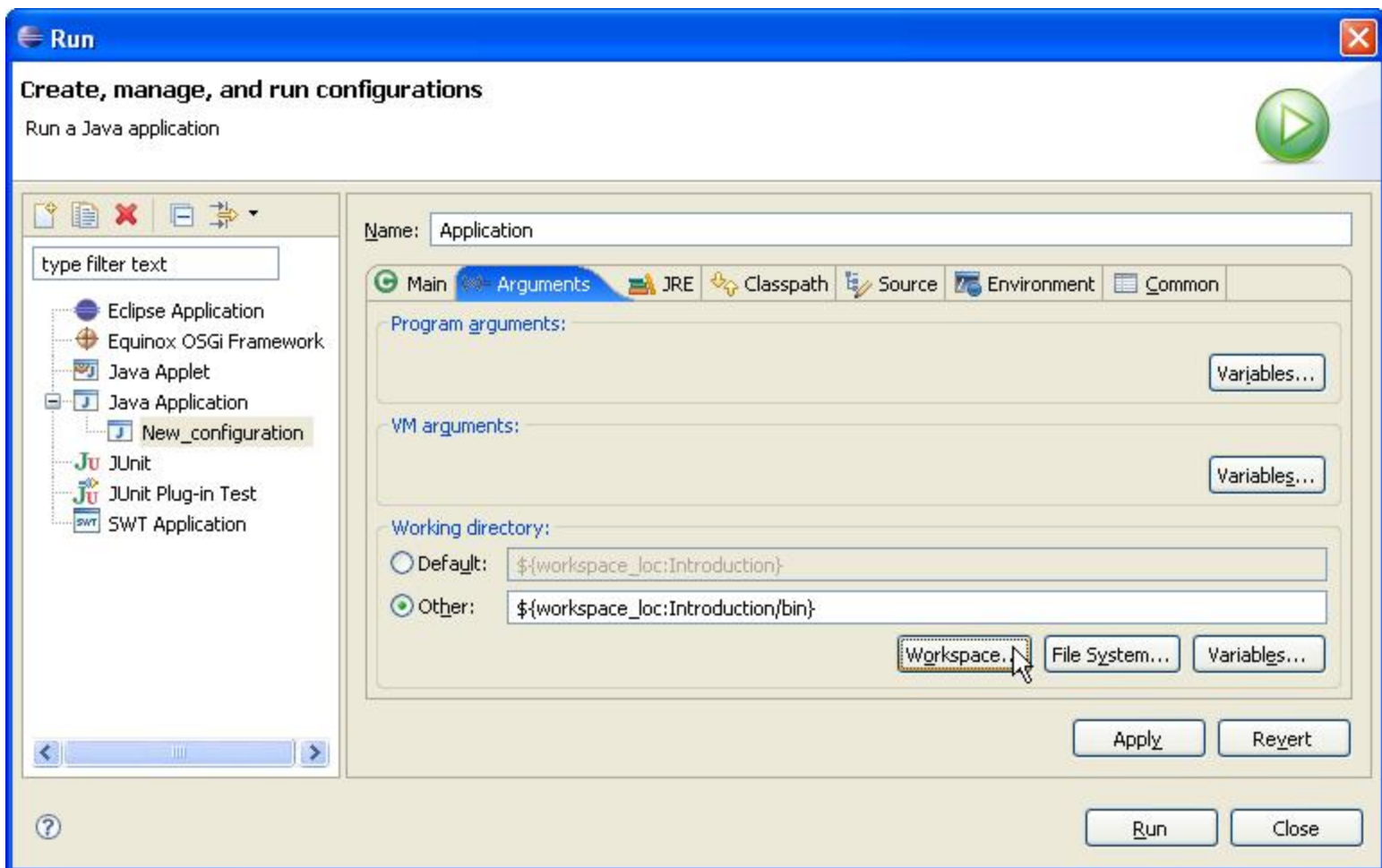
In the “**Main**” tab, enter the following information:

- For the configuration **Name**, type **Application**.
- For the **Project**, type **Introduction** (if that is not already the value.)
- For the **Main class**, type **com.as.tutorial.main.Application**.

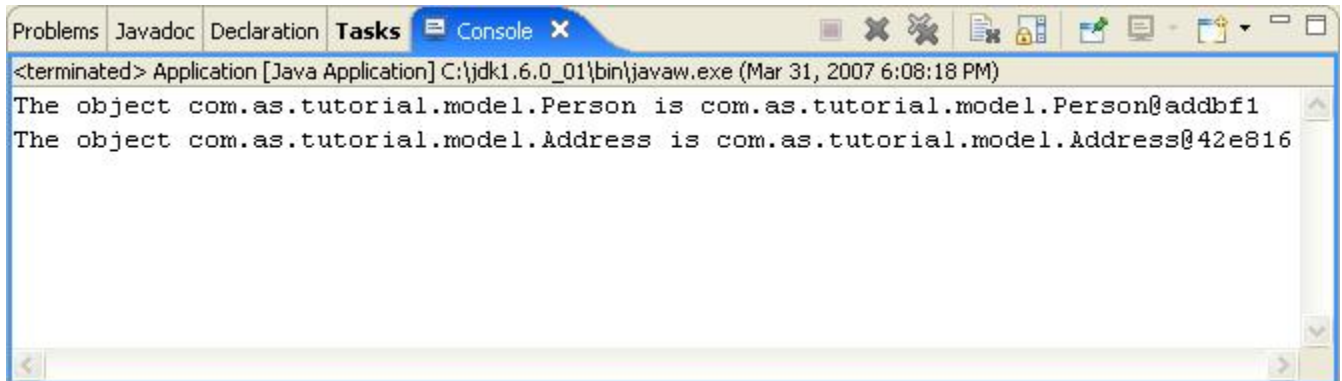


Now click the “**Arguments**” tab and enter the following information:

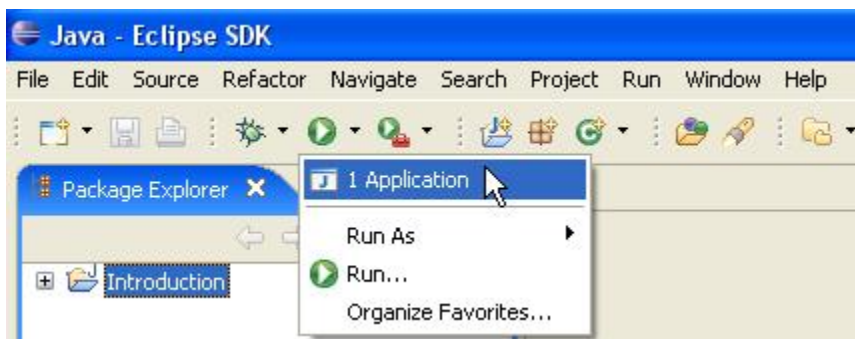
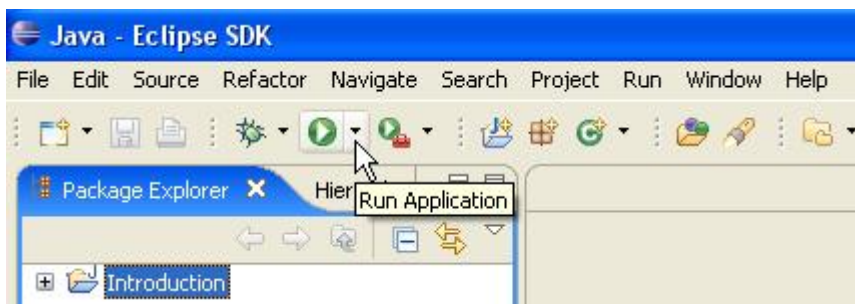
- Click the Working directory “Other” radio button.
- Click the **Workspace...** button and in the “**Folder Selection**” dialog drill down and select **Introduction/bin** and click OK. Back in the Run dialog your “Working directory” should appear as in the image below. The reason, you will recall, is that when we created our project we specified “Introduction/bin” as the “Default output folder” (for the compiled .class files.)
- Now click the **Run** button to execute our application.



After the application executes, the Console, which you previously opened, should have the following output:



Note that you can subsequently run the application from the toolbar. In the “Run” toolbar drop-down menu, select our “Application” configuration to run the application and see the output again.



## Examine application output.

Let's examine the output from the first run of the application:

The object com.as.tutorial.model.Person is com.as.tutorial.model.Person@addbf1  
The object com.as.tutorial.model.Address is com.as.tutorial.model.Address@42e816

This output came from the following method in our Application class, and in particular from the “**println(...)**” method. Note that the plus (+) symbols perform String concatenation.

```
public void display(Object object) {  
    System.out.println("The object " + object.getClass().getName() +  
        " is " + object);  
}
```

When we examine the output lines, it is fairly clear that the fragments “com.as.tutorial.model.Person” and “com.as.tutorial.model.Address” came from the code

```
object.getClass().getName()
```

In other words this code adds, to the output, the fully-qualified class name of the object that is passed in to the display(...) method.

Furthermore, we deduce that the output fragments “com.as.tutorial.model.Person@addbf1” and “com.as.tutorial.model.Address@42e816” came from the parameter “object” within the println(...) method.

```
... + object);
```

In other words, the output includes a String representation of the **object** that is passed in to the display(...) method.

How is this String representation created? The answer lies in the superclass of our Person and Address classes. Remember that the superclass is **java.lang.Object** (since there is no **extends** keyword in the class signature.)

The String representation of a class is implemented by its **toString()** method. Since neither our Person nor our Address class has an explicit toString() method, our classes inherit a toString() implementation from **java.lang.Object**, which creates a String representation of an instance of a class that looks like “com.as.tutorial.model.Person@addbf1”.

But when you examine the `display(...)` method body, you see that the code does not call the **object** parameter's `toString()` method. The reason is that when Java needs the String representation of an object, it calls the `toString()` method automatically. So our `display(...)` method could very well have been coded to call the `toString()` method explicitly, as follows, and our output will be the same.

```
public void display(Object object) {  
    System.out.println("The object " + object.getClass().getName() +  
        " is " + object.toString());  
}
```

How would you know that our “object” parameter of type `java.lang.Object` offers such a `toString()` method? This question allows us to look at one of the most useful features of Eclipse, **Ctrl+Spacebar**.

Within Eclipse, open our Application class in a Java editor. In the `display(...)` method, type a **period** after the `object` parameter in the `println(...)` method. Then press **Ctrl+Spacebar**.

```
public void display(Object object) {  
    System.out.println("The object " + object.getClass().getName() +  
        " is " + object.);  
}
```

- hashCode() int - Object
- toString() String - Object
- equals(Object obj) boolean - Object
- getClass() Class<? extends Object> - Object
- notify() void - Object
- notifyAll() void - Object
- wait() void - Object
- wait(long timeout) void - Object
- wait(long timeout, int nanos) void - Object

Press 'Ctrl+Space' to show Template Proposals

Returns a string representation of the object. The `toString` method returns a string that represents this object. The result should be a short and informative representation that is easy to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the hexadecimal representation of the hash code of the object. In other words, this method returns

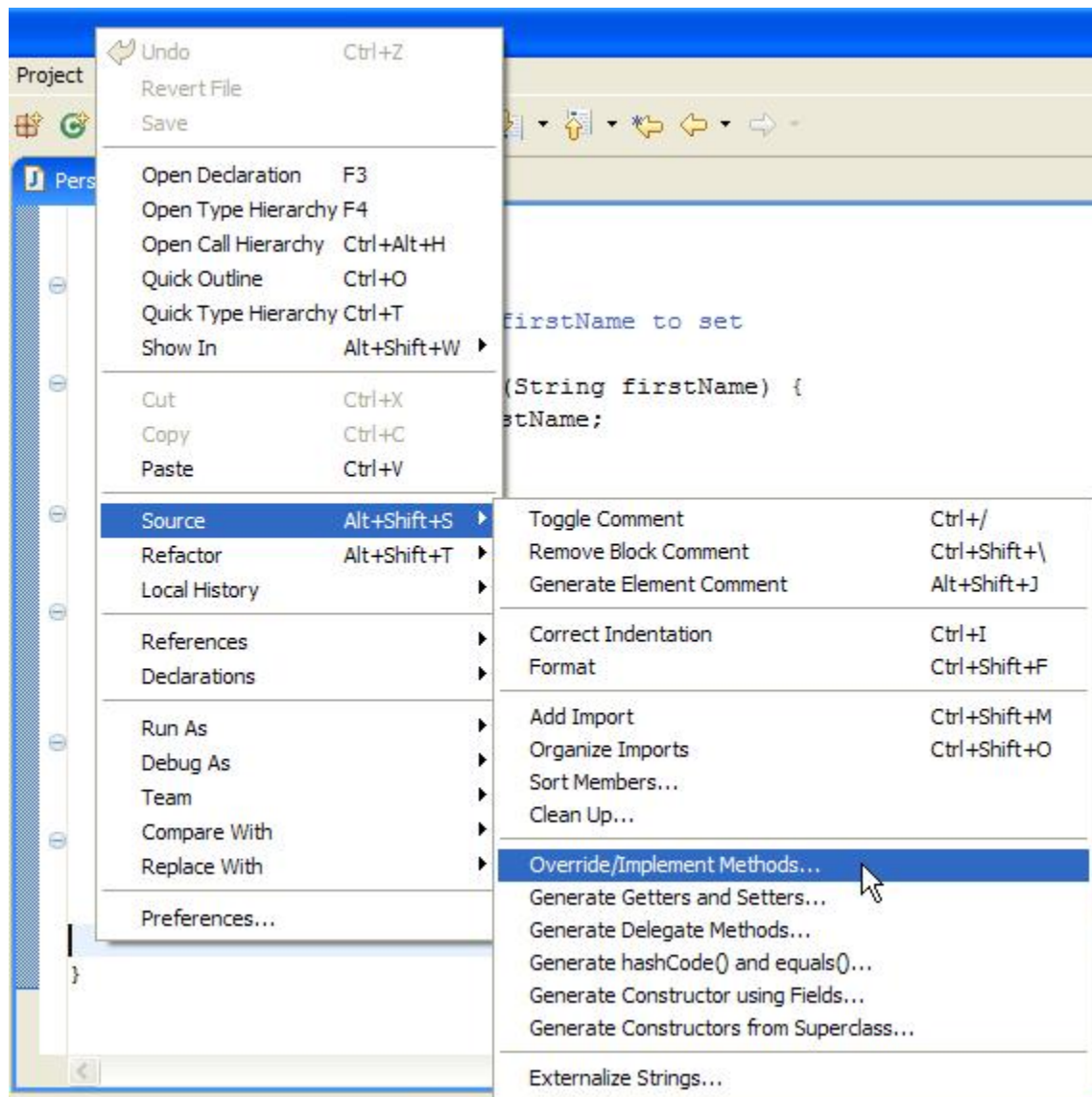
Notice that Eclipse shows you the members (data and method) available for instances of the particular class. If the source code is attached, Eclipse even displays the Javadoc class documentation for a selected member. You can double-click a member to select it and add it to the source code.

Either add `.toString()` to the end of the **object** parameter and save the change, or leave it as it is – the output will remain unchanged.

## Override toString().

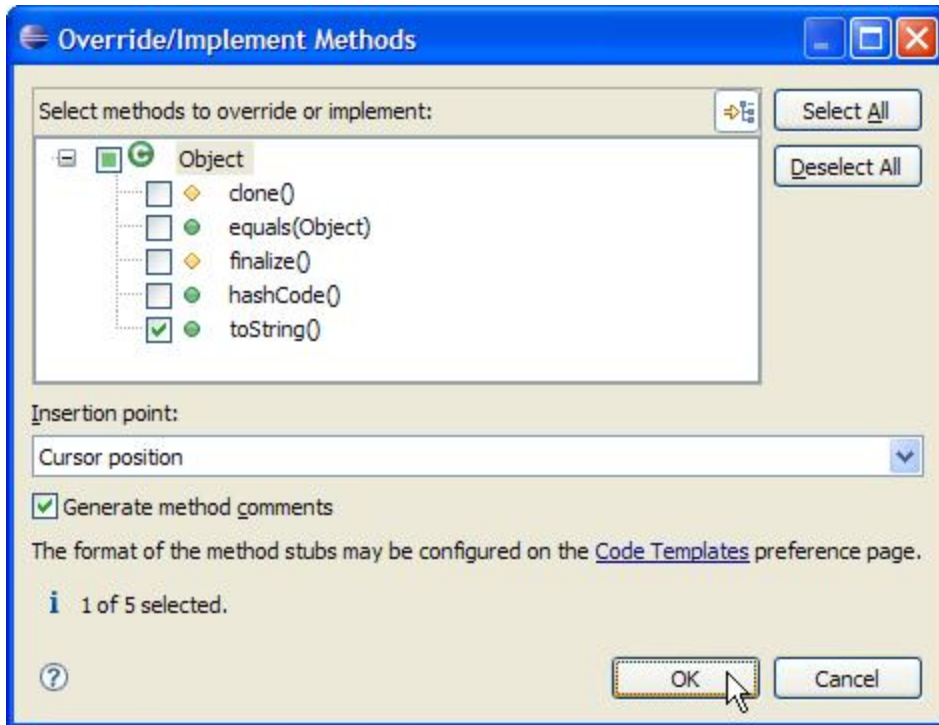
Let us now **override** the toString() method in our Person and Address classes. To *override a method* means to *replace the superclass implementation in a subclass*.

Open the Person class in an Eclipse Java editor. Click just before the closing brace in the class. Then right-click and take the option **Source, Override/Implement Methods....**





In the “**Override/Implement Methods**” dialog, check the “toString()” check box and click the “OK” button.



Eclipse generates a default toString() implementation in the Person class.

```
/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
    // TODO Auto-generated method stub
    return super.toString();
}
```



Change the implementation of the toString() method in the **Person** class to that shown in the image below, or just copy and paste the following lines into the method body. Add some Javadoc, and press **Ctrl+S** to save your changes.

```
return "My name is " + this.getFirstName() +  
       " and my birth year is " + this.getBirthYear();
```

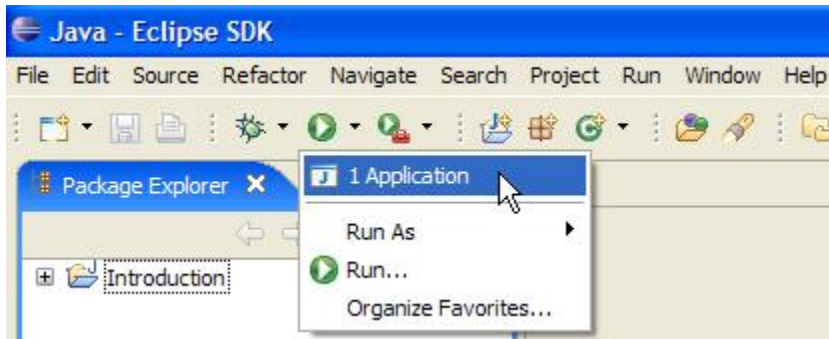
```
/**  
 * Person toString() method.  
 */  
@Override  
public String toString() {  
    return "My name is " + this.getFirstName() +  
           " and my birth year is " + this.getBirthYear();  
}
```

In a similar manner, override the implementation of the toString() method in the **Address** class to that shown in the image below, or just copy and paste the following lines into the method body. Press **Ctrl+S** to save your changes.

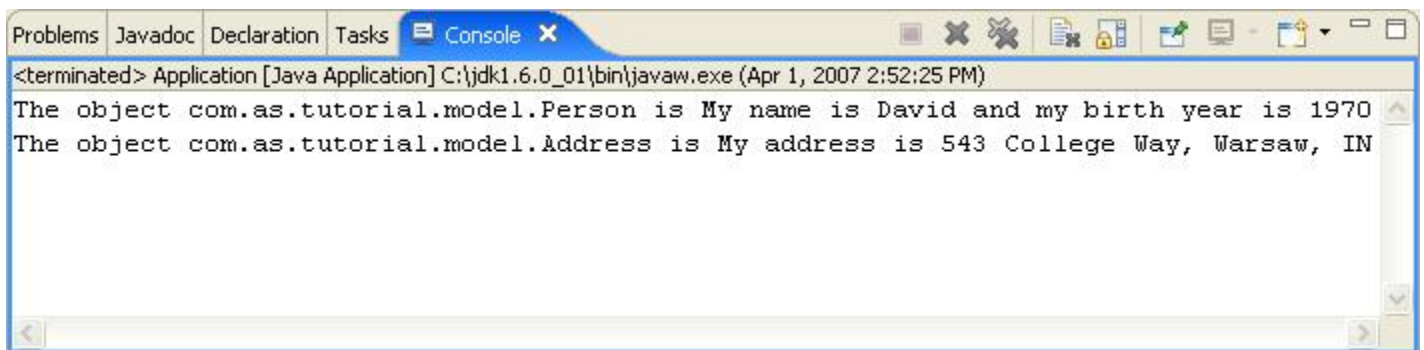
```
return "My address is " + this.getAddressLine1() +  
       ", " + this.getCity() + ", " + this.getState();
```

```
/**  
 * Address toString() method.  
 */  
@Override  
public String toString() {  
    return "My address is " + this.getAddressLine1() +  
           ", " + this.getCity() + ", " + this.getState();  
}
```

Now run the application again. Use the “Run” toolbar drop-down menu and select our “Application” configuration.



Now the Console output displays the results using our custom toString() methods!



Let's revisit the code in the Application display(...) method:

```
public void display(Object object) {  
    System.out.println("The object " + object.getClass().getName() +  
        " is " + object);  
}
```

Consider the output after the following lines execute in the **Application main(...)** method:

Executable line:     application.display(person);

Output:             The object com.as.tutorial.model.Person is My name is David and my  
                      birth year is 1970

Executable line:     application.display(person.getAddress());

Output:             The object com.as.tutorial.model.Address is My address is 543 College  
                      Way, Warsaw, IN

In the first case we pass the display(...) method an instance of the Person class as an Object, and in the second case we pass the display(...) method an instance of the Address class as an Object. But in both cases Java calls the correct toString() method (and we can see this in the output.)

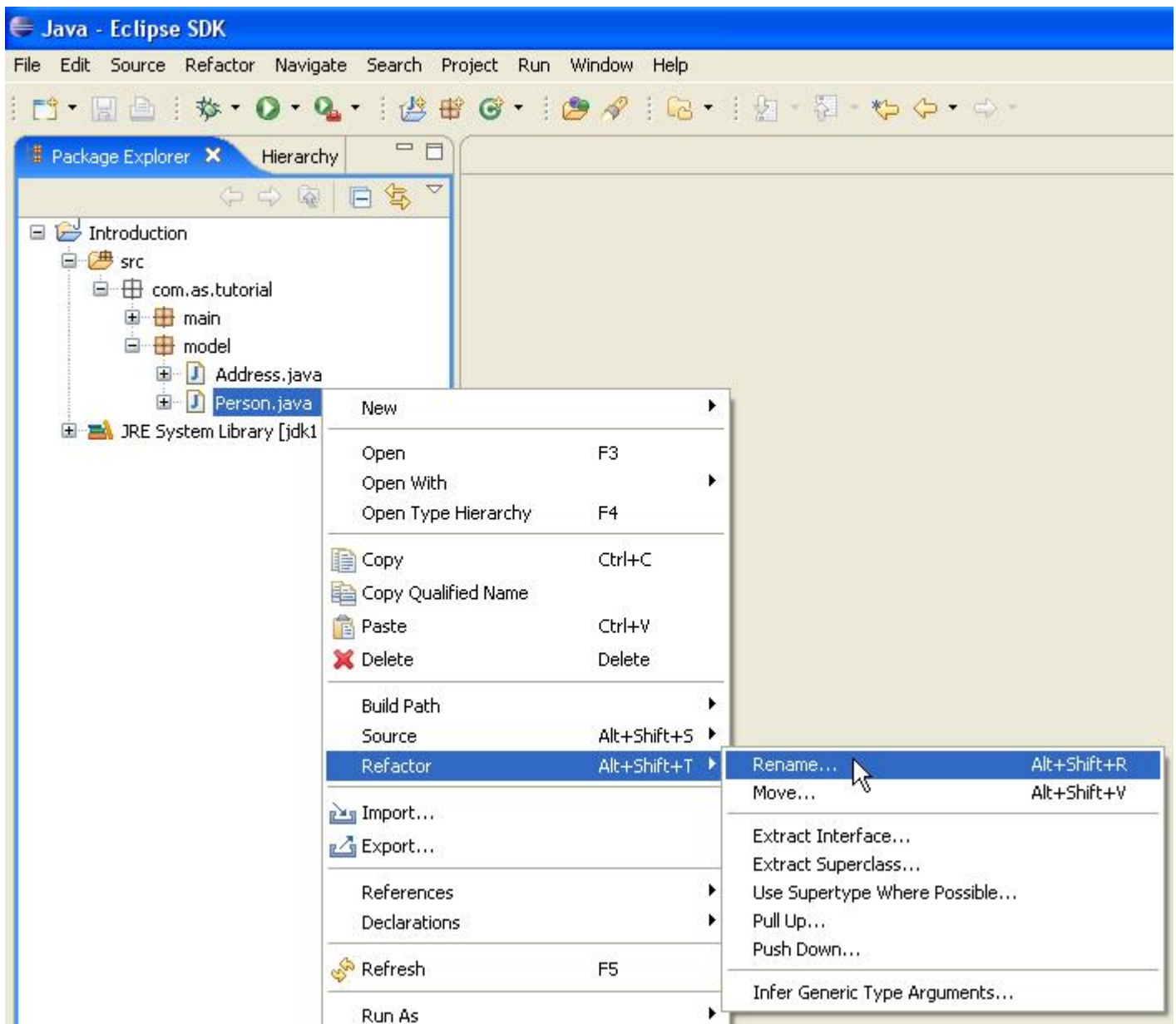
This phenomenon is known as **polymorphism**. Even though the parameter is of type Object, the references know their identity (Person and Address respectively) and Java calls the respective toString() methods.

## Refactoring.

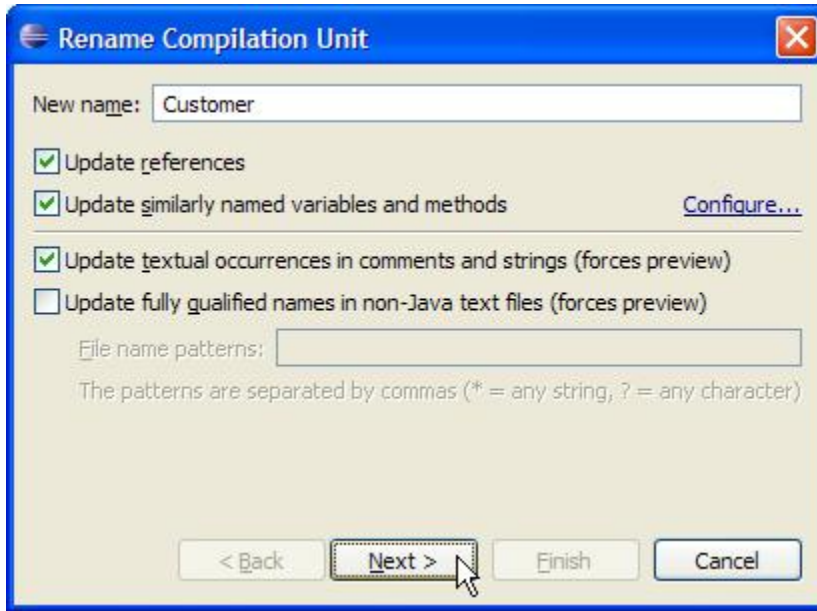
Close any open Java editors within Eclipse. Now that our application is completed, what if you decide that the “Person” class should really be “Customer”? Think of all the references you would have to change.

Eclipse offers a way to make changes, and ripple these changes across an application, with its **refactoring** functions.

In the Eclipse Package Explorer, right-click the Person class and take the option **Refactor, Rename....**



In the “Rename Compilation Unit” dialog box, enter a **New name** of **Customer**. Check the boxes for “Update references”, “Update similarly named variables and methods” and “Update textual occurrences in comments and strings”, then click “**Next**”. Still in the “Rename Compilation Unit” dialog verify that Eclipse’s proposed variable and method renaming changes are correct then click “Next” again. Finally, verify the rest of the changes and click “Finish”.



Eclipse renames the type from Person to Customer, and changes the reference in your code automatically!

```
package com.as.tutorial.main;

import com.as.tutorial.model.Address;
import com.as.tutorial.model.Customer;

public class Application {

    /**
     * Application entry method.
     *
     * @param args
     */
    public static void main(String[] args) {
        Application application = new Application();
        Customer customer = new Customer("David", "Grace", 1970);
        Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
        customer.setAddress(address);
        application.display(customer);
        application.display(customer.getAddress());
    }
}
```

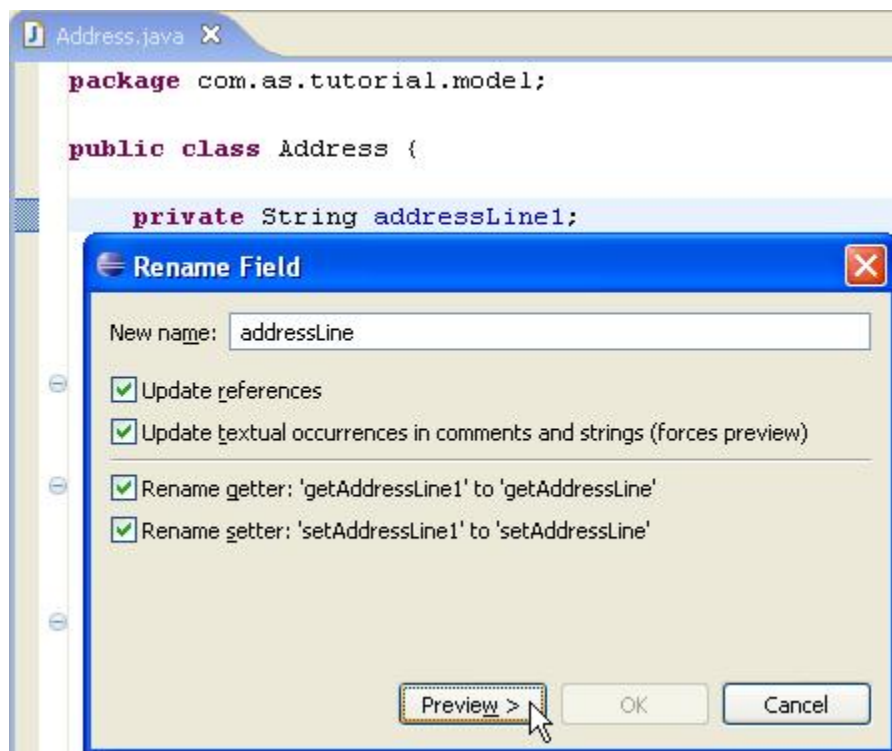
Notice in the code above that the variable name is now **customer**, because we checked the “Update similarly named variables and methods” box.

If you open the “Customer” class in an Eclipse editor, you will also notice that Eclipse changed the word “Person” to “Customer” in the Javadoc, because we checked the “Update textual occurrences in comments and strings” box.

```
/**
 * Creates an instance of Customer.
 */
public Customer() {
    super();
}
```

Use the Eclipse refactoring functionality to perform such tasks as **changing the name** of a package or type (class or interface), or **moving** a type (class or interface) to a completely different package. Eclipse will resolve the references for you.

Within a class, you can even refactor/rename a field, and Eclipse will ripple that new name throughout the class, and its renamed getters and setters throughout the project.



## Debugging.

Let's run our application in the Eclipse debugger. Debugging an application is a good way to determine the cause of bugs (errors) in the code. This exercise assumes that you have performed the previous exercise of renaming the Person class to "Customer" but if you did not just remember that your class name will still be "Person".

First we must set a *breakpoint*, which is a place in the code at which program execution suspends.

Open the Application class in an Eclipse editor. Double-click the vertical ruler down the left side of the editor, at a point which coincides with the first line of code in our Application class main(...) method.



```
/**
 * Application entry method.
 *
 * @param args
 */
public static void main(String[] args) {
    Application application = new Application();
    Customer customer = new Customer("David", "Grace", 1970);
    Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
    customer.setAddress(address);
    application.display(customer);
    application.display(customer.getAddress());
}
```

After you double-click, you will notice a small blue circle, which represents a breakpoint.



```
/**
 * Application entry method.
 *
 * @param args
 */
public static void main(String[] args) {
    Application application = new Application();
    Customer customer = new Customer("David", "Grace", 1970);
    Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
    customer.setAddress(address);
    application.display(customer);
    application.display(customer.getAddress());
}
```

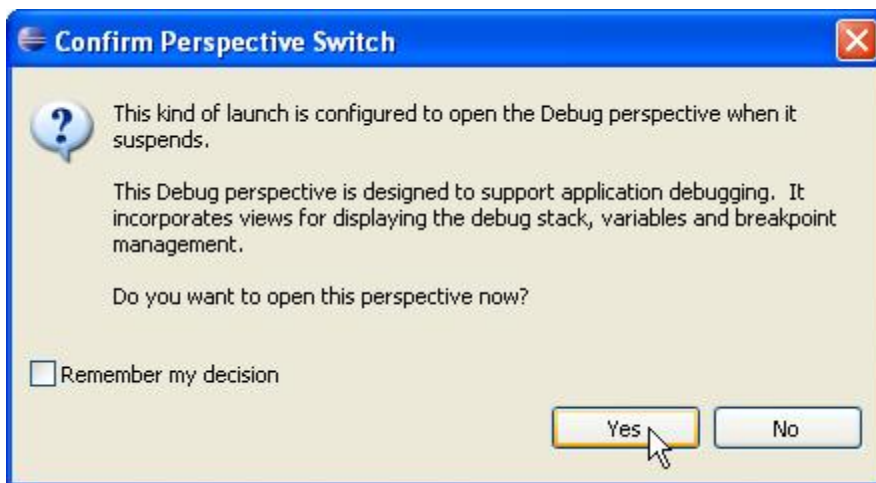


Now take the menu option Run, Debug....

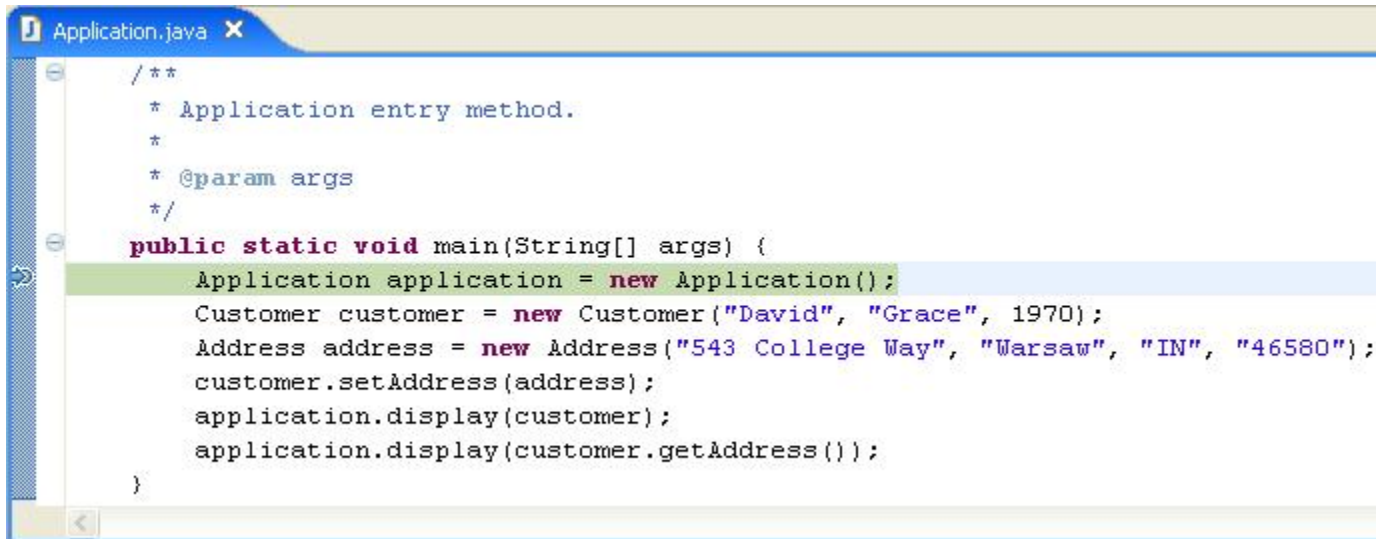


In the "Debug" dialog box, make sure that our "Application" launch configuration is selected, and click the "Debug" button.

In the "Confirm Perspective Switch" dialog, choose "Yes" to switch to the Eclipse *Debug Perspective*.



Notice that in the Debug Perspective, execution suspends at the breakpoint that we previously set.

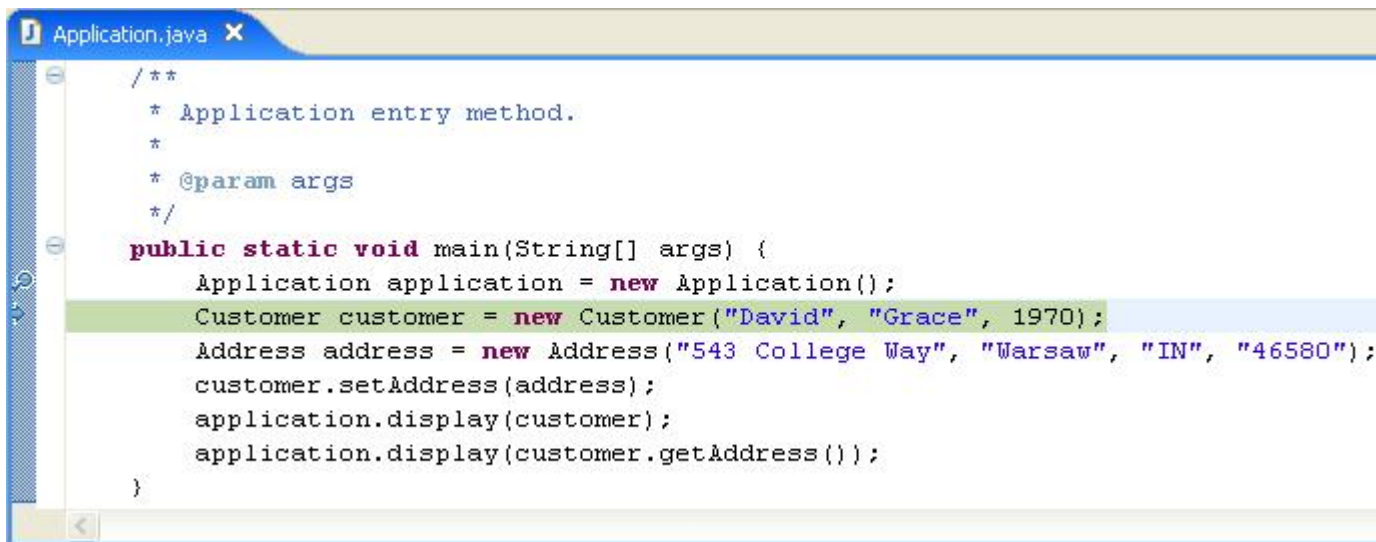


The screenshot shows the Eclipse IDE with a single editor window titled 'Application.java'. The code is as follows:

```
/**
 * Application entry method.
 *
 * @param args
 */
public static void main(String[] args) {
    Application application = new Application();
    Customer customer = new Customer("David", "Grace", 1970);
    Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
    customer.setAddress(address);
    application.display(customer);
    application.display(customer.getAddress());
}
```

A blue horizontal bar highlights the line `Application application = new Application();`. On the left margin, a breakpoint icon (a small circle with a vertical line) is positioned next to this line.

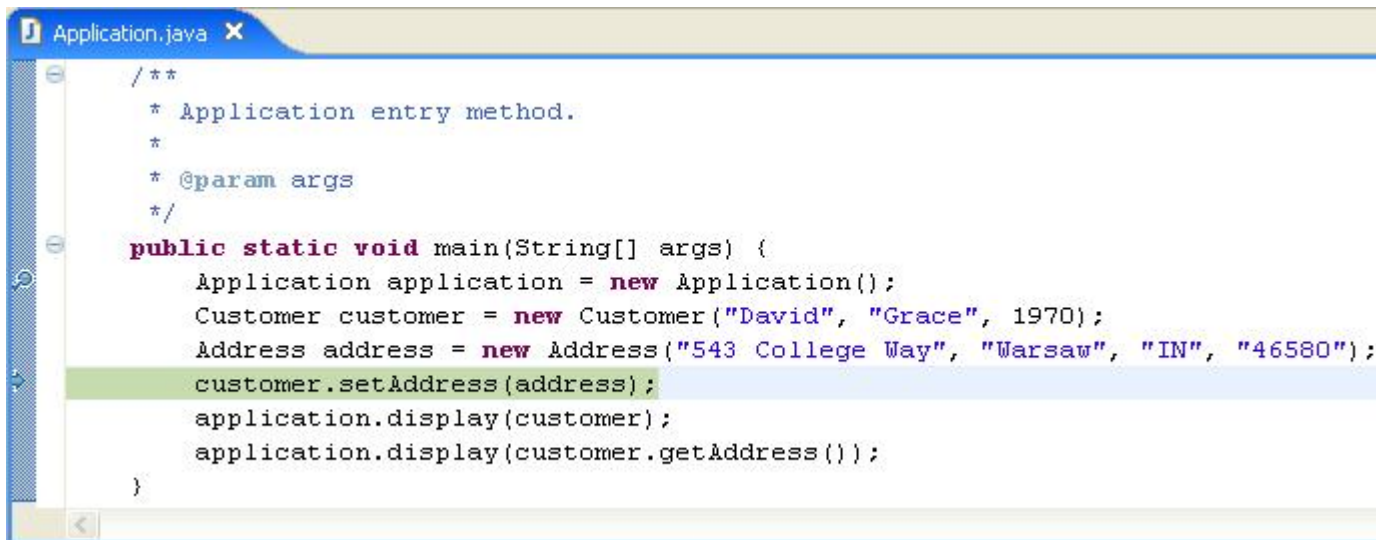
Press **F6** to **Step Over** code, in this case the construction of an instance of the Application class. Notice that now execution suspends at the next line of code.



The screenshot shows the same Eclipse IDE window. The code is identical to the previous one. The blue highlight bar has moved to the next line: `Customer customer = new Customer("David", "Grace", 1970);`. The breakpoint icon on the left margin is now positioned next to this line.

Press F6 until you reach the line

```
customer.setAddress(address);
```



The screenshot shows the Eclipse IDE with a single editor window titled 'Application.java'. The code is as follows:

```
/**
 * Application entry method.
 *
 * @param args
 */
public static void main(String[] args) {
    Application application = new Application();
    Customer customer = new Customer("David", "Grace", 1970);
    Address address = new Address("543 College Way", "Warsaw", "IN", "46580");
    customer.setAddress(address);
    application.display(customer);
    application.display(customer.getAddress());
}
```

The line `customer.setAddress(address);` is highlighted with a green background, indicating it is the current line of execution.

Now press **F5** to **Step Into** the call to `setAddress(...)`. Now the Customer (renamed from Person) class opens in the Debug Perspective, at the next line of execution.



The screenshot shows the Eclipse IDE with two editor windows: 'Application.java' and 'Customer.java'. The 'Customer.java' window is active and shows the following code:

```
return address;
}

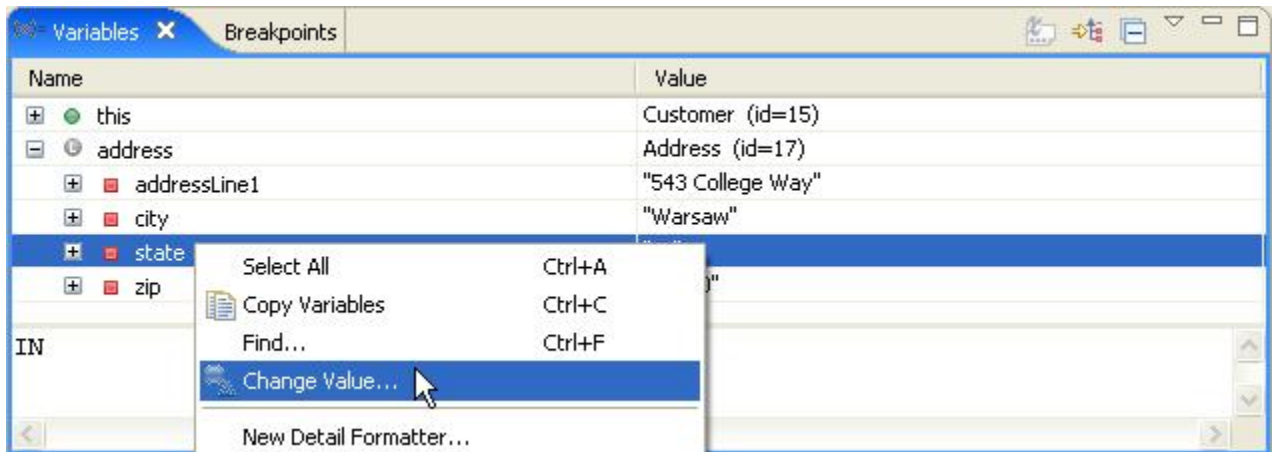
public void setAddress(Address address) {
    this.address = address;
}

public int getBirthYear() {
    return birthYear;
}

public void setBirthYear(int birthYear) {
    this.birthYear = birthYear;
}
```

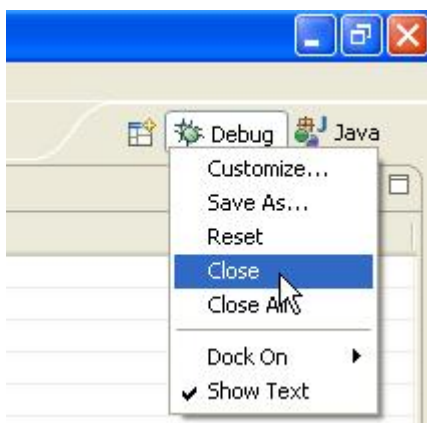
The line `this.address = address;` within the `setAddress` method is highlighted with a green background, indicating it is the current line of execution.

Within the *Variables View* (Window → Show View → Variables) you can see variable values. The context menu for a variable offers a variety of options, including the ability to change a variable value in order to see how subsequent execution handles that value.



Still within the Debug Perspective, press **F8** to **Resume** execution and run to completion (or to the next breakpoint, if you have set another one.)

With the Debug Perspective the active perspective, you can close it by taking the menu option Window, Close Perspective. You can also take the “Close” option in the perspective indicator context menu.



You can learn more about debugging within Eclipse by searching Eclipse Help for the topic *Debugging your programs*.

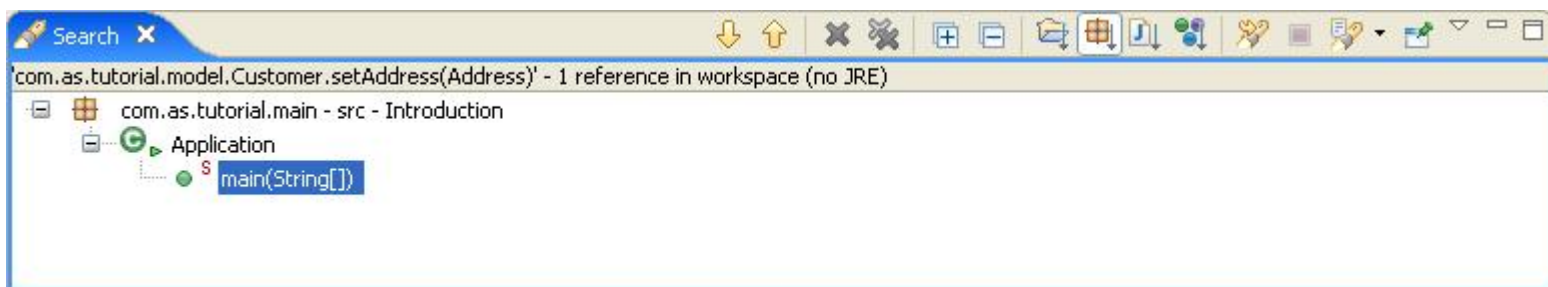
## Searching.

There are various ways to search for information within Eclipse. Here we briefly show you some of our favorites.

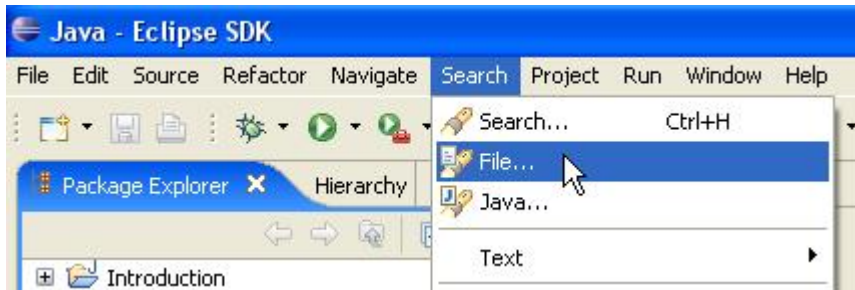
1. It is often useful to find all of the callers of a particular method. In this example we will search for the caller(s) of the Customer class setAddress(Address) method. To find the callers of this method, click once in the method name and press Ctrl+Shift+G (or right-click and in the context menu select *References*, *Workspace*.)



In the “Search” view, Eclipse reports that the Customer setAddress(Address) method is called from the Application class main(String[]) method. You can double-click this search result to open the Application class in an Eclipse editor, with the call to setAddress(Address) highlighted.



2. Another useful technique is searching the Eclipse workspace for occurrences of a particular text string. From the Eclipse menu select *Search, File*.

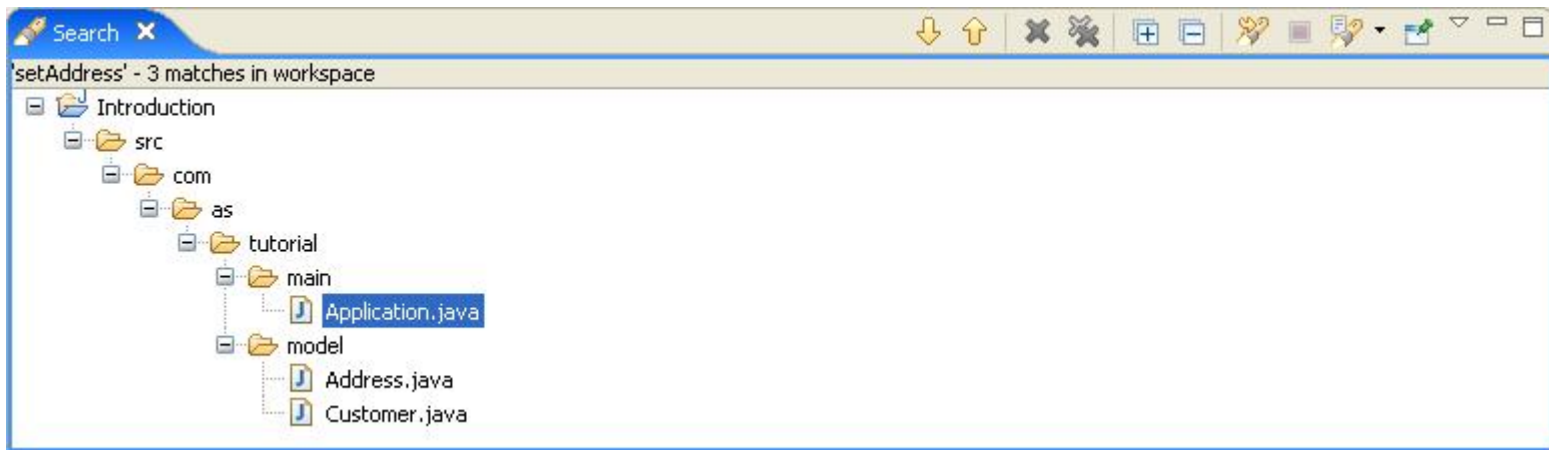


Within the “Search” dialog in this example we have entered the text *setAddress* in the “Containing text” field. Then click the “Search” button.





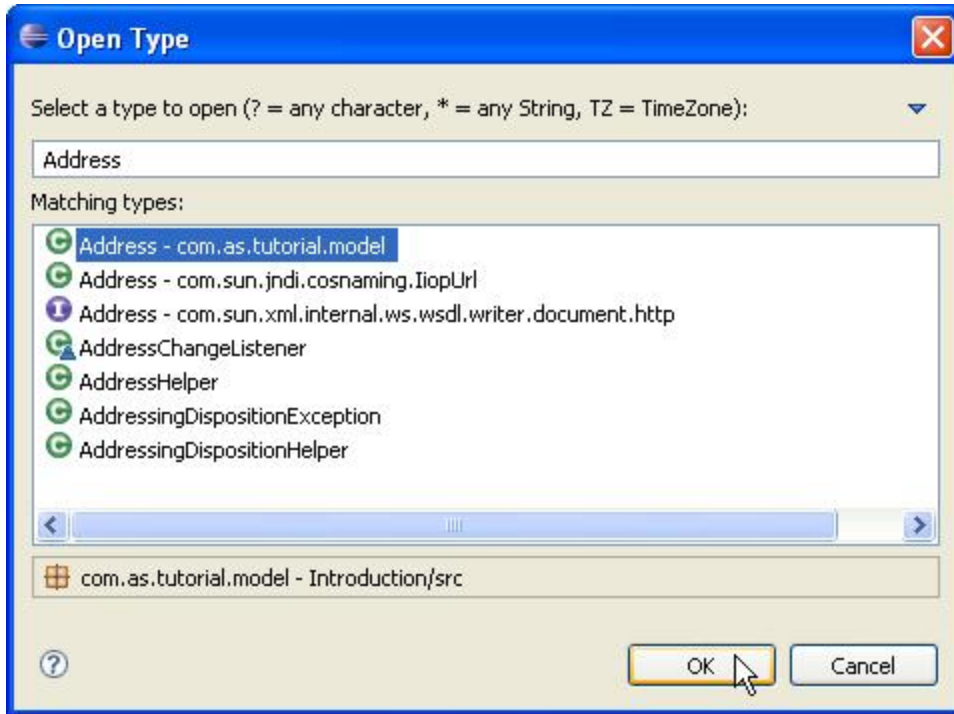
Once again, in the “Search” view, Eclipse displays the resources that contain the text *setAddress*. You can double-click each resource to open it in an Eclipse editor, with the search result(s) indicated.





3. To search for a class or interface, within the Eclipse workspace press Ctrl+Shift+T (or from the Eclipse menu select *Navigate, Open Type...*).

In the “Open Type” dialog start typing the name of the type (without the package.) When you see what you are looking for, select it in the “Matching types” list and click “OK”. Eclipse opens the type in an editor.



## **Conclusion.**

Before you go, we'll leave you with one final tip. Within an Eclipse editor, holding down the Ctrl key and clicking a class or method name, opens a source code editor for that class or method! It's a very handy feature for drilling down through code, and the last tip with which we will leave you in this tutorial.

We hope you have enjoyed this tour of Eclipse. Send us your comments and suggestions, and we look forward to hearing from you. Thank you for using this tutorial.

If you would like to learn more about web application development, please consider purchasing one of our other tutorials:

- Struts 1 with Spring and Hibernate
- Spring MVC with Hibernate
- Struts 2 with Spring and Hibernate

Each tutorial leads you through the development process from start to finish. The download package includes all of the tutorial resources including a style sheet, images, pages, build script, and all of the source code. Better still, these tutorials are nominally priced for the information that they contain.

Thank you again.

<http://www.arctechsoftware.com>