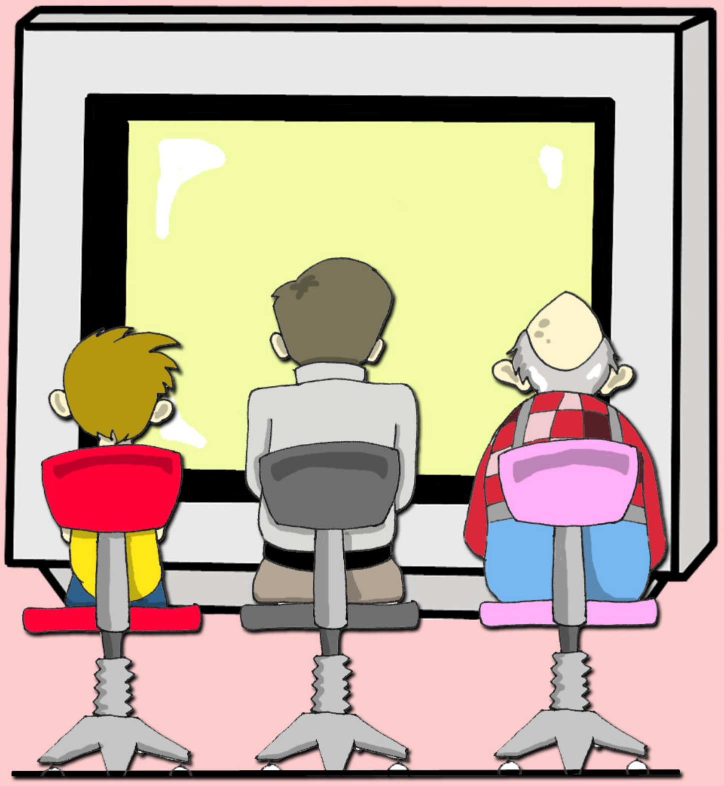
JAVA™ PROGRAMMING
for kids, parents and grandparents



YAKOV FAIN

Java™ Programming for Kids, Parents and GrandParents

Yakov Fain

Java Programming for Kids, Parents and Grandparents

by Yakov Fain

Copyright © 2004 Smart Data Processing, Inc.
14 Molly Pitcher Dr.
Manalapan, New Jersey, 07726, USA

All rights reserved. No part of this book may be reproduced, in any form or by any, without permission in writing from the publisher.

Cover design and illustrations: Yuri Fain

Adult technical editor: Yuri Goncharov

Kid technical editor: David Fain

May 2004: First Electronic Edition

The information in this book is distributed without warranty. Neither the author nor the publisher shall have any liability to any person or entitle to any liability, loss or damage to be caused directly or indirectly by instructions contained in this book or by the computer software or hardware products described herein.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows 98, Windows NT, Windows 2000 and Windows XP are trademarks of Microsoft Corporation. All other product names and company names are the property of their respective owners.

The publisher offers discount on this book when ordered in bulk quantities. For more information, send an e-mail at books@smartdataprocessing.com.

ISBN: 0-9718439-5-3

Table of Contents

Preface

One day my son Davey-steamboat showed up in my office with my rated "R" Java tutorial in his hands. He asked me to teach him programming so he could create computer games. At that time I've already written a couple of books on Java and taught multiple classes about computer programming, but all of this was for grownups! A search on Amazon could not offer anything but books for dummies, but Davey is not a dummy! After spending hours on Google I found either some poor attempts to create Java courses for kids, or some reader-rabbit-style books. Guess what? I decided to write one. To help me understand the mentality of the little people, I decided to ask Davey to become my first kid student.

This book will be useful for the following groups of people

- Kids from 11 to 18 years old
- School computer teachers
- Parents who want to teach their kids programming
- Complete beginners in programming (your age does not matter)

Even though I use a simple language while explaining programming, I promise to treat my readers with respect - I'm not going to write something like "Dear friend! You are about to begin a new and exciting journey…". Yeah, right! Just get to the point!

First chapters of the book will end with simple game-like programs with detailed instructions on how to make them work. Also we are going to create a calculator that looks and works similarly to the one that you have in your computer. In the second part of the book we'll create together game programs Tic-Tac-Toe and Ping-Pong.

You'll need to get used to the slang of professional programmers, and all important words will be printed in *this font.*

Java language elements and programs will be shown in a different font, for example `String`.

This book does not cover each and every element of the Java language, otherwise it would be too fat and boring. But at the end of each chapter there is a section *Additional Reading* wit links to Web sites with more detailed explanations of the subject.

You'll also find assignments at the end of each chapter. Every reader has to complete assignments given in the section *Practice.*

If these assignments are too easy for you, I challenge you to do assignments from the section *Practice for Smarty Pants*. Actually, if you are reading this book, you are a smart person and should try to complete all the assignments.

To get the most out of this book, read it from the beginning to the end. Do not move on until you understand the chapter you are reading now. Teenagers, parents and grandparents should be able to master this book without asking for help, but younger kids should read this book with an adult.

Acknowledgements

Thank you all architects and developers who worked for free on Eclipse – one of the best available Integrated Development Environment for Java.

Special thanks to New Jersey Transit bus drivers for the smooth ride – a half of this book has been written while commuting to work on the bus #139.

Thanks to a lovely lady and my wife Natasha for successfully running a business called family.

Special thanks to Yuri Goncharov - an expert Java programmer from Toronto, Canada. He reviewed the book, tested every code example, and provided a valuable feedback to make this book a little better.

Chapter 1. Your First Java Program

People talk to each other using different languages. Similarly, they write computer programs like games, calculators, text editors using different programming languages. Without programs, your computer would be useless, and its screen would be always black. Computer parts are called *hardware*, and programs are known as *software*. The most popular computer languages are Visual Basic, C++, and Java. What makes the Java language different from many others?

First of all, the same Java program can *run* (work) on different computers like PC, Apple and others without changes. As a matter of fact, Java programs do not even know where they run, because they run inside of a special software shell called Java Virtual Machine (JVM). If, for example, your Java program needs to print some messages, it asks JVM to do this, and JVM know how to deal with your printer.

Second, Java makes it easy to translate your programs (screens, menus and messages) to different human languages.

Third, Java allows you to create program elements (*classes*) that represent objects from the real world. For example, you can create a Java *class* called `Car` and set attributes of this class like doors, wheels, similarly to what the real cars have. After that, based on this class you can create another class, for example `Ford`, which will have all the features of the class `Car` plus something that only Fords have.

Fourth, Java is more powerful than many other languages.

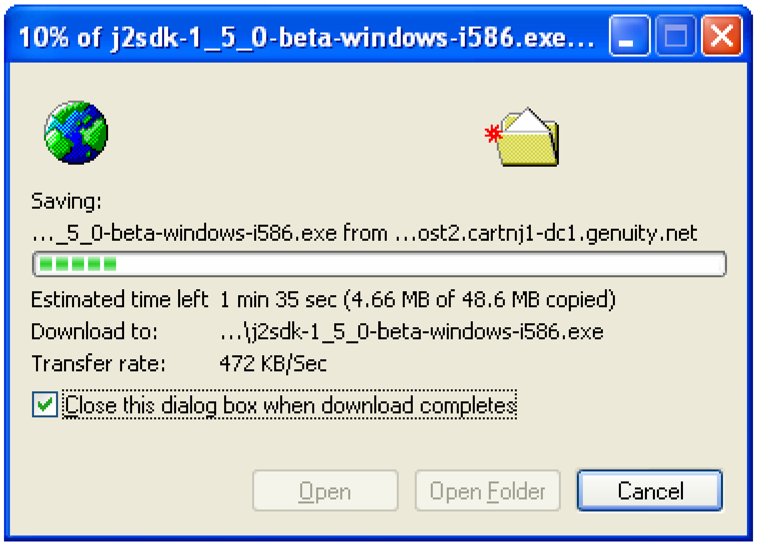
Fifth, Java is free! You can find everything for creating your Java programs on the Internet without paying a penny!

How to Install Java on Your Computer

To start programming in Java you need to download a special software from the Web site of the company called Sun Microsystems, that created this language. The full name of this software is Java 2 Software Development Kit (J2SDK). At the time of this writing its latest version 1.5.0 could be downloaded from this Web site:

http://java.sun.com/j2se

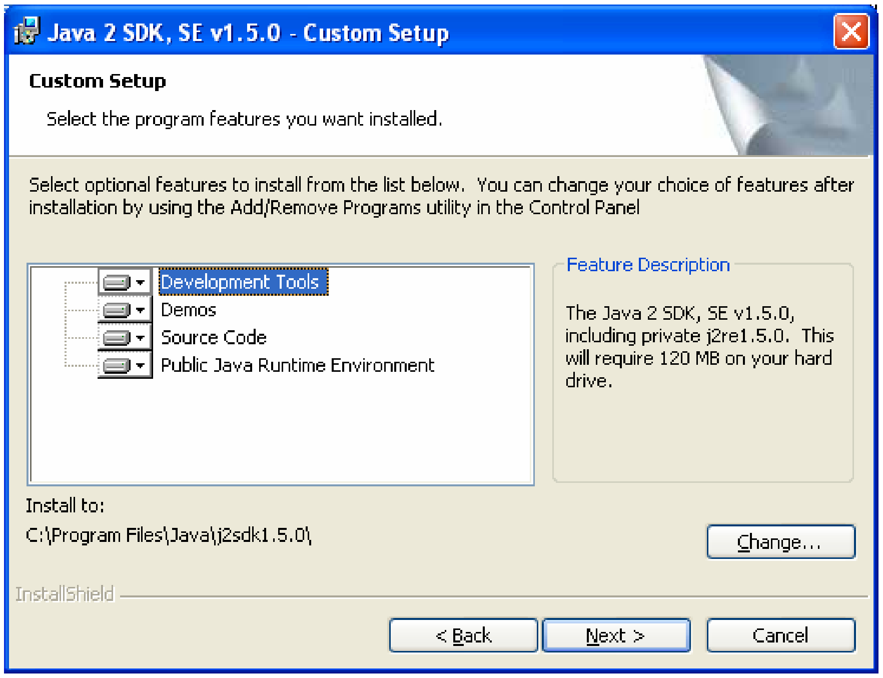
Select *release* J2SE 1.5.0 or the newer one, and on the next Web page under the title Downloads click on the *link* to this release. Then click on the word Download under the title SDK. Accept the license agreement and select *Windows Offline Installation* (unless you have a Mac, Linux or Solaris computer). Press the button *Save* on the next screen and select the folder on your hard disk where you'd like to save the Java installation file. The file download will start.


10% of j2sdk-1_5_0-beta-windows-i586.exe...

Saving:
..._5_0-beta-windows-i586.exe from ...ost2.cartnj1-dc1.genuity.net

Estimated time left 1 min 35 sec (4.66 MB of 48.6 MB copied)
Download to: ...\j2sdk-1_5_0-beta-windows-i586.exe
Transfer rate: 472 KB/Sec
[✓] Close this dialog box when download completes

After the download ends, start the installation process – just double-click on the file that you've downloaded, and this will install J2SDK on your disk. For example, on Windows computer it will create a folder like this one:
`c:\Program Files\java\j2sdk1.5.0`, where `c:` is the name of your hard disk.


Java 2 SDK, SE v1.5.0 - Custom Setup

Custom Setup

Select the program features you want installed.

Select optional features to install from the list below. You can change your choice of features after installation by using the Add/Remove Programs utility in the Control Panel

Development Tools
Demos
Source Code
Public Java Runtime Environment

Feature Description

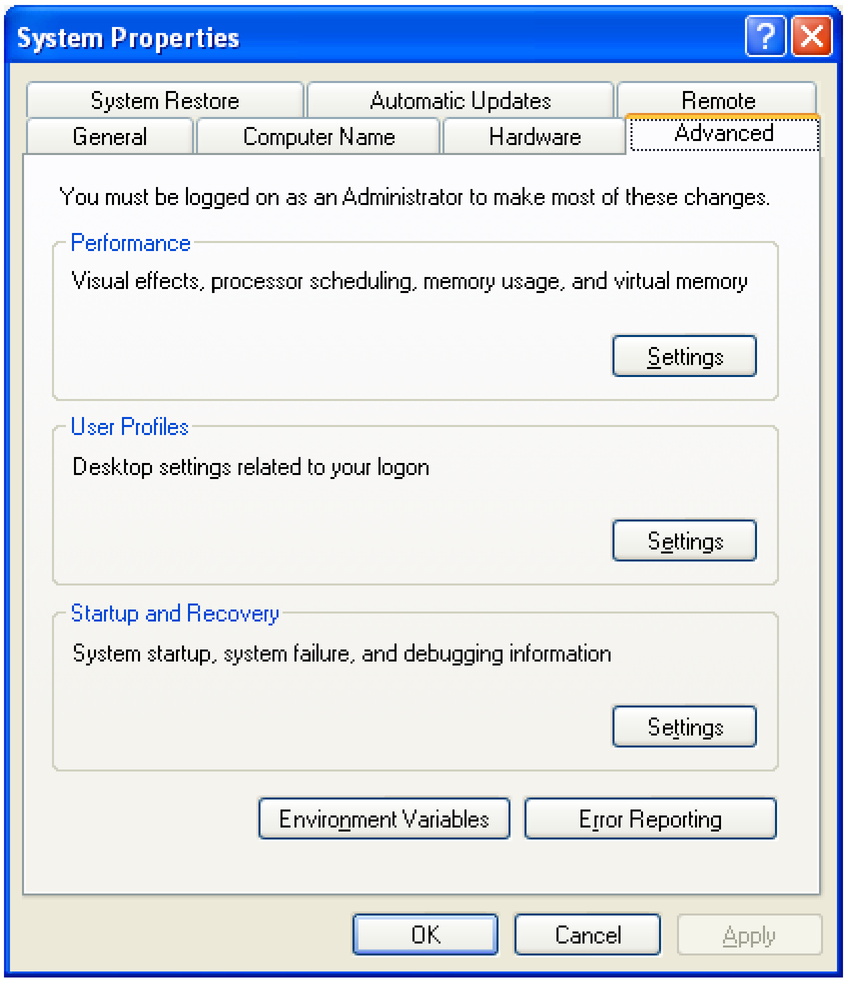
The Java 2 SDK, SE v1.5.0, including private j2re1.5.0. This will require 120 MB on your hard drive.

Install to:
C:\Program Files\Java\j2sdk1.5.0\

If you do not have enough room on your `c:` drive, select a different one, otherwise, just keep pressing the buttons *Next*, *Install* and *Finish* on the windows that will be popping up on your screen. In several minutes the installation of Java on your computer will be complete.

In the next step of installation, you need to define two *system variables*. For example, in Windows click on the button *Start*, and get to the *Control Panel* (it might be hidden behind the menu *Settings*), and click on the icon *System*. Select there a tab *Advanced*, and click on the button *Environment Variables*.

On the next page you can see how this screen looks like on my Windows XP notebook.



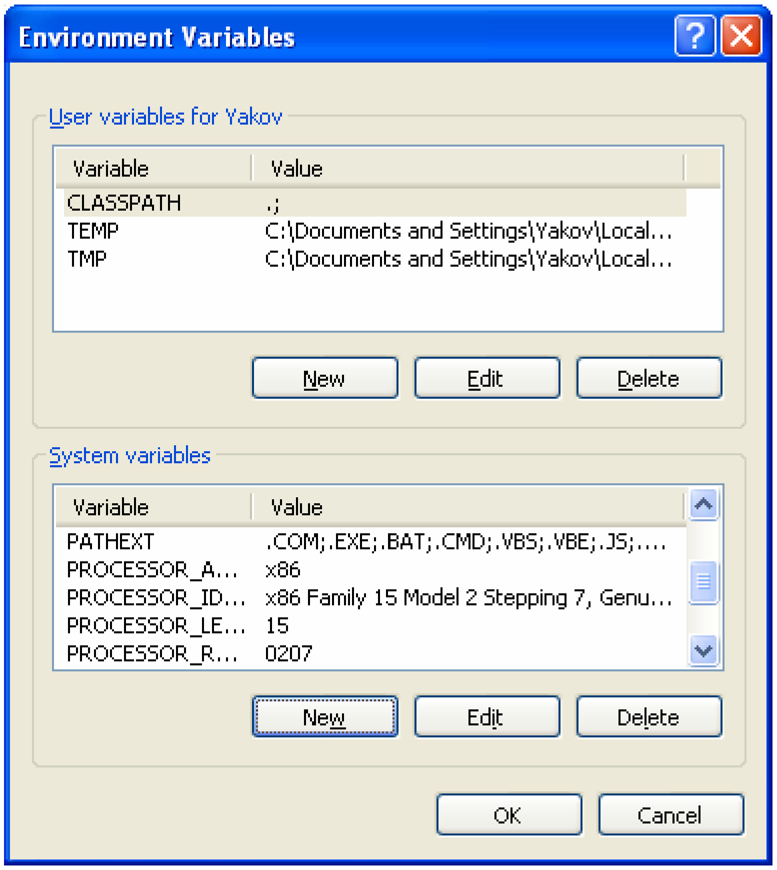
You must be logged on as an Administrator to make most of these changes.

Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your logon

System startup, system failure, and debugging information

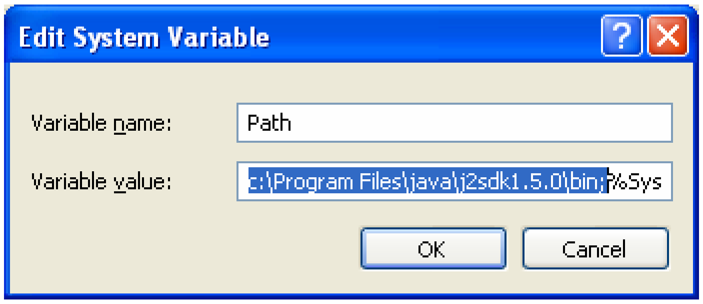
The next window will show all system variables that already exist in your system.



Variable	Value
CLASSPATH	.;
TEMP	C:\Documents and Settings\Yakov\Local…
TMP	C:\Documents and Settings\Yakov\Local…

Variable	Value
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;….
PROCESSOR_A…	x86
PROCESSOR_ID…	x86 Family 15 Model 2 Stepping 7, Genu…
PROCESSOR_LE…	15
PROCESSOR_R…	0207

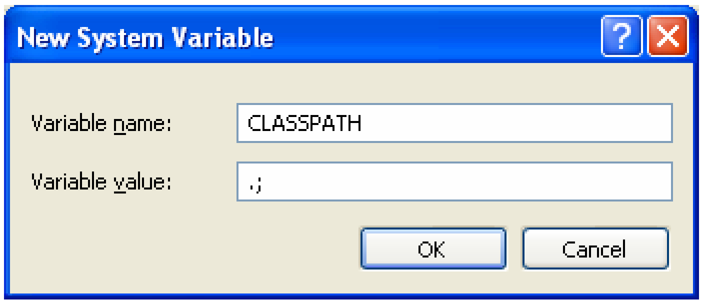
Press the lower button *New* and declare the variable `Path` that will help Windows (or Unix) find J2SDK on your machine. Double check the name of the folder where you've installed Java. If the variable `Path` already exists, just add the new Java directory and a semicolon to the very beginning of the box *Variable Value*:



Variable name: Path

Variable value: c:\Program Files\java\j2sdk1.5.0\bin;%Sys

Also, declare the variable CLASSPATH by entering a period and a semicolon as its value. This system variable will help Java find your programs. The period means that Java has to start looking for your programs from the current disk folder, and the semicolon is just a separator:



Variable name: CLASSPATH

Variable value: .;

Now the installation of J2SDK is complete!

If you have an old Windows 98 computer, you'll need to set the `PATH` and `CLASSPATH` variable in a different way. Find the file autoexec.bat on your `c:` drive, and using Notepad or other text editor enter the proper values for these variable at end of this file, for example:

```
SET CLASSPATH=.;
SET PATH=c:\j2sdk1.5.0\bin;%PATH%
```

After making this change you'll need to restart your computer.

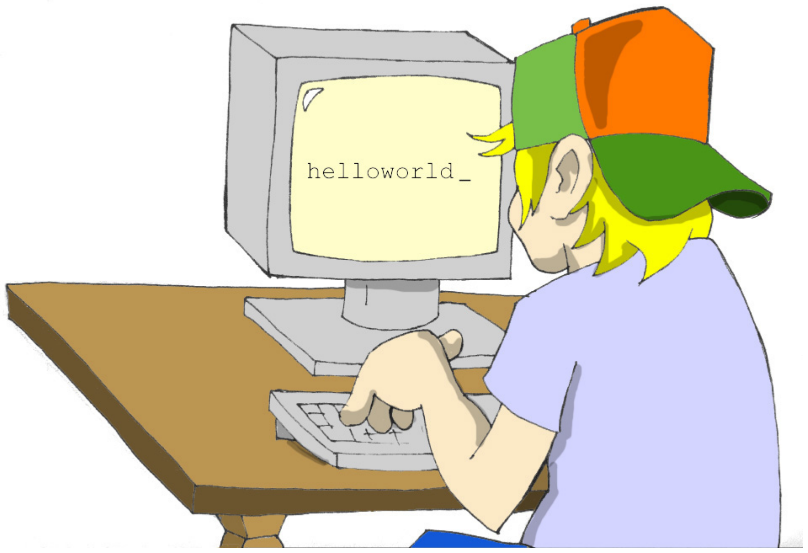
Three Main Steps in Programming

To create a working Java program you need to go through the following tree steps:

- ✓ *Write* the program in Java and save it on a disk.
- ✓ *Compile* the program to translate it from Java language into a special *byte code* that JVM understands.
- ✓ *Run* the program.

Step 1 – Type the Program

You can use any text editor to write Java programs, for example Notepad.



First, you'll need to type the program and save it in a text file with a name ending in *.java*. For example, if you want to write a program called `HelloWorld`, enter its text (we call it *source code*) in Notepad and save it in the file named `HelloWorld.java`. Please do not use blanks in Java file names.

Here is the program that prints on the screen the words *Hello World*:

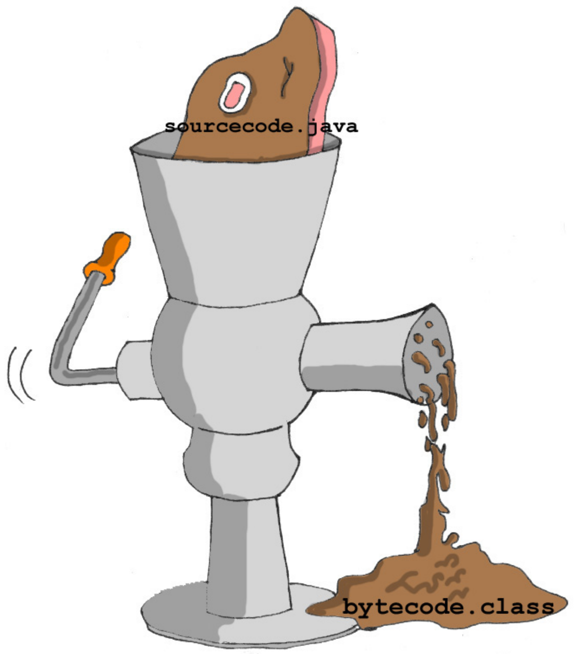
```
public class HelloWorld {

      public static void main(String[] args) {
            System.out.println("Hello World");

      }
}
```

I'll explain how this program works a little later in this chapter, but at this point just trust me – this program will print the words *Hello World* in the step 3.

Step 2 – Compile the Program

Now you need to compile this program. You'll be using the `javac` *compiler*, which is a part of J2SDK.

Let's say you've saved your program in the directory called `c:\practice`. Select the menus *Start*, *Run*, and enter the word `cmd` to open a black command window.



Just to make sure that you've set the system variables `PATH` and `CLASSPATH` correctly, enter the word `set` and take another look at their values.
Change the current folder to `c:\practice` and compile the program:

```
cd \practice

javac HelloWorld.java
```

You do not have to name the folder *practice* – give it any name you like.

In Windows 98 select the "MS DOS Prompt" from the Start menu to open a command prompt window.

The program `javac` is Java *compiler*. You won't see any confirmation that your program `HelloWorld` has been compiled successfully. This is the case when no news is good news. Type a command `dir` and it'll show you all the files that exist in your folder. You should see there a new file named `HelloWorld.class`. This proves that your program has been successfully compiled. Your original file `HelloWorld.java` is also there, and you can modify this file later to print *Hello Mom* or something else.

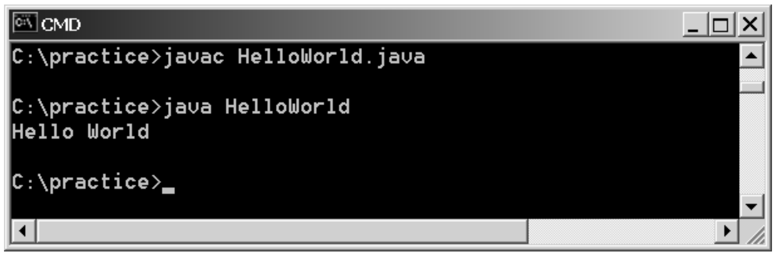
If the program has syntax errors, let's say you forgot to type the last curly brace, Java compiler will print an error message. Now you'd need to fix the error, and recompile the program again. If you have several errors, you may need to repeat these actions more than once until the file `HelloWorld.class` is created.

Step 3 – Run the Program

Now let's run the program. In the same command window enter the following:

```
java HelloWorld
```

Have you noticed that this time you've used the program `java` instead of `javac?` This program is called *Java Run-time Environment (JRE),* or you may call it *JVM* like I did before.



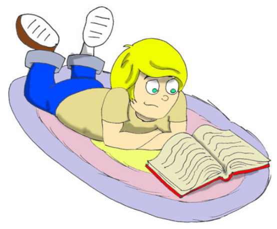
```
C:\practice>javac HelloWorld.java

C:\practice>java HelloWorld
Hello World

C:\practice>_
```

Keep in mind that Java does not treat capital and small letters the same, which means that if you named the program `HelloWorld` with a capital `H` and a capital `W`, do not try to start the program `helloworld` or `helloWorld` – JVM will complain.

Now let's have some fun - try to guess how to change this program. I'll explain how this program works in the next chapter, but still, try to guess how to change it to say hello to you pet, friend or print your address. Go through all three steps to see if the program still works after your changes ☺.

In the next chapter I'll show you how to type, compile and run your programs in a more fancy place than a text editor and a black command window.

Additional Reading



Creating your first application:
http://java.sun.com/docs/books/tutorial/getStarted/cupojava/win32.html

Java installation instructions for Windows:

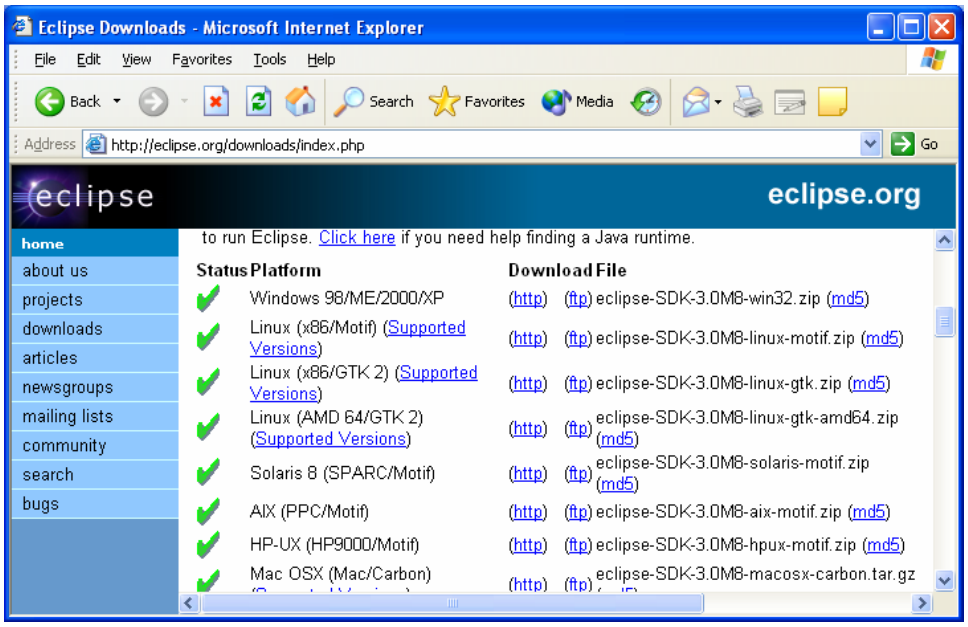
http://java.sun.com/j2se/1.5.0/install-windows.html

Chapter 2. Moving to Eclipse

Programmers usually work in so-called *Integrated Development Environment* (IDE). You can write, compile and run programs there. IDE also has a *Help* thingy that describes all elements of the language, and makes it easier to find and fix errors in your programs. While some IDE programs are expensive, there is an excellent free IDE called Eclipse. You can download it from the Web site www.eclipse.org. In this chapter I'll help you to download and install Eclipse IDE on your computer, create there a project called `Hello World`, and after this we'll be creating all our programs there. Make yourself comfortable in Eclipse – it's an excellent tool that many professional Java programmers use.

Installing Eclipse

Open the Web page www.eclipse.org and click on the *Download* menu on the left (http). Click on the link *Main Eclipse Download Site* and select the version of Eclipse you want to download. They usually have one *latest release* and several *stable builds*. The latest release is an officially released product. Even though stable builds may have more features, they still may have some minor problems. At the time of this writing the latest stable build is 3.0M8. Select this build and you'll see the following window:


Eclipse Downloads - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address Go
eclipse
eclipse.org
home
about us
projects
downloads
articles
newsgroups
mailing lists
community
search
bugs

http://eclipse.org/downloads/index.php

to run Eclipse. Click here if you need help finding a Java runtime.

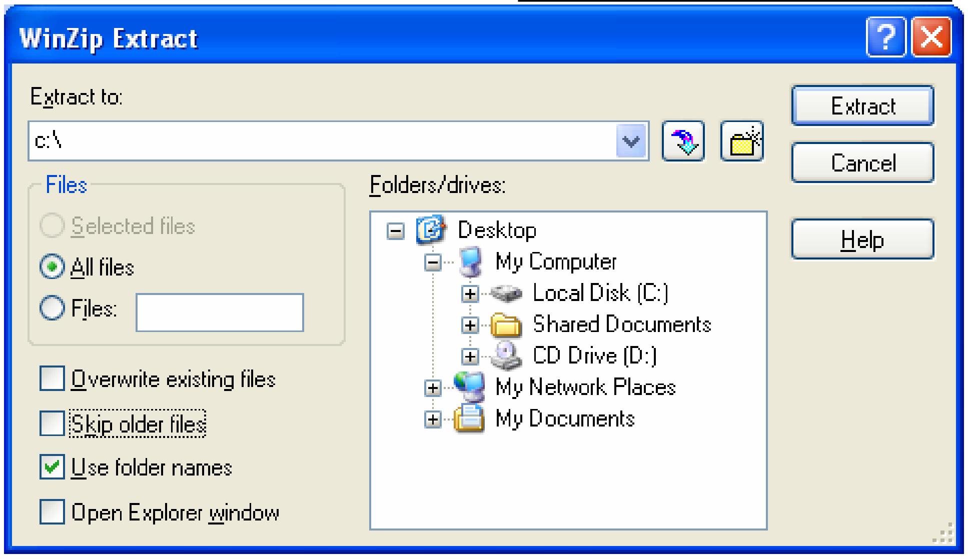
Status	Platform	Download	File
✓	Windows 98/ME/2000/XP	(http) (ftp)	eclipse-SDK-3.0M8-win32.zip (md5)
✓	Linux (x86/Motif) (Supported Versions)	(http) (ftp)	eclipse-SDK-3.0M8-linux-motif.zip (md5)
✓	Linux (x86/GTK 2) (Supported Versions)	(http) (ftp)	eclipse-SDK-3.0M8-linux-gtk.zip (md5)
✓	Linux (AMD 64/GTK 2) (Supported Versions)	(http) (ftp)	eclipse-SDK-3.0M8-linux-gtk-amd64.zip (md5)
✓	Solaris 8 (SPARC/Motif)	(http) (ftp)	eclipse-SDK-3.0M8-solaris-motif.zip (md5)
✓	AIX (PPC/Motif)	(http) (ftp)	eclipse-SDK-3.0M8-aix-motif.zip (md5)
✓	HP-UX (HP9000/Motif)	(http) (ftp)	eclipse-SDK-3.0M8-hpux-motif.zip (md5)
✓	Mac OSX (Mac/Carbon)	(http) (ftp)	eclipse-SDK-3.0M8-macosx-carbon.tar.gz

Click on the link (http) next to the word Windows, Mac, or Linux depending on your computer, and download the file with this long name that ends with *.zip* to any folder on your disk.

Now you just have to unzip this file into your `c:` drive. If you already have the program WinZip installed on your computer, *right-click* on this file and select the WinZip on the menu and the option *Extract To*. If you have room on your c: drive, press the button *Extract*, otherwise select another disk that has more space available.

Files with the name suffix *`.zip`* are archives, and they contain many other files inside. *To unzip the file* means *to extract* the content of this archive on the disk. The most popular archive program is called WinZip and you can download its *trial version* at www.winzip.com.

You'll need it to complete installation of Eclipse.

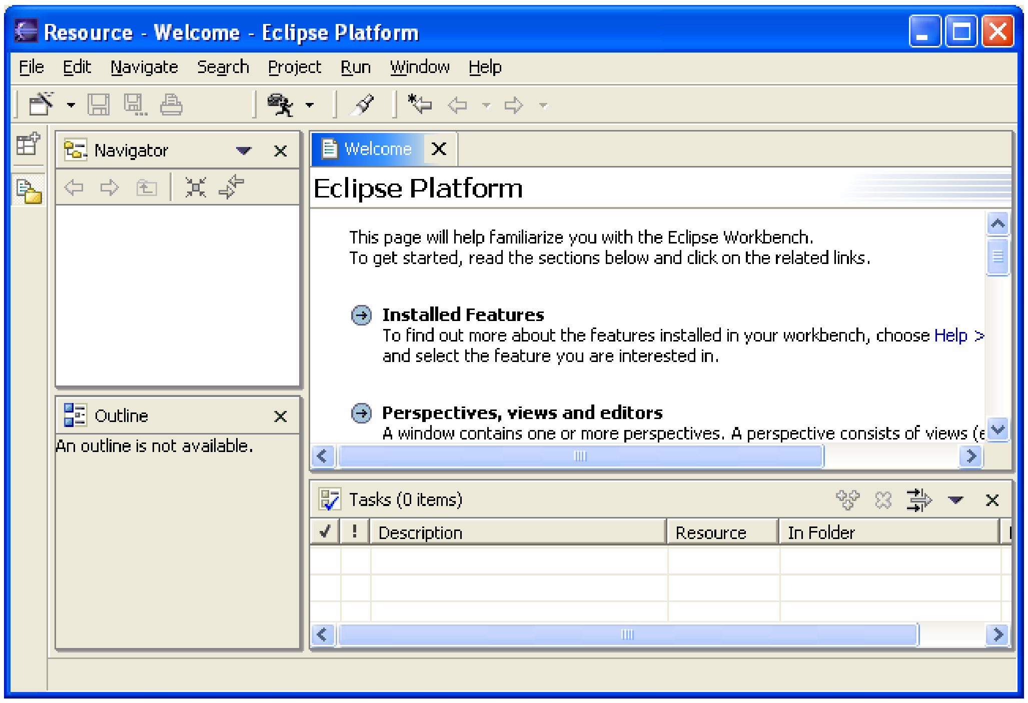


Extract to: c:\

Files: Selected files [✓ All files] Files:

Overwrite existing files [ ]
Skip older files [ ]
Use folder names [✓]
Open Explorer window [ ]

Installation of Eclipse is complete! For your convenience, create the *shortcut* for Eclipse. Right-click on the desktop of your computer, then press *New*, *Shortcut*, *Browse*, and select the file `eclipse.exe` in the folder `c:\eclipse`. To start the program, *double-click* on the blue icon *Eclipse*, and you'll see the first Welcome screen (this screen is changing sligtly with each Eclipse build):


Resource - Welcome - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Navigator
Outline
Welcome
Tasks (0 items)
✓ ! Description Resource In Folder

Eclipse Platform

This page will help familiarize you with the Eclipse Workbench.
To get started, read the sections below and click on the related links.

Installed Features
To find out more about the features installed in your workbench, choose Help >
and select the feature you are interested in.

Perspectives, views and editors
A window contains one or more perspectives. A perspective consists of views (e

An outline is not available.

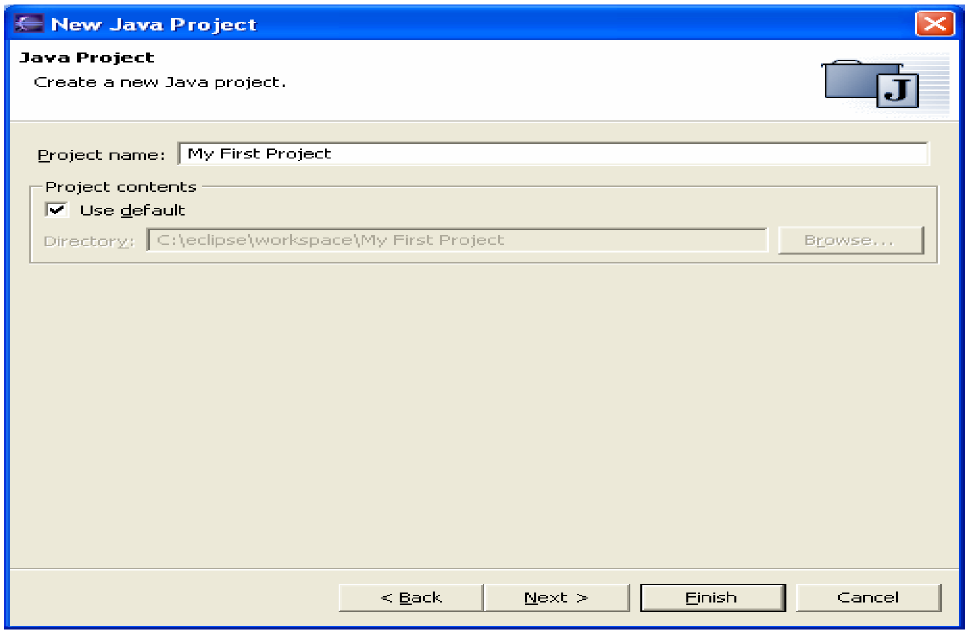
If your screen looks different, proceed to so-called *Workbench*, which is the working area for your Java projects.

Getting Started with Eclipse

In this section I'll show you how you can quickly create and run Java programs in Eclipse. You can also find a nice tutorial under the menus *Help*, *Help Contents*, and *Java Development User Guide*.

To start working on a program you'll need to create a new project. A simple project like our `HelloWorld` will have just one file – `HelloWorld.java`. Pretty soon we'll create more advanced projects that will consist of several files.

To create a brand new project in Eclipse just click on the menus *File*, *New*, *Project*, and then press the button *Next* on the *New Project Window*. Now you'll need to enter the name of your new project, for example *My First Project*:



New Java Project

Java Project

Create a new Java project.

Project name: My First Project

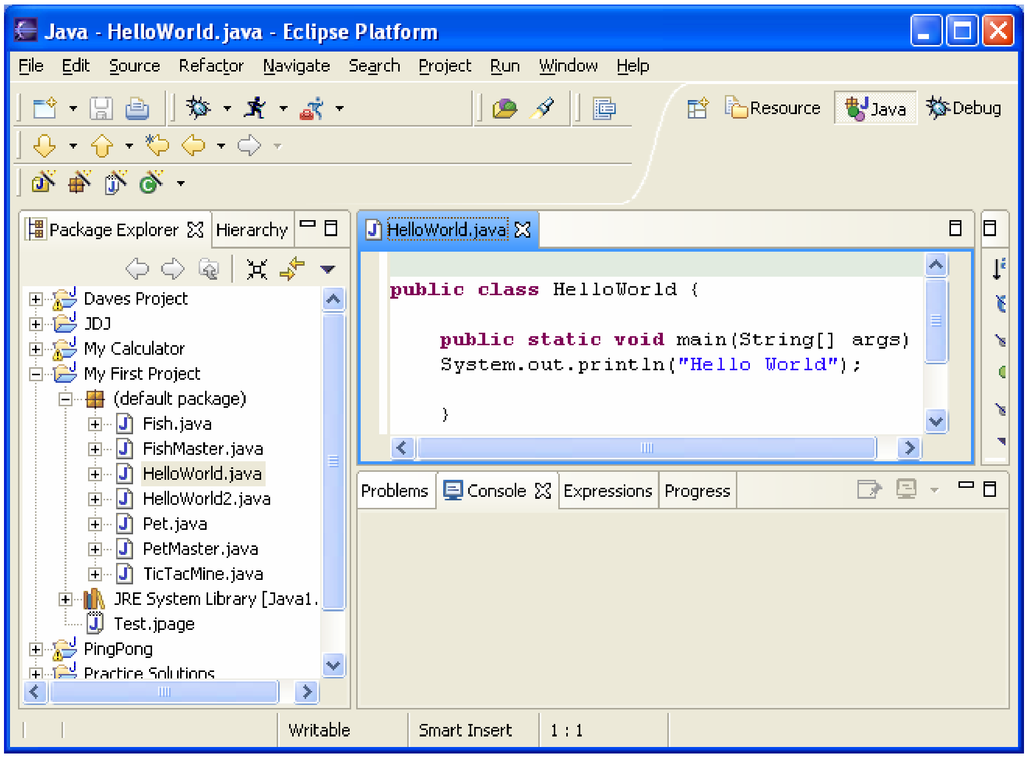
Project contents

[✓] Use default

Directory: C:\eclipse\workspace\My First Project

Look at the grayed out box *Directory*. It tells you where the files of this project will be located on the disk. Eclipse has a special folder `workspace`, where it keeps all files for your projects. Later on, you'll create separate projects for a calculator program, a Tic-Tac-Toe game, and other programs. There will be several projects in the workspace folder by the end of this book.

Eclipse workbench has several smaller areas called *perspectives* which are different views of your projects.



Java - HelloWorld.java - Eclipse Platform

```
public class HelloWorld {

    public static void main(String[] args)
    System.out.println("Hello World");

    }
```

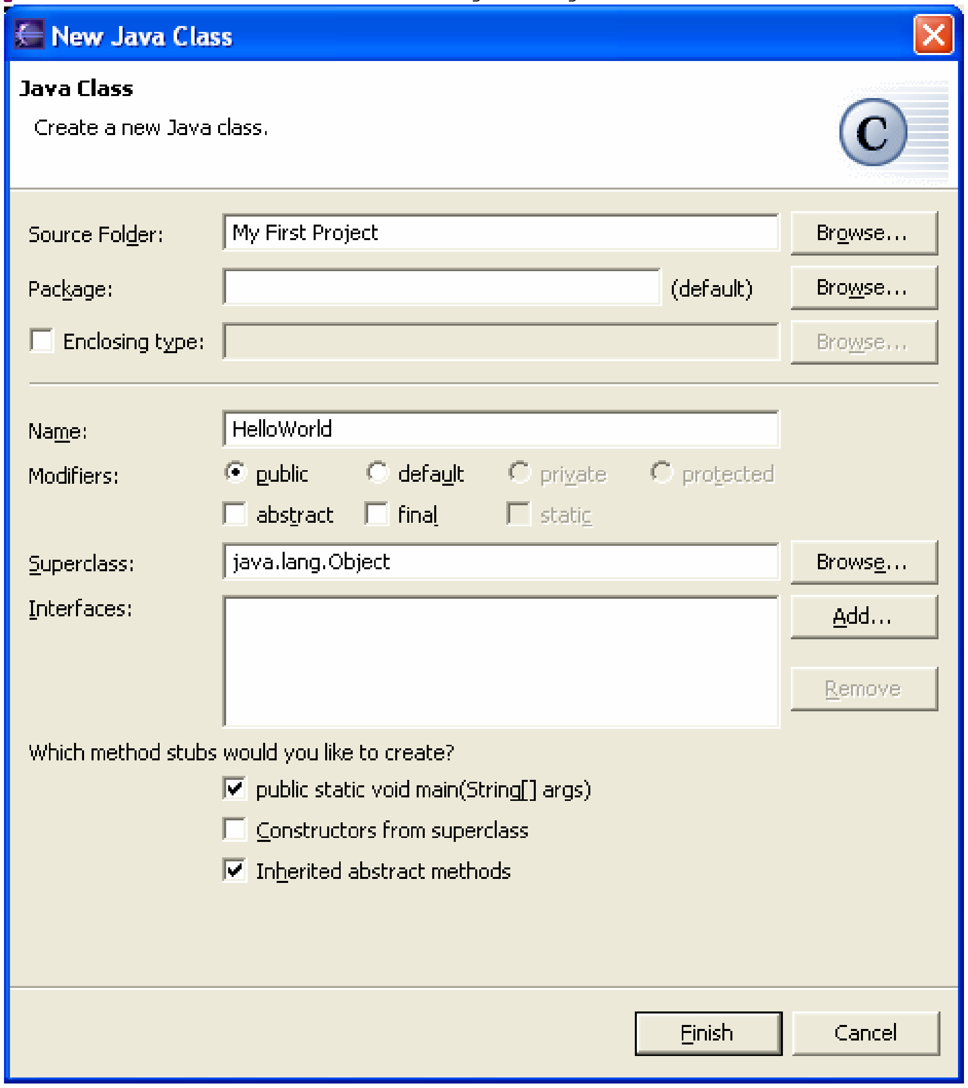
If you click on the little plus sign by *My First Project*, it'll expand showing you an item `Java Run-time Environment (JRE) System Library` which is a part of the project If for any reason you do not see JRE there, click on the menus *Windows, Preferences, Java, Editor, Installed JREs, Add*, and, using the button *Browse* find the folder where you have installed Java, for example c:\j2sdk1.5.0.

Creating Programs in Eclipse

Let's recreate the `HelloWorld` program from Chapter 1 in Eclipse. Java programs are *classes* that represent objects from real life. You'll learn more about classes in the next chapter.

To create a class in Eclipse select the menus *File*, *New*, *Class* and enter `HelloWorld` in the field *Name*. Also, in the section *Which methods stubs you would like to create*, check off the box

```
public static void main(String[] args)
```



Source Folder: My First Project

Package: (default)

☐ Enclosing type:

Name: HelloWorld

Modifiers: [✓ public] default private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Press the button *Finish*, and you'll see that Eclipse created for you the class HelloWorld. It placed *program comments* (the text between /* and */) on top - you should change them to describe your class. After the comments you'll find the code of the class HelloWorld with an empty *method* main(). The word *method* means *action*. To run a Java class as a program, this class must have a method called main().

```
public class HelloWorld {

      public static void main(String[] args) {
      }
}
```

To complete our program, place the cursor after the curly brace in the line with main, push the button *Enter* and type the following on the new line:

```
System.out.println("Hello World");
```

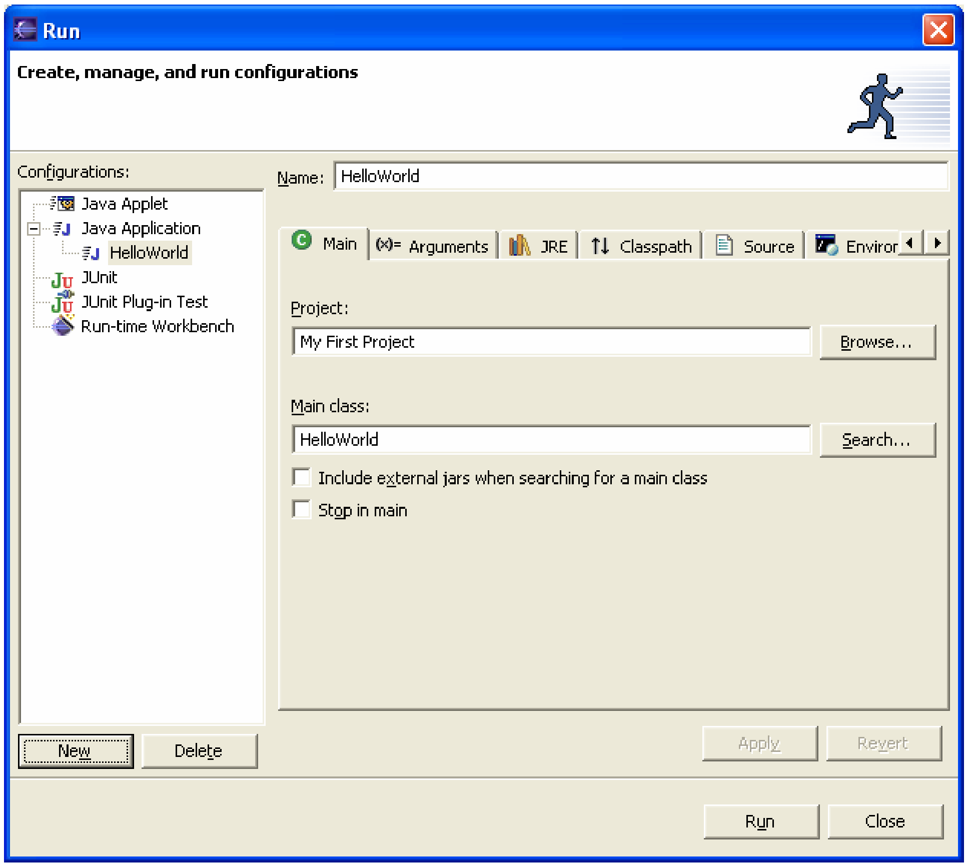
To save the program on disk and compile it, just press at the same time two buttons on your keyboard: *Ctrl-S*. If you did not make any syntax errors, you won't see any messages – the program is compiled. But let's make an error on purpose to see what's going to happen. Erase the last curly brace and hit *Ctrl-S* again. Eclipse will display the Unmatched Brace error in the tasks perspective, and also it will place a red mark at the line that has a problem.

As your projects become larger, they'll have several files and compiler may generate more than one error. You can quickly find (not fix though) the problematic lines by double-clicking on the error message in the *tasks perspective*. Let's put the curly brace back and hit *Ctrl-S* again – voila, the error message is gone!

Running HelloWorld in Eclipse

Our simple program is a one-class project. But pretty soon you projects will have several Java classes. That's why before running the project for the first time, you need to tell Eclipse which class in this project is the main one.

Select the menu *Run*, then *Run…*(make sure that *Java Application* is selected in the top left corner), and enter the names of the project and the main class:



Name: HelloWorld

Project: My First Project

Main class: HelloWorld

☐ Include external jars when searching for a main class

☐ Stop in main

Now press the button *Run*, to start the the program. It will print the words *Hello World* in the *console view* the same way as it did in Chapter 1.

Now you can run this project by selecting the menus *Run*, *Run Last Launched* or by pressing the buttons *Ctrl-F11* on the keyboard.

How `HelloWorld` Works?

Let's start learning what's actually happening in the program `HelloWorld`.

The class `HelloWorld` has only one method `main`(), which is an entry point of a Java *application* (program). You can tell that `main` is a method, because it has parentheses after the word `main`. Methods can *call* (use) other methods, for example our method `main()` calls the method `println()` to display the text `Hello World` on the screen.

Each method starts with a *declaration line* called *a method signature*:

```
public static void main(String[] args)
```

This method signature tells us the following:

- Who can access the method - `public`. The keyword `public` means that the method `main()` could be accessed by any other Java class or JVM itself.

- Instructions on how to use it - `static`. The keyword `static` means that you don't have to create an *instance* (a copy) of `HelloWorld` object in memory to use this method. We'll talk about class instances more in the next chapter.

- Does the method *return* any data? The keyword `void` means that the method `main()` doesn't return any data to the calling program, which is Eclipse in this case. But if for example, a method had to perform some calculations, it could have returned a resulting number to its caller.

- The name of the method is `main`.

- The list of arguments – some data that could be given to the method - `String[] args`. In the method `main()` the `String[] args` means that this method can receive an *array* of `Strings` that represent text data. The values that are being passed to a method are called *arguments*.

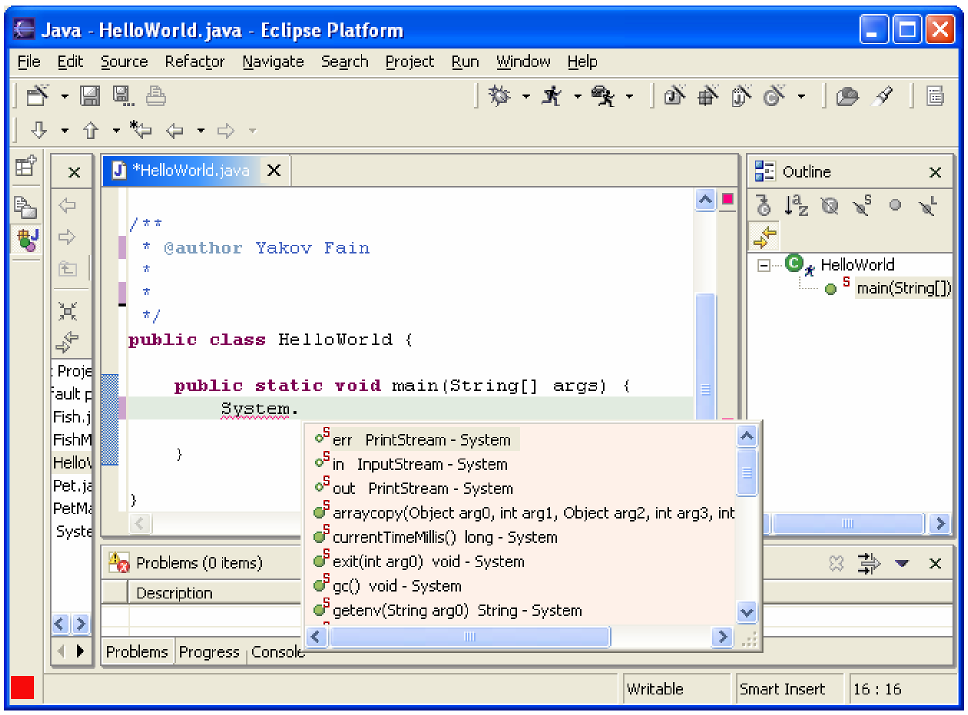
As I said before, you can have a program that consists of several classes, but one of them has the method `main()`. Java class usually have several methods. For example, a class Game can have the methods `startGame()`, `stopGame()`, `readScore()`, and so on.

The body of our method `main()`has only one line :

```
System.out.println("Hello World");
```

Every command or a method call must end with a semicolon `;`. The method `println()`knows how to print data on the system *console* (command window). Java's method names are always followed by parentheses. If you see a method with empty parentheses, this means that this method does not have any arguments.

The `System.out` means that the variable `out` is defined inside the class `System` that comes with Java. How are you supposed to know that there's something called `out` in the class `System`? Eclipse will help you with this. After you type the word `System` and a dot, Eclipse will show you everything that is available in this class. At any time you can also put a cursor after the dot and press *Ctrl-Space* to bring up a help box similar to this one:


Java - HelloWorld.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
*HelloWorld.java
Outline
HelloWorld
main(String[])
Problems (0 items)
Description
Problems Progress Console
Writable
Smart Insert
16 : 16

```
/**
 * @author Yakov Fain
 *
 *
 */
public class HelloWorld {

    public static void main(String[] args) {
        System.

    }
}
```

```
err  PrintStream - System
in  InputStream - System
out  PrintStream - System
arraycopy(Object arg0, int arg1, Object arg2, int arg3, int
currentTimeMillis()  long - System
exit(int arg0)  void - System
gc()  void - System
getenv(String arg0)  String - System
```

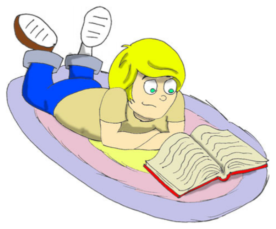
The `out.println()` tells us that there is an object represented by a *variable* `out` and this "something called `out`" has a method called `println()`. The dot between a class and a method name means that this method exists inside this class. Say you have a class `PingPongGame` that has a method `saveScore()`. This is how you can *call* this method for Dave who won three games:

```
PingPongGame.saveScore("Dave", 3);
```

Again, the data between parentheses are called *arguments* or *parameters*. These parameters are given to a method for some kind of processing, for example saving data on the disk. The method `saveScore()` has two arguments –a text string "Dave", and the number 3.

Eclipse will add fun to writing Java programs. Appendix B has some useful tips and tricks that will speed up your Java programming in this excellent IDE.

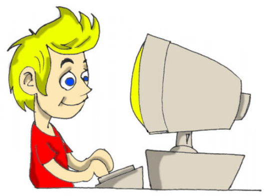
Additional Reading



Eclipse Web Page:

http://www.eclipse.org

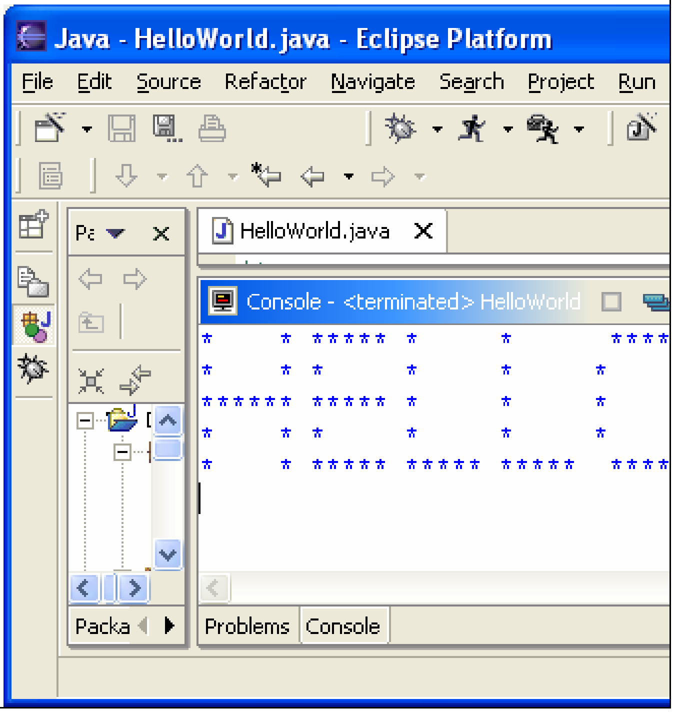
Practice



Change the class `HelloWorld` to print your address using several calls to `println()`.

Practice for Smarty Pants

Change the class `HelloWorld` to print the word *Hello* like this:


Java - HelloWorld.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run
Pa
HelloWorld.java
Console - <terminated> HelloWorld
Packa
Problems Console

```
*    * ***** *     *      ****
*    * *     *     *     *
****** ***** *     *     *
*    * *     *     *     *
*    * ***** ***** *****  ****
```

Chapter 3. Pet and Fish – Java Classes

Java programs consist of *classes* that represent objects from the real world. Even though people may have different preferences as to how to write programs, most of them agree that it's better to do it in a so-called *object-oriented* style. This means that good programmers start with deciding which objects have to be included in the program and which Java classes will represent them. Only after this part is done, they start writing Java code.

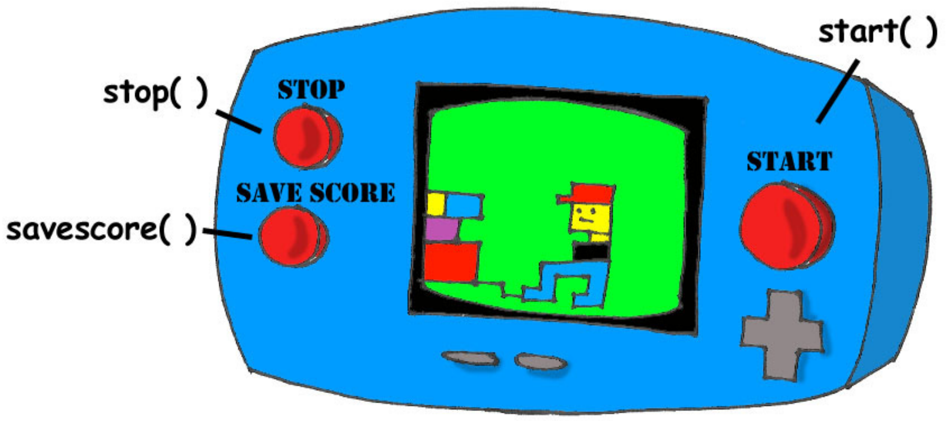
Classes and Objects

Classes in Java may have *methods* and *attributes*.

Methods define actions that a class can perform.

Attributes describe the class.

Let's create and discuss a class named `VideoGame`. This class may have several methods, which can tell *what objects of this class can do*: start the game, stop it, save the score, and so on. This class also may have some attributes or properties: price, screen color, number of remote controls and others.



In Java language this class may look like this:

```
class VideoGame {
  String color;
  int price;

  void start () {
  }
  void stop () {
  }
  void saveScore(String playerName, int score) {
  }
}
```

Our class `VideoGame` should be similar to other classes that represent video games – all of them have screens of different size and color, all of them perform similar actions, and all of them cost money.

We can be more specific and create another Java class called `GameBoyAdvance`. It also belongs to the family of video games, but has some properties that are specific to the model GameBoy Advance, for example a cartridge type.

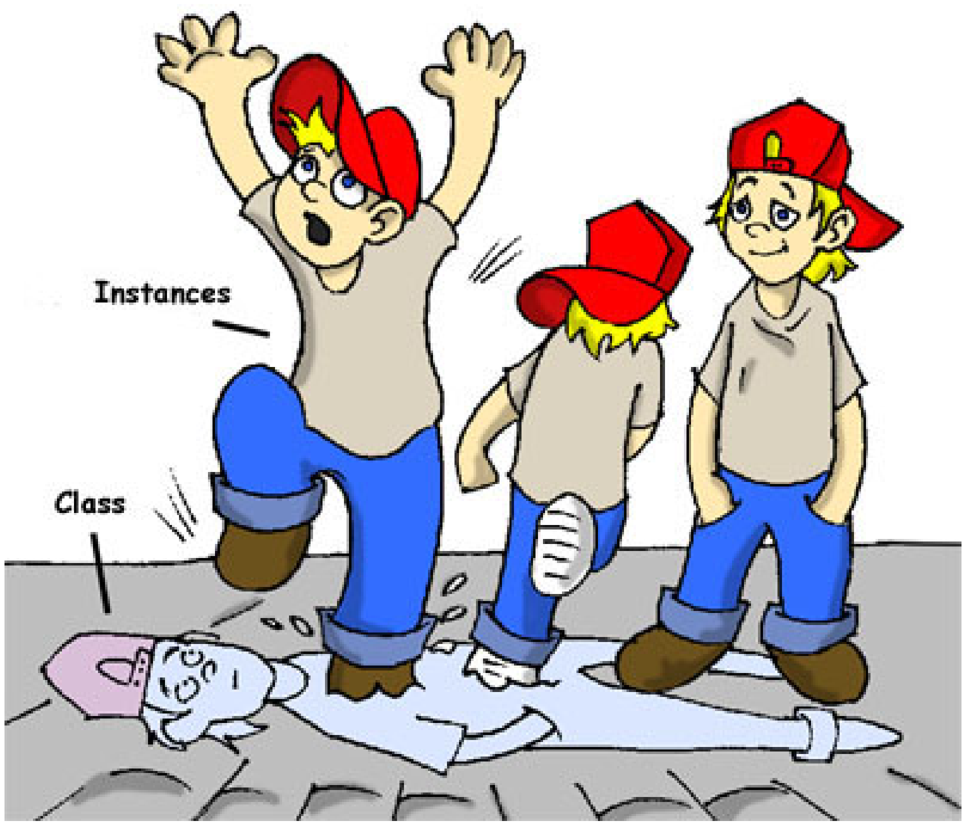
```
class GameBoyAdvance {
  String cartridgeType;
  int screenWidth;

  void startGame() {

  }
  void stopGame() {

  }
}
```

In this example the class `GameBoyAdvance` defines two attributes – `cartridgeType` and `screenWidth` and two methods – `startGame()` and `stopGame()`. But these methods can't perform any actions just yet, because they have no Java code between the curly braces.

In addition to the word *class*, you'll have to get used to the new meaning of the word *object*.

The phrase "to create an instance of an object" means to create a copy of this object in the computer's memory according to the definition of its class.

A factory description of the GameBoy Advance relates to an actual game the same way as a Java class relates to its instance in memory. The process of building actual games based on this description in the game factory is similar to the process of creating instances of `GameBoy` objects in Java.



In many cases, a program can use a Java class only after its instance has been created. Vendors also create thousands of game copies based on the same description. Even though these copies represent the same class, they may have different *values* in their attributes - some of them are blue, while others are silver, and so on. In other words, a program may create *multiple instances* of the `GameBoyAdvance` objects.

Data Types

Java *variables* represent attributes of a class, method arguments or could be used inside the method for a short-time storage of some data. Variables have to be declared first, and only after this is done you can use them.

Remember equations like y=x+2? In Java you'd need to declare the variables x and y of some numeric *data type* like integer or `double`:

```
int x;
int y;
```

The next two lines show how you can assign a *value* to these variables. If your program assigns the value of five to the variable `x`, the variable `y` will be equal to seven:

```
x=5;
y=x+2;
```

In Java you are also allowed to change the value of a variable in a somewhat unusual way. The following two lines change the value of the variable `y` from five to six:

```
int y=5;
y++;
```

Despite the two plus signs, JVM is still going to increment the value of the variable `y by one`.

After the next code fragment the value of the variable `myScore` is also six:

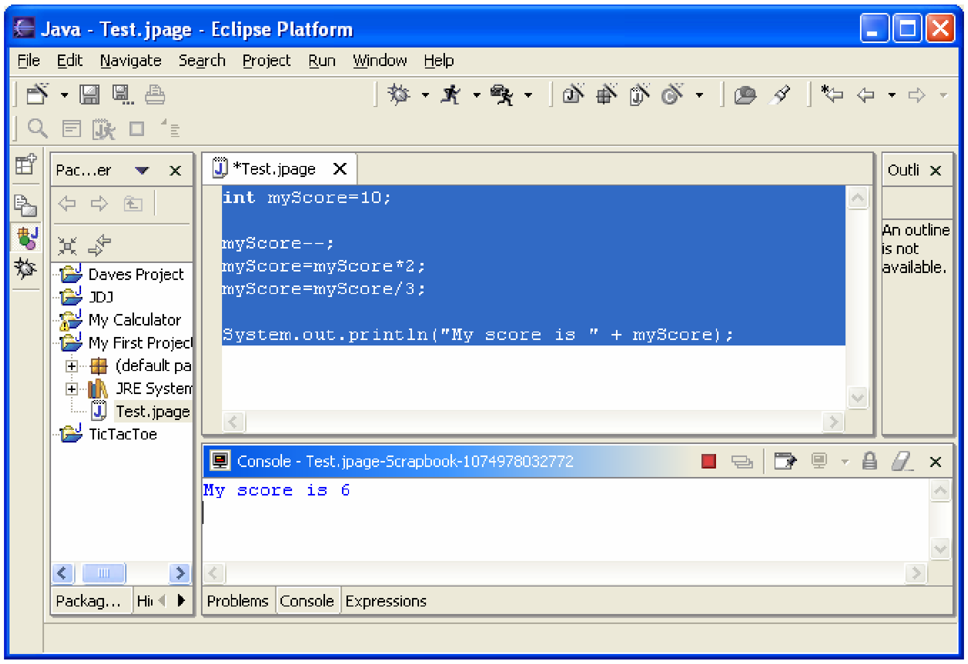
```
int myScore=5;
myScore=myScore+1;
```

You can also use multiplication, division and subtraction the same way. Look at the following piece of code:

```
int myScore=10;

myScore--;
myScore=myScore*2;
myScore=myScore/3;

System.out.println("My score is " + myScore);
```

What this code prints? Eclipse has a cool feature called a scrapbook that allows quickly test any code snippet (like the one above) without even creating a class. Select menus *File*, *New*, *Scrapbook Page* and type the word `Test` as the name of your scrapbook file.

Now enter these five lines that manipulate with `myScore` in the scrap book, highlight them, and click on the little looking glass on the toolbar.


Java - Test.jpage - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Pac...er
*Test.jpage
Outli
An outline is not available.
Daves Project
JDJ
My Calculator
My First Project
(default pa
JRE System
Test.jpage
TicTacToe
Console - Test.jpage-Scrapbook-1074978032772
Packag... Hi
Problems Console Expressions

```
int myScore=10;

myScore--;
myScore=myScore*2;
myScore=myScore/3;

System.out.println("My score is " + myScore);
```

```
My score is 6
```

To see the result of the score calculations, just click on the console tab at the bottom of the screen:

```
My score is 6
```

In this example the argument of the method `println()` was glued from two pieces – the text “My score is ” and the value of the variable `myScore`, which was six. Creation of a `String` from pieces is called *concatenation*. Even though `myScore` is a number, Java is smart enough to convert this variable into a `String`, and then attach it to the text *My Score is*.

Look at some other ways of changing the values of the variables:

`myScore=myScore*2;` is the same as `myScore*=2;`
`myScore=myScore+2;` is the same as `myScore+=2;`
`myScore=myScore-2;` is the same as `myScore-=2;`
`myScore=myScore/2;` is the same as `myScore/=2;`

There are eight simple, or *primitive* data types in Java, and you have to decide which ones to use depending on the type and size of data that you are planning to store in your variables:

✓ Four data types for storing integer values – `byte`, `short`, `int`, and `long`.

✓ Two data types for values with a decimal point – `float` and `double`.

✓ One data type for storing a single character – `char`.

✓ One *logical* data type called `boolean` that allows only two values: `true` or `false`.

You can assign an initial value to a variable during its declaration and this is called *variable initialization*:

```
char grade = 'A';
int chairs = 12;
boolean playSound = false;
double nationalIncome = 23863494965745.78;
float gamePrice = 12.50f;
long totalCars =4637283648392l;
```

In the last two lines `f` means float and `l` means long.

If you don't initialize the variables, Java will do it for you by assigning zero to each numeric variable, `false` to `boolean` variables, and a special code '\u0000' to a `char`.

There is also a special keyword `final`, and if it's used in a variable declaration, you can assign a value to this variable only once, and this value cannot be changed afterwards. In some languages the final variables are called *constants*. In Java we usually name final variables using capital letters:

```
final String STATE_CAPITAL="Washington";
```

In addition to primitive data types, you can also use Java classes to declare variables. Each primitive data type has a corresponding *wrapper* class, for example `Integer`, `Double`, `Boolean`, etc. These classes have useful methods to convert data from one type to another.

While a `char` data type is used to store only one character, Java also has a class `String` for working with a longer text, for example:

```
String lastName="Smith";
```

In Java, variable names can not start with a digit and can not contain spaces.

A *bit* is the smallest piece of data that can be stored in memory. It can hold either 1 or 0.

A `byte` consists or eight bits.

A `char` in Java occupies two bytes in memory.

An `int` and a `float` in Java take four bytes of memory.

Variables of `long` and `double` types use eight bytes each.

Numeric data types that use more bytes can store larger numbers.

1 kilobyte (KB) has 1024 bytes

1 megabyte (MB) has 1024 kilobytes

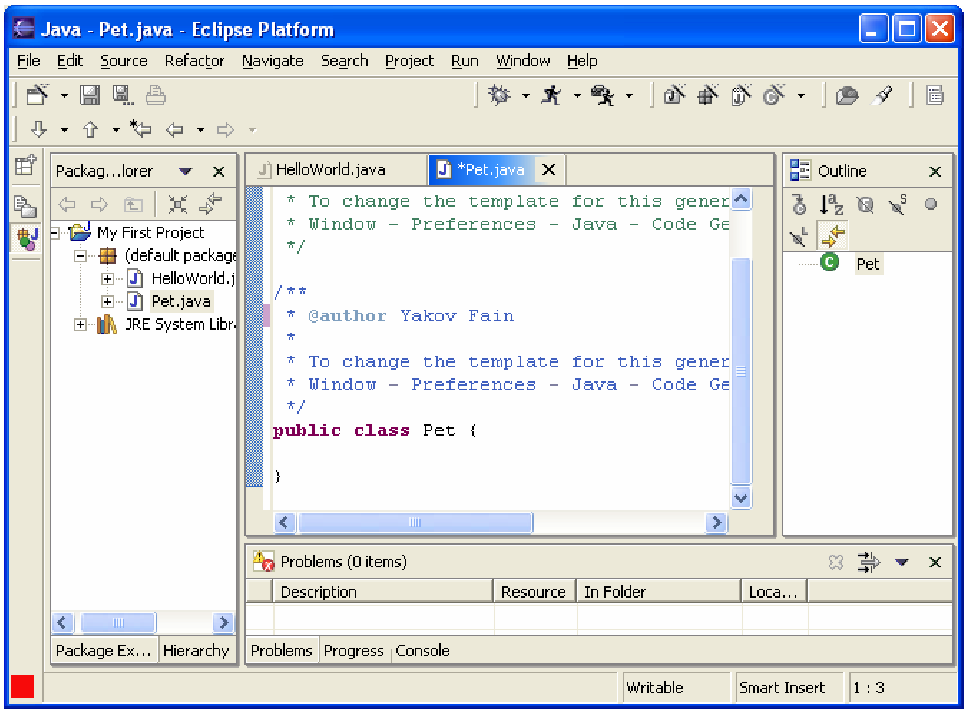
1 gigabyte (GB) has 1024 megabytes

Creation of a Pet

Let's design and create a class `Pet`. First we need to decide what actions our pet will be able to do. How about eat, sleep, and say? We'll program these actions in the methods of the class `Pet`. We'll also give our pet the following attributes: age, height, weight, and color.

Start with creating a new Java class called `Pet` in *My First Project* as described in Chapter 2, but do not mark the box for creation of the method `main()`.

Your screen should look similar to this one:



```
 * To change the template for this gener
 * Window - Preferences - Java - Code Ge
 */

/**
 * @author Yakov Fain
 *
 * To change the template for this gener
 * Window - Preferences - Java - Code Ge
 */
public class Pet {

}
```

Now we are ready to declare attributes and methods in the class `Pet`. Java classes and methods enclose their bodies in curly braces. Every open curly brace must have a matching closing brace:

```
class Pet{
}
```

To declare variables for class attributes we should pick data types for them. I suggest an `int` type for the age, `float` for weight and height, and `String` for a pet's color.

```
class Pet{
   int age;
   float weight;
   float height;
   String color;
}
```

The next step is to add some methods to this class. Before declaring a method you should decide if it should take any arguments and return a value:

✓ The method `sleep()` will just print a message *Good night, see you tomorrow* – it does not need any arguments and will not return any value.

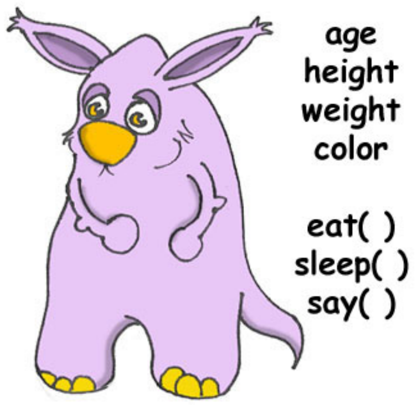
- ✓ The same is true for the method `eat()`.It will print the message *I'm so hungry…let me have a snack like nachos!*.

- ✓ The method `say()` will also print a message, but the pet will "say" (print) the word or a phrase that we give to it. We'll pass this word to the method `say()` as a *method argument*. The method will build a phrase using this argument and will return it back to the calling program.

The new version of the class `Pet` will look like this:

```
public class Pet {
 int age;
 float weight;
 float height;
 String color;

 public void sleep(){
        System.out.println(
                    "Good night, see you tomorrow");
 }

 public void eat(){
  System.out.println(
  "I'm so hungry…let me have a snack like nachos!");
 }

 public String say(String aWord){
      String petResponse = "OK!! OK!! " +aWord;
            return petResponse;
 }
}
```

This class represents a friendly creature from the real world:



Let's talk now about the signature of the method `sleep()`:

```
public void sleep()
```

It tells us that this method can be called from any other Java class (`public`), it does not return any data (`void`). The empty parentheses mean that this method does not have any arguments, because it does not need any data from the outside world – it always prints the same text.

The signature of the method `say()` looks like this:

```
public String say(String aWord)
```

This method can also be called from any other Java class, but has to return some text, and this is the meaning of the keyword `String` in front of the method name. Besides, it expects some text data from outside, hence the argument `String aWord`.

How do you decide if a method should or should not return a value? If a method performs some data manipulations and has to give the result of these manipulations back to a calling class, it has to return a value. You may say, that the class `Pet` does not have any calling class! That's correct, so let's create one called `PetMaster`. This class will have a method `main()` containing the code to communicate with the class `Pet`. Just create another class `PetMaster`, and this time select the option in Eclipse that creates the method `main()`. Remember, without this method you can not *run* this class as a program. Modify the code generated by Eclipse to look like this:

```
public class PetMaster {

  public static void main(String[] args) {

   String petReaction;

   Pet myPet = new Pet();

   myPet.eat();
   petReaction = myPet.say("Tweet!! Tweet!!");
   System.out.println(petReaction);

   myPet.sleep();

  }
}
```

Do not forget to press *Ctrl-S* to save and compile this class!
To run the class `PetMaster` click on the Eclipse menus *Run*, *Run…*, *New* and type the name of the main class: `PetMaster`. Push the button *Run* and the program will print the following text:

```
I'm so hungry…let me have a snack like nachos!
OK!! OK!! Tweet!! Tweet!!
Good night, see you tomorrow
```

The `PetMaster` is the *calling class*, and it starts with creating an *instance* of the object `Pet`. It declares a variable `myPet` and uses the Java operator `new`:

```
Pet myPet = new Pet();
```

This line declares a variable of the type `Pet` (that's right, you can treat any classes created by you as new Java data types). Now the variable `myPet` knows where the `Pet` instance was created in the computer's memory, and you can use this variable to call any methods from the class `Pet`, for example:

```
myPet.eat();
```

If a method returns a value, you should call this method in a different way. Declare a variable that has the same type as the return value of the method, and assign it to this variable. Now you can call this method:

```
String petReaction;

petReaction = myPet.say("Tweet!! Tweet!!");
```

At this point the returned value is stored in the variable `petReaction` and if you want to see what's in there, be my guest:

```
System.out.println(petReaction);
```

Inheritance – a Fish is Also a Pet

Our class `Pet` will help us learn yet another important feature of Java called *inheritance*. In the real life, every person inherits some features from his or her parents. Similarly, in the Java world you can also create a new class, based on the existing one.

The class `Pet` has behavior and attributes that are shared by many pets – they eat, sleep, some of them make sounds, their skins have different colors, and so on. On the other hand, pets are different - dogs bark, fish swim and do not make sounds, parakeets talk better than dogs. But all of them eat, sleep, have weight and height. That's why it's easier to create a class `Fish` that will *inherit* some common behaviors and attributes from the class `Pet`, rather than creating `Dog`, `Parrot` or `Fish` from scratch every time.

A special keyword `extends` that will do the trick:

```
class Fish extends Pet{

}
```

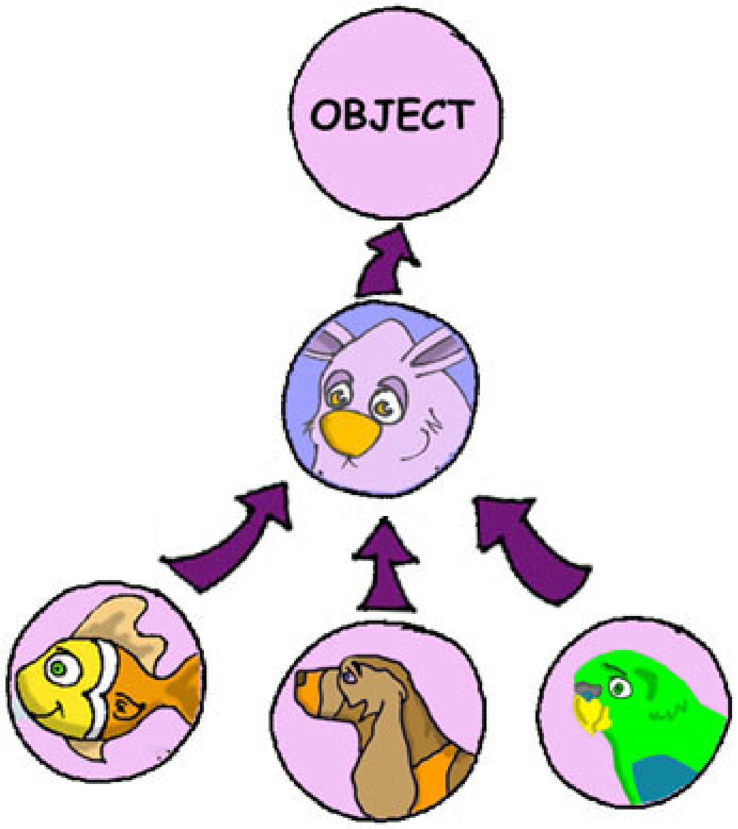
You can say that our `Fish` is a *subclass* of the class `Pet`, and the class `Pet` is a *superclass* of the class `Fish`. In other words, you use the class `Pet` as a template for creating a class `Fish`.

Even if you will leave the class `Fish` as it is now, you can still use every method and attribute inherited from the class `Pet`. Take a look:

```
Fish myLittleFish = new Fish();
myLittleFish.sleep();
```

Even though we have not declared any methods in the class `Fish` yet, we are allowed to call the method `sleep()` from its superclass!

Creation of subclasses in Eclipse is a piece of cake! Select the menus *File*, *New*, *Class*, and type `Fish` as the name of the class. Replace the `java.lang.Object` in the field superclass with the word `Pet`.



Let's not forget, however, that we're creating a subclass of a `Pet` to add some new features that only fish have, and reuse some of the code that we wrote for a general pet.

It's time to reveal a secret – all classes in Java are inherited from the super-duper class `Object`, regardless if you do use the word `extends` or not.

But Java classes can not have two separate parents. If this would happen with people, kids would not be subclasses of their parents, but all the boys would descendents of Adam, and all the girls descendents of Eve ☺.

Not all pets can dive, but fish certainly can. Let's add a new method `dive()` to the class `Fish` now.

```
public class Fish extends Pet {

   int currentDepth=0;

   public int dive(int howDeep){
      currentDepth=currentDepth + howDeep;
      System.out.println("Diving for " + howDeep +
                                           " feet");
      System.out.println("I'm at " + currentDepth +
                             " feet below sea level");
      return currentDepth;
   }
}
```

The method `dive()` has an *argument* `howDeep` that tells the fish how deep it should go. We've also declared a class variable `currentDepth` that will store and update the current depth every time you call the method `dive()`. This method returns the current value of the variable `currenDepth` to the calling class.

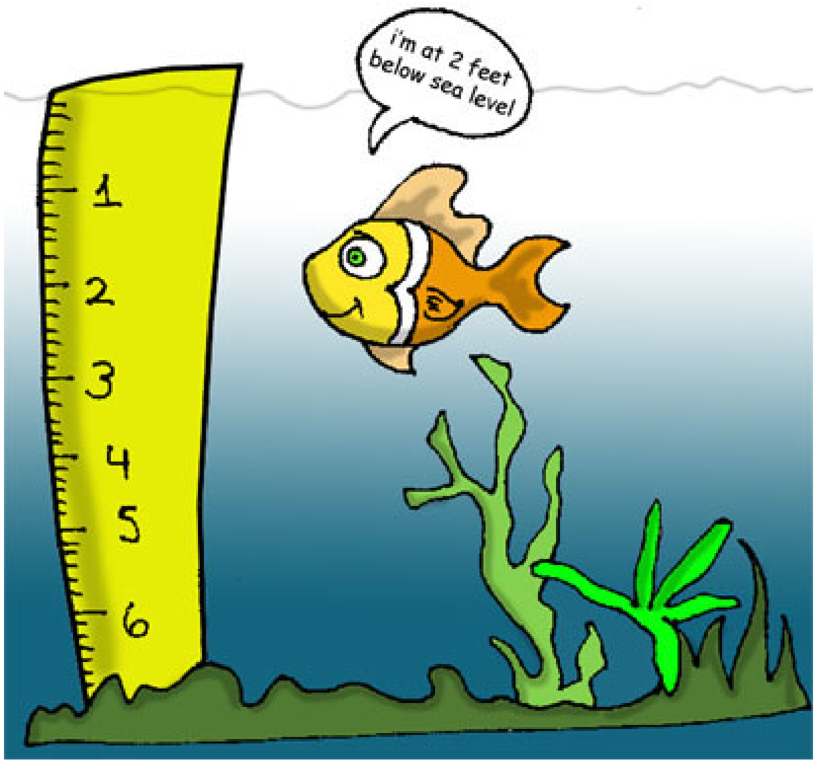
Please create another class `FishMaster` that will look like this:

```
public class FishMaster {

      public static void main(String[] args) {

             Fish myFish = new Fish();

             myFish.dive(2);
             myFish.dive(3);

             myFish.sleep();
      }
}
```

The method main() instantiates the object Fish and calls its method dive() twice with different arguments. After that, it calls the method sleep(). When you run the program FishMaster, it will print the following messages:

```
Diving for 2 feet
I'm at 2 feet below sea level
Diving for 3 feet
I'm at 5 feet below sea level
Good night, see you tomorrow
```

Have you noticed that beside methods defined in the class Fish, the FishMaster also calls methods from its superclass Pet? That's the whole point of inheritance – you do not have to copy and paste code from the class Pet – just use the word extends, and the class Fish can use Pet's methods!



One more thing, even though the method dive() returns the value of currentDepth, our FishMaster does not use it. That's fine, our FishMaster does not need this value, but there may be some other classes that will also use Fish, and they may find it useful. For example, think of a class FishTrafficDispatcher that has to know positions of other fish under the sea before allowing diving to avoid traffic accidents ☺.

Method Overriding

As you know, fish do not speak (at least they do not do it aloud). But our class `Fish` has been inherited from the class `Pet` that has a method `say()`. This means that nothing stops you from writing something like this:

```
myFish.say();
```

Well, our fish started to talk… If you do not want this to happen, the class `Fish` has to *override* the Pet's method `say()`. This is how it works: if you declare in a subclass a method with exactly the same signature as in its superclass, the method of the subclass will be used instead of the method of the superclass. Let's add the method `say()` to the class `Fish`.

```
public String say(String something){
  return "Don't you know that fish do not talk?";
}
```

Now add the following method call to the method `main()` of the class `FishMaster`:

```
myFish.say("Hello");
```

Run the program and it'll print

```
Don't you know that fish do not talk?
```

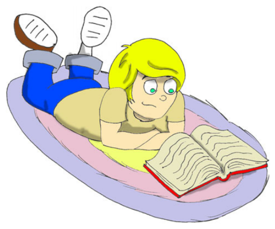
This proves that `Pet`'s method `say()` has been *overridden*, or in other words suppressed.

If a method signature includes the keyword `final`, such method can not be overridden, for example:

```
final public void sleep(){…}
```

Wow! We've learned a lot in this chapter – let's just take a break.

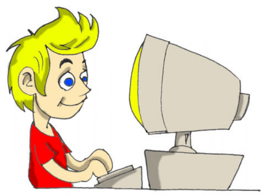
Additional Reading



1.Java Data Types:
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

2.About inheritance:
http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html

Practice



1. Create a new class `Car` with the following methods:

```
public void start()
public void stop()
public int drive(int howlong)
```

The method `drive()` has to return the total distance driven by the car for the specified time. Use the following formula to calculate the distance:

```
distance = howlong*60;
```

2. Write another class `CarOwner` and that creates an instance of the object `Car` and call its methods. The result of each method call has to be printed using `System.out.println()`.

Practice for Smarty Pants

Create a subclass of `Car` named `JamesBondCar` and override the method `drive()` there. Use the following formula to calculate the distance:

```
distance = howlong*180;
```

Be creative, print some funny messages!

Chapter 4. Java Building Blocks

You can add any text comments to your program to explain what a particular line, method or a class is for. Sometimes people forget why they have written the program this way. The other reason for writing comments is to help other programmers understand you code.

Program Comments

There are three different types of comments:

- If your comment fits in one line, start it with two slashes:

```
// This method calculates the distance
```

- Longer multi-line comments have to be surrounded with these symbols: `/*` and `*/`, for example:

```
/* the next 3 lines store the current
   position of the Fish.
*/
```

- Java comes with a special program `javadoc` that can extract all comments from your programs into a separate help file. This file can be used as a *technical documentation* for your programs. Such comments are enclosed in symbols `/**` and `*/`. Only the most important comments like description of the class or a method should be placed between these symbols.

```
/** This method calculates the discount that depends
   on the price. If the price is more than $100,
   it gives you 20% off, otherwise only 10%.
*/
```

From now on, I'll be adding comments to the code samples to give you a better idea how and where to use them.

Making Decisions with `if` Statements

We always make decisions in our life: *If she is going to tell me this – I'm going to answer that, otherwise I'll do something else.* Java has an `if` statement that checks if a particular *expression* is `true` or `false`.

Based on the result of this expression, your program execution splits, and only the one matching portion of the code will work.

For example, if an expression *Do I want to go to grandma?* returns `true`, turn to the left, otherwise turn to the right.

If an expression *returns* *`true`*, JVM will execute the code between the first curly braces, otherwise it goes to the the code after `else` statement. For example, if a price is more than a hundred dollars, give a 20% discount, otherwise take only 10% off.

```
// More expensive goods get 20% discount
if (price > 100){
  price=price*0.8;
  System.out.println("You'll get a 20% discount");
}
else{
  price=price*0.9;
  System.out.println("You'll get a 10% discount");
}
```

Let's modify the method `dive()` in the class `Fish` to make sure that our fish will never dive below 100 feet:

```
public class Fish extends Pet {
  int currentDepth=0;
  public int dive(int howDeep){
    currentDepth=currentDepth + howDeep;
      if (currentDepth > 100){
          System.out.println("I am a little fish and "
                        + " can't dive below 100 feet");
          currentDepth=currentDepth - howDeep;
      }else{
        System.out.println("Diving for " + howDeep +
                                   " feet");
        System.out.println("I'm at " + currentDepth +
                          " feet below the sea level");
      }
      return currentDepth;
      }
      public String say(String something){
       return "Don't you know that fish do not talk?";
      }
}
```

Now just make a little change to the `FishMaster` – let it try to make our fish go deep under the sea:

```
public class FishMaster {

  public static void main(String[] args) {

    Fish myFish = new Fish();

    // Try to have the fish go below 100 feet
      myFish.dive(2);
      myFish.dive(97);
      myFish.dive(3);

      myFish.sleep();
  }
}
```

Run this program and it'll print the following:

```
Diving for 2 feet
I'm at 2 feet below the sea level
Diving for 97 feet
I'm at 99 feet below the sea level
I am a little fish and  can't dive below 100 feet
Good night, see you tomorrow
```

Logical Operators

Sometimes, to make a decision you may need to check more than one conditional expression, for example if the name of the state is Texas or California, add the state tax to the price of every item in the store. This is an example of the *logical or* case – either Texas or California. In Java the sign for a logical `or` is one ore two vertical bars. It works like this – if any of the two conditions is true, result of the entire expression is `true`. In the following examples I use use a variable of type `String`. This Java class has a method `equals()`, and I use it to compare the value of the variable state with *Texas* or *California*:

```
if (state.equals("Texas") | state.equals("California"))
```

You can also write this `if` statement using two bars:

```
if (state.equals("Texas") || state.equals("California"))
```

The difference between the two is that if you use two bars, and the first expression is `true`, the second expression won't even be checked. If you place just a single bar, JVM will evaluate both expressions.

The *logical and* is represented by one or two ampersands (`&&`) and the whole expression is `true` if every part of it is `true`. For example, charge the sales tax only if the state is New York and the price is more than $110. Both conditions must be `true` *at the same time*:

```
if (state.equals("New York") && price >110)
```

or

```
if (state.equals("New York") & price >110)
```

If you use double ampersand and the first expression is `false`, the second one won't even be checked, because the entire expression will be `false` anyway. With the single ampersand both expressions will be evaluated.

The *logical not* is represented by the exclamation point, and it changes expression to the opposite meaning. For example, if you want to perform some actions only if the state is not New York, use this syntax:

```
if (!state.equals("New York"))
```

Here's anoher example - the following two expressions will produce the same result:

```
if (price < 50)

if (!(price >=50))
```

The *logical not* here is applied to the expression in parentheses.

Conditional operator

There is another flavor of an `if` statements called *conditional operator*. This statement is used to assign a value to a variable based on an expression that ends with a question mark. If this expression is true, the value after the question mark is used, otherwise the value after the colon is assigned to the variable on the left:

```
discount = price > 50? 10:5;
```

If the `price` is greater than fifty, the variable `discount` will get the value of 10, otherwise the value of 5. It's just a shorter replacement of a regular `if` statement:

```
if (price > 50){
   discount = 10;
} else {
   discount = 5;
}
```

Using `else if`

You are also allowed to build more complex `if` statements with several `else if` blocks. This time we'll create a new class called `ReportCard`. This class has to have the method `main()` and also a method that will have one argument - numeric test result. Depending on the number, it should print your grade like A, B, C, D, or F. We'll name this method `convertGrades`().

```
public class ReportCard {

/**
 This method takes one integer argument - the result of
the  test and returns one letter A, B, C or D depending
on the argument.
*/
   public char convertGrades( int testResult){
      char grade;

      if (testResult >= 90){
            grade = 'A';
      }else if (testResult >= 80 && testResult < 90){
            grade = 'B';
      }else if (testResult >= 70 && testResult < 80){
            grade = 'C';
      }else {
            grade = 'D';
      }
      return grade;
   }

   public static void main(String[] args){

      ReportCard rc = new ReportCard();

      char yourGrade = rc.convertGrades(88);
      System.out.println("Your first grade is " +
                                    yourGrade);

      yourGrade = rc.convertGrades(79);
      System.out.println("Your second grade is " +
                                    yourGrade);
   }
}
```

Beside using the `else if` condition, this example also shows you how to use variables of type `char`. You can also see that with the `&&` operator you can check if a number falls into some range. You can not write simply `if testResult between 80 and 89`, but in Java we write that at the same time `testResult` has to be greater or equal to 80 *and* less then 89:

```
testResult >= 80 && testResult < 89
```

Think about why we could not use the `||` operator here.

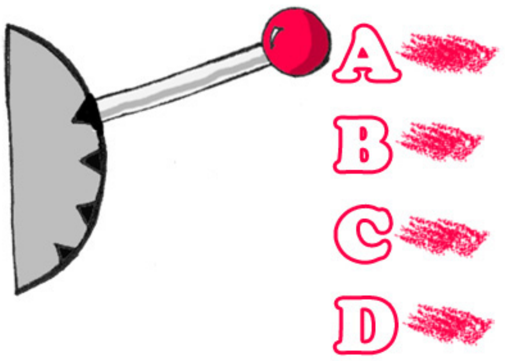
Making Decisions With `switch` Statement

The `switch` statement sometimes can be used as an alternative to `if`. The variable after the keyword `switch` is evaluated, and program goes only to one of the `case` statements:

```
public static void main(String[] args){

 ReportCard rc = new ReportCard();
 char yourGrade = rc.convertGrades(88);

  switch (yourGrade){

    case 'A':
       System.out.println("Excellent Job!");
       break;
    case 'B':
       System.out.println("Good Job!");
       break;
    case 'C':
       System.out.println("Need to work more!");
       break;
    case 'D':
       System.out.println("Change your attitude!");
       break;
 }
}
```

Do not forget to put the keyword `break` at the end of each `case` – the code has to jump out of the `switch` statement. Without the `break` statements this code will print all four lines, even though the variable `yourGrade` will have only one value.

Java `switch` statement has a restriction – the variable that's being evaluated must have one of these types:
`char`
`int`
`byte`
`short`.



How Long Variables Live?

Class `ReportCard` declares a variable `grade` inside the method `convertGrades()`. If you declare a variable inside any method, such variable is called *local*. This means that this variable is available only for the code *within this method*. When the method completes, this variable automatically gets removed from memory.

Programmers also use the word *scope* to say how long a variable will live, for example you can say that the variables declared inside a method have a local scope.

If a variable has to be reused by several method calls, or it has to be visible from more than one method in a class, you should declare such variable outside of any method. In class `Fish`, `currentDepth` is a *member variable*. These variables are "alive" until the instance of the object `Fish` exists in memory, that's why they are also called *instance variables*. They could be shared and reused by all methods of the class, and in some cases they can even be visible from external classes, for example in our classes the *statement* `System.out.println()` is using the class variable `out` that was declared in the class `System`.

Wait a minute! Can we even use a member variable from the class `System` if we have not created an instance of this class? Yes we can, if this variable was declared with a keyword `static`. If declaration of a member variable or a method starts with `static`, you do not have to create an instance of this class to use it. Static members of a class are used to store the values that are the same for all instances of the class.

For example, a method `convertGrades()` can be declared as `static` in the class `ReportCard`, because its code does not use member variables to read/store values specific to a particular instance of the class. This is how you call a `static` method:

```
char yourGrade = ReportCard.convertGrades(88);
```

Here's another example: there is a class `Math` in Java that contains several dozens of mathematical methods like `sqrt()`, `sin()`, `abs()` and others. All these methods are `static` and you do not need to create an instance of the class `Math` to call them, for example:

```
double squareRoot = Math.sqrt(4.0);
```

Special Methods: Constructors

Java uses operator `new` to create instances of objects in memory, for example:

```
Fish myFish = new Fish();
```

Parentheses after the word `Fish` tell us that this class has some method called `Fish()`. Yes, there are special methods that are called *constructors* , and these methods have the following features:

- Constructors are called only once during construction of the object in memory.
- They must have the same name as the class itself.
- They do not return a value, and you do not even have to use the keyword `void` in constructor's signature.

Any class can have more than one constructor. If you do not create a constructor for the class, Java automatically creates during the compilation time so-called *default no-argument constructor*. That's why Java compiler has never complained about such statement as `new Fish()`, even though the class `Fish` did not have any constructors.

In general, constructors are used to assign initial values to member variables of the class, for example the next version of class `Fish` has one-argument constructor that just assigns the argument's value to the instance variable `currentDepth` for future use.

```
public class Fish extends Pet {
    int currentDepth;

      Fish(int startingPosition){
            currentDepth=startingPosition;
      }
}
```

Now the class `FishMaster` can create an instance of the `Fish` and assign the initial position of the fish. The next example creates an instance of the `Fish` that is "submerged" 20 feet under the sea:

```
Fish myFish = new Fish(20);
```

If a constructor with arguments has been defined in a class, you can no longer use default no-argument constructor. If you'd like to have a constructor without arguments - write one.

The Keyword `this`

The keyword `this` is useful when you need to refer to the instance of the object you are in. Look at the next example:

```
class Fish {
  int currentDepth ;

  Fish(int currentDepth){
      this.currentDepth = currentDepth;
  }
}
```

A keyword this helps to avoid name conflicts, for example this.currentDepth refers to a member variable currentDepth, while the currentDepth refers to the argument's value.

In other words, the instance of the object Fish is pointing to itself.

You'll see another important example of using keyword this in Chapter 6 in the section *How to Pass Data Between Classes*.

Arrays

Let's say your program has to store names of the four game players. Instead of declaring four different String variables, you can declare one String *array* that has four *elements*.
Arrays are marked by placing square brackets either after the variable name, or after the data type:

```
String [] players;
```

or

```
String players[];
```

These lines just tells Java compiler that you are planning to store several text strings in the array players. Each element has its own index starting from zero. The next sample actually creates an instance of an array that can store four String elements and assigns the values to the elements of this array:

```
players = new String [4];

players[0] = "David";
players[1] = "Daniel";
players[2] = "Anna";
players[3] = "Gregory";
```

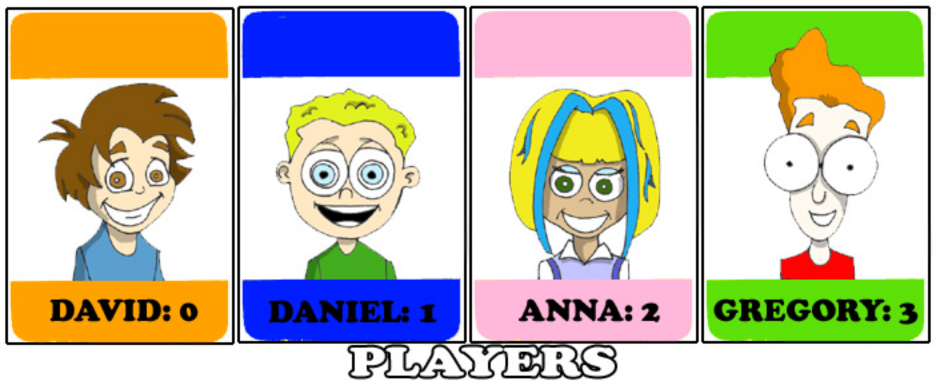
You must know the size of the array before assigning values to its elements. If you do not know in advance how many elements you are going to have, you can not use arrays, but should look into other Java classes, for example `Vector`, but let's concentrate on arrays at this point.

Any array has an attribute called `length` that "remembers" the number of elements in this array, and you can always find out how many elements are there:

```
int  totalPlayers = players.length;
```

If you know all the values that will be stored in the array at the time when you declare it, Java allows you to declare and initialize such array in one shot:

```
String [] players = {"David", "Daniel", "Anna", "Gregory"};
```



Imagine that the second player is a winner and you'd like to print congratulations to this kid. If the players' name are stored in an array, we need to get its second element:

```
String theWinner = players[1];
System.out.println("Congratulations, " + theWinner + "!");
```

Here's the output of this code:

```
Congratulations, Daniel!
```

Do you know why the second element has the index [1]? Of course you do, because the index of the first element is always [0].

Array of players in our example are *one-dimensional*, because we store them sort of in a row. If we wanted the store the values as a matrix, we can create a two-dimensional array. Java allows creation of *multi-dimensional* arrays. You can store any objects in arrays, and I'll show you how to do this in Chapter 10.

Repeating Actions with Loops

Loops are used to repeat the same action multiple times, for example we need to print congratulation to several winners.
When you know in advance how many times this action has to be repeated - use a loop with a keyword `for`:

```
int  totalPlayers = players.length;
int counter;

for (counter=0; counter <totalPlayers; counter++){
 String thePlayer = players[counter];
 System.out.println("Congratulations,"+
                                    thePlayer+"!");
}
```

JVM executes every line between the curly braces and then returns back to the first line of the loop to increment the counter and check the conditional expression. This code means the following:

Print the value of the array element whose number is the same as the current value of the `counter`*. Start from the element number 0 (*`counter=0`*), and increment the value of the* `counter` *by one (*`counter++`*). Keep doing this while the* `counter` *is less than* `totalPlayers (counter<totalPlayers)`*.*

There is another keyword for writing loops - `while`. In these loops you do not have to know exactly how many times to repeat the action, but you still need to know when to end the loop. Let's see how we can congratulate players using the `while` loop that will end when the value of the variable `counter` becomes equal to the value of `totalPlayers`:

```
int  totalPlayers = players.length;
int counter=0;

while (counter< totalPlayers){
  String thePlayer = players[counter];
  System.out.println("Congratulations, "
                              + thePlayer + "!");
  counter++;
}
```

In Chapter 9 you'll learn how to save data on the disks and how to read them back into computer's memory. If you read game scores from the disk file, you do not know in advance how many scores were saved there. Most likely you'll be reading the scores using the `while` loop.

You can also use two important keywords with loops: `break` and `continue`.

The keyword `break` is used to jump out of the loop when some particular condition is `true`. Let's say we do not want to print more than 3 congratulations, regardless of how many players we've got. In the next example, after printing the array elements 0, 1 and 2, the `break` will make the code go out of the loop and the program will continue from the line after the closing curly brace.

The next code sample has a double equal sign in the `if` statement. This means that you are comparing the value of the variable `counter` with number 3. A single equal sign in the here would mean assignment of the value of 3 to the variable `counter`. Replacing == with = in an `if` statement is a very tricky mistake, and it can lead to unpredictable program errors that may not be so easy to find.

```
int counter =0;
while (counter< totalPlayers){

  if (counter == 3){
   break; // Jump out of the loop
  }
  String thePlayer = players[counter];
  System.out.println("Congratulations, "+thePlayer+ "!");
  counter++;
}
```

The keyword `continue` allows the code to skip some lines and return back to the beginning of the loop. Imagine that you want to congratulate everyone but David – the keyword `continue` will return the program back to the beginning of the loop:

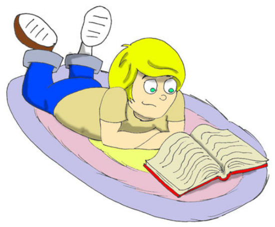
```
while (counter< totalPlayers){
  counter++;

  String thePlayer = players[counter];

  if (thePlayer.equals("David"){
      continue;
  }
  System.out.println("Congratulations, "+ thePlayer+ !");
}
```

There is yet another flavor of the `while` loop that starts with the word `do`, for example:

```
do {
  // Your code goes here
 } while (counter< totalPlayers);
```

Such loops check an expression *after* executing the code between curly braces, which means that code in the loop will be executed *at least once*. Loops that start with the keyword `while` might not be executed at all if the loop expression is false to begin with.

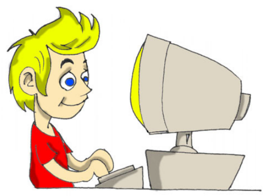
Additional Reading



1. jGuru: Language Essentials. Short Course: http://java.sun.com/developer/onlineTraining/JavaIntro/contents.html

2.Scope of variables: http://java.sun.com/docs/books/tutorial/java/nutsandbolts/scope.html

Practice



1. Create a new class named `TemperatureConverter` that will have a method with the following signature:

```
public String convertTemp
       (int temperature, char convertTo)
```

If the value of the argument `convertTo` is `F`, the temperature has to be converted to Fahrenheit, and if it's `C`, convert it to Celsius. When you'll be calling this method, put the value of the argument `char` in single quotes.

2. Declare a method `convertGrades()` of the class `ReportCard` as `static` and remove the line that instantiates this class from the method `main()`.

Practice for Smarty Pants

Have you noticed that in the example with the keyword `continue` we've moved up the line `counter++;`?
What would have happened if we left this line at the end of the loop as it was in the example `with break`?

Chapter 5. A Graphical Calculator

Java comes with the whole bunch of classes that you'll be using to create graphical applications. There are two main groups of classes (libraries) that are used for creating windows in Java: AWT and Swing.

AWT and Swing

When Java was originally created, only AWT library was available for working with graphics. This library is a simple set of classes like `Button`, `TextField`, `Label` and others. Pretty soon, another and more advanced library called Swing was introduced. It also includes buttons, text fields, and other window controls. The names of the Swing components start with the letter `J`, for example `JButton`, `JTextField`, `JLabel`, and so on.

Everything is a little better, faster, and more convenient in Swing, but in some cases our programs will run on computers with older JVMs that may not support Swing classes. You'll see the examples of working with AWT later in Chapter 7, but in this chapter we'll create a calculator program using Swing.

There is yet another set of Java classes which is a part of Eclipse platform called Standard Widget Toolkit (SWT), but we won't use it in this book.

Packages and Import Statements

Java comes with many useful classes that are organized in *packages*. Some packages include classes responsible for drawing, while other packages have classes to work with the Internet, and so on. For example the class `String` is located in the package called `java.lang`, and the full name of the class `String` is `java.lang.String`.

Java compiler knows where to find classes that are located in `java.lang`, but there are many other packages with useful classes, and it's your responsibility to let the compiler know where the classes from your program live. For example, most of the Swing classes live in one of the following two packages:

```
javax.swing
javax.swing.event
```

It would be annoying to write a full class name every time you use it, and to avoid this you can write `import` statements just once above the class declaration line, for example:

```
import javax.swing.JFrame;
import javax.swing.JButton;

class Calculator{
  JButton myButton = new JButton();
  JFrame myFrame = new JFrame();
}
```

These `import` statements will allow you to use the short class names like `JFrame` or `JButton`, and Java compiler will know where to look for these classes.

If your need to use several classes from the same package, you do not have to list each of them in the `import` statement, just use the *wild card*. In the following example the star (asterisk) makes all classes from the package `javax.swing` *visible* to your program:

```
import javax.swing.*;
```

Still, it's better to use separate import statements, so you can see what exactly the class is importing from each package. We'll talk more about Java packages in Chapter 10.

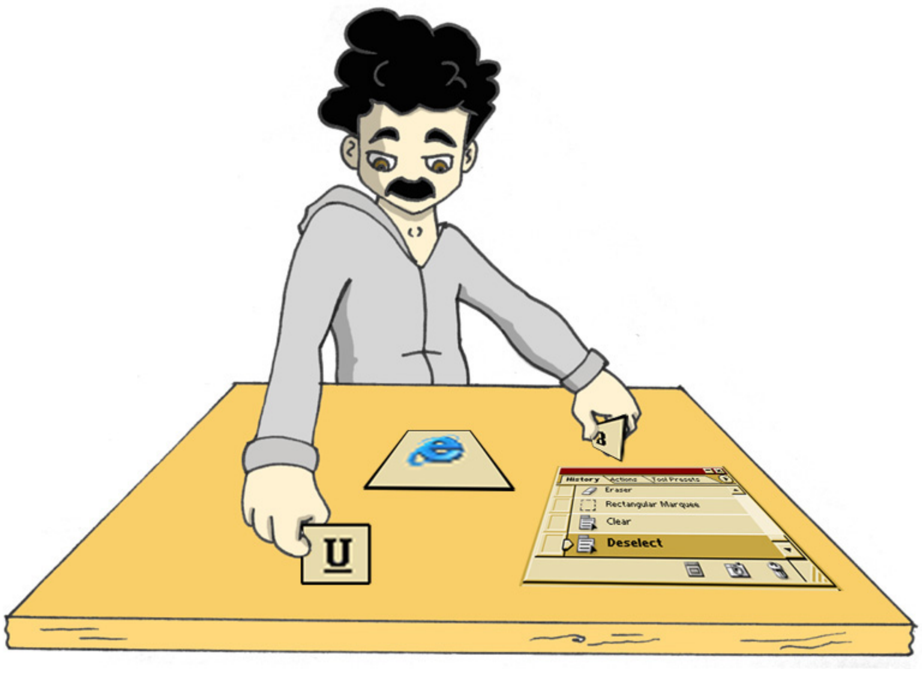
Major Swing Elements

These are some of the major objects that Swing applications consist of:

- A window or a *frame* that can be created using the class `JFrame`.

- An invisible *panel* or a *pane* that holds all these buttons, text fields, labels, and other components. Panels are created by the class `JPanel`.
- Window controls like buttons (`JButton`), text fields (`JTextfield`), lists (`JList`), and so on.
- Layout managers that help arrange all these buttons and fields on a panel.

Usually a program creates an instance of a `JPanel` and assigns the layout manager to it. Then, it can create some window controls and add them to the panel. After that, add the panel to the frame, set the frame's size and make it visible.



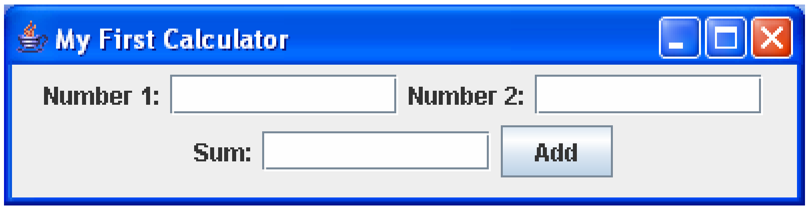
But displaying a frame is only half of the job, because the window controls should know how to respond to various events, for example a click on the button.

In this chapter we'll learn how to display nice-looking windows, and the next chapter is about writing code that will respond to *events* that may happen with elements of this window.

Our next goal is to create a simple calculator that knows how to add two numbers and display the result. Create a new project in Eclipse named *My Calculator* and add a new class `SimpleCalculator` with the following code:

```java
import javax.swing.*;
import java.awt.FlowLayout;

public class SimpleCalculator {
 public static void main(String[] args) {
  // Create a panel
      JPanel windowContent= new JPanel();

  // Set a layout manager for this panel
      FlowLayout fl = new FlowLayout();
      windowContent.setLayout(fl);
  // Create controls in memory
      JLabel label1 = new JLabel("Number 1:");
      JTextField field1 = new JTextField(10);
      JLabel label2 = new JLabel("Number 2:");
      JTextField field2 = new JTextField(10);
      JLabel label3 = new JLabel("Sum:");
      JTextField result = new JTextField(10);
      JButton go = new JButton("Add");

  // Add controls to the panel
      windowContent.add(label1);
      windowContent.add(field1);
      windowContent.add(label2);
      windowContent.add(field2);
      windowContent.add(label3);
      windowContent.add(result);
      windowContent.add(go);

  // Create the frame and add the panel to it
  JFrame frame = new JFrame("My First Calculator");

  frame.setContentPane(windowContent);

  // set the size and make the window visible
  frame.setSize(400,100);
  frame.setVisible(true);
 }
}
```

Compile and run this program and it'll display a window that looks like this one:



This may not be the best-looking calculator, but it'll give us a chance to learn how to add components and display a window. Iin the next section we'll try make it look better with the help of *layout managers*.

Layout Managers

Some old-fashioned programming languages force you to set exact coordinates and sizees of each window component. This works fine if you know the screen settings (*resolution*) of all people that will use your program. By the way, we call people who use your programs *users*. Java has layout managers that help you arrange components on the screen without assigning strict positions to the window controls. Layout managers will ensure that their screen will look nice regardless of the window size.

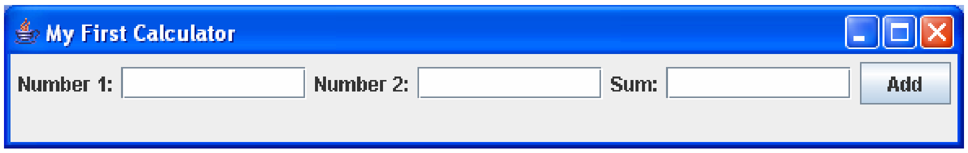
Swing offers the following layout managers:

- `FlowLayout`
- `GridLayout`
- `BoxLayout`
- `BorderLayout`
- `CardLayout`
- `GridBagLayout`

To use any layout manager, a program needs to instantiate it, and then assign this object to a *container* , for example to a panel as in the class `SimpleCalculator`.

Flow Layout

This layout arranges components in a window row by row. For example, labels, text fields and buttons will be added to the first imaginary row until there is room there. When the current row is filled, the rest of the components will go to the next row, and so on. If a user changes the size of the window, it may mess up the picture. Just grab the corner of our calculator window and resize it. Watch how the manager `java.awt.FlowLayout` rearranges controls as the size of the window changes.



In the next code sample, the keyword `this` represents an instance of the object `SimpleCalculator`.

```
FlowLayout fl = new FlowLayout();
this.setLayoutManager(fl);
```

Well, the `FlowLayout` is not the best choice for our calculator. Let's try something different now.

Grid Layout

The class `java.awt.GridLayout` allows you to arrange components as *rows* and *columns* in a grid. You'll be adding components to imaginary cells of this grid. If the screen gets resized, grid cells may become bigger, but the relative positions of window components will stay the same. Our calculator has seven components – three labels, three text fields and a button. We may arrange them as a grid of four rows and two columns (one cell stays empty):

```
GridLayout gr = new GridLayout(4,2);
```

You can also assign some horizontal and vertical space gaps between the cells, for example five pixels:

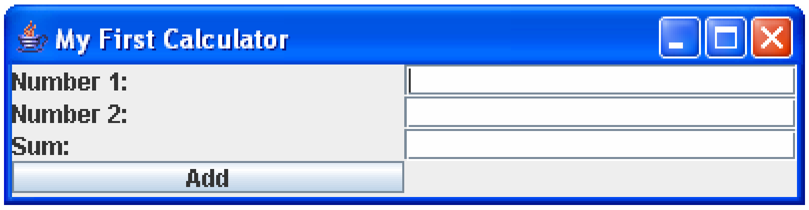
```
GridLayout gr = new GridLayout(4,2,5,5);
```

After minor changes in our calculator (they are highlighted below), our calculator will look a lot prettier.

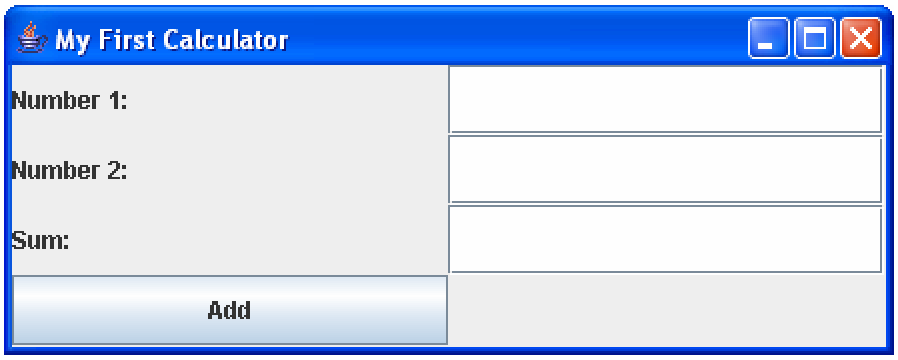
Create and compile a new class `SimpleCalculatorGrid` in the project *My Calculator*.

```
import javax.swing.*;
import java.awt.GridLayout;

public class SimpleCalculatorGrid {
 public static void main(String[] args) {
  // Create a panel
      JPanel windowContent= new JPanel();

  // Set the layout manager for this panel
      GridLayout gl = new GridLayout(4,2);
      windowContent.setLayout(gl);

 // Create controls in memory

      JLabel label1 = new JLabel("Number 1:");
      JTextField field1 = new JTextField(10);
      JLabel label2 = new JLabel("Number 2:");
      JTextField field2 = new JTextField(10);
      JLabel label3 = new JLabel("Sum:");
      JTextField result = new JTextField(10);
      JButton go = new JButton("Add");

 // Add controls to the panel
      windowContent.add(label1);
      windowContent.add(field1);
      windowContent.add(label2);
      windowContent.add(field2);
      windowContent.add(label3);
      windowContent.add(result);
      windowContent.add(go);

  // Create the frame and add the panel to it
      JFrame frame = new JFrame(
                         "My First Calculator");
      frame.setContentPane(windowContent);

 // set the size and display the window
 //frame.pack();
      frame.setSize(400,100);
      frame.setVisible(true);
 }
}
```

Run the program `SimpleCalculatorGrid`, and you'll see this:



Try to resize this window - controls will grow with the window, but their relative positions will not change:



There is one more thing to remember about the grid layout – all cells of the grid have the same width and height.

Border Layout

Class `java.awt.BorderLayout` divides a window into a `South`, `West`, `North`, `East`, and `Center` areas. The `North` area stays always on top of the window, the `South` at the bottom, the `West` is on the left and the `East` is on the right.
For example, in the calculator that is shown the next page, a text field that displays numbers is located in the `North` area.

This is how you can create a `BorderLayout` and place a text field there:

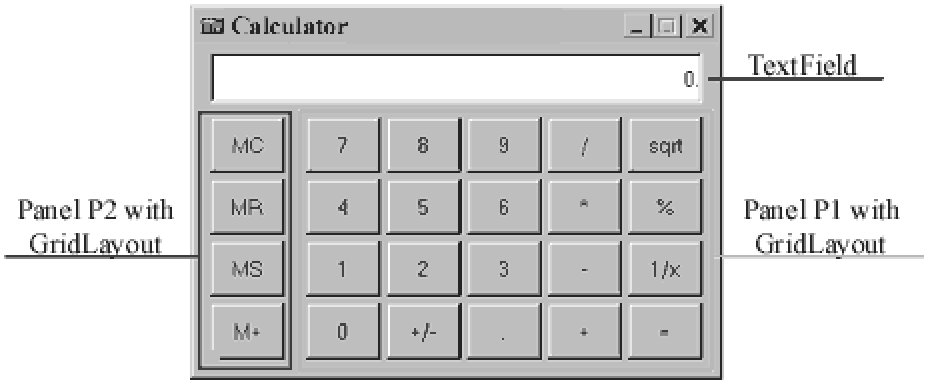
```
BorderLayout bl = new BorderLayout();
this.setLayoutManager(bl);

JTextField  txtDisplay = new JTextField(20);
this.add("North",   txtDisplay);
```

You do not have to put window controls in all five areas. If you only need `North`, `Center`, and `South` areas, the `Center` area will become wider since you are not going to use the `East` and `West`.

I'll use a `BorderLayout` a little later in the next version of our calculator called `Calculator.java`.

Combining Layout Managers

Do you think that the `GridLayout` will allow you to create a calculator window that looks like the one that comes with Microsoft Windows?



Unfortunately it won't, because cells have different sizes in this calculator - the text field is much wider than the buttons. You could combine layout managers using panels that have their own layout managers.

To combine layout managers in the new calculator, let's do the following:

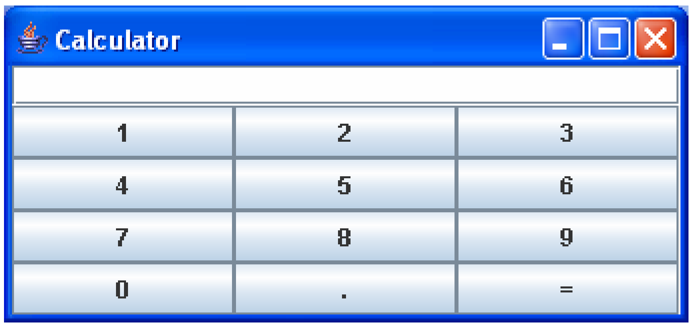
✓ Assign a border layout to the content panel of the frame.

✓ Add a `JTextField` to the `North` area of the screen to display the numbers.

✓ Create a panel `p1` with the `GridLayout`, add 20 buttons to it, and then add `p1` to the `Center` area of the content pane.

✓ Create a panel `p2` with the `GridLayout`, add four buttons to it, then add `p2` to the `West` area of the content pane.

Let's start with a little simpler version of the calculator screen that will look like this:



Create a new class `Calculator` and run the program. Read the program comments in the next code example to understand how it works.

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.BorderLayout;
public class Calculator {
    // Declaration of all calculator's components.
      JPanel windowContent;
      JTextField displayField;
      JButton button0;
      JButton button1;
      JButton button2;
      JButton button3;
      JButton button4;
      JButton button5;
      JButton button6;
      JButton button7;
      JButton button8;
      JButton button9;
      JButton buttonPoint;
      JButton buttonEqual;
      JPanel p1;

      // Constructor  creates the components in memory
      // and adds the to the frame using combination of
      // Borderlayout and Gridlayout
      Calculator(){
        windowContent= new JPanel();

      // Set the layout manager for this panel
        BorderLayout bl = new BorderLayout();
        windowContent.setLayout(bl);

     // Create the display field and place it in the
     // North area of the window
        displayField = new JTextField(30);
        windowContent.add("North",displayField);

     // Create buttons using constructor of the
     // class JButton that takes the label of the
     // button as a parameter
        button0=new JButton("0");
        button1=new JButton("1");
        button2=new JButton("2");
        button3=new JButton("3");
        button4=new JButton("4");
        button5=new JButton("5");
```

Class `Calculator` (part 1 of 2)

```
        button6=new JButton("6");
        button7=new JButton("7");
        button8=new JButton("8");
        button9=new JButton("9");
        buttonPoint = new JButton(".");
        buttonEqual=new JButton("=");

     // Create the panel with the GridLayout
     // that will contain 12 buttons - 10 numeric
     // ones, and  buttons with the point and the
     // equal sign
        p1 = new JPanel();
        GridLayout gl =new GridLayout(4,3);
        p1.setLayout(gl);
     //  Add window controls to the panel p1
        p1.add(button1);
        p1.add(button2);
        p1.add(button3);
        p1.add(button4);
        p1.add(button5);
        p1.add(button6);
        p1.add(button7);
        p1.add(button8);
        p1.add(button9);
        p1.add(button0);
        p1.add(buttonPoint);
        p1.add(buttonEqual);

     // Add the panel p1 to the center area
     // of the window
        windowContent.add("Center",p1);
     //Create the frame and set its content pane
        JFrame frame = new JFrame("Calculator");
        frame.setContentPane(windowContent);
     // set the size of the window to be big enough
     // to accomodate all controls
        frame.pack();
     // Finally, display the window
        frame.setVisible(true);
        }

        public static void main(String[] args) {
              Calculator calc = new Calculator();
        }
}
```

Class `Calculator` (part 2 of 2)

Box Layout

Class `java.swing.BoxLayout` allows multiple window components to be laid out either horizontally (along the `X`-axis) or vertically (along the `Y`-axis). Unlike with the `FlowLayout` manager, when the window with the `BoxLayout` is resized, its controls are not getting wrapped up. With `BoxLayout`, window controls can have different sizes (this is not allowed in the `GridLayout`).

The next two lines of code set a box layout with vertical alignment in a `JPanel`.

```
JPanel p1= new JPanel();
setLayout(new BoxLayout(p1, BoxLayout.Y_AXIS));
```

To make this code shorter, I do not declare a variable to store a reference to the object `BoxLayout`, but rather create an instance of this object and immediately pass it to the method `setLayout()` as an argument.

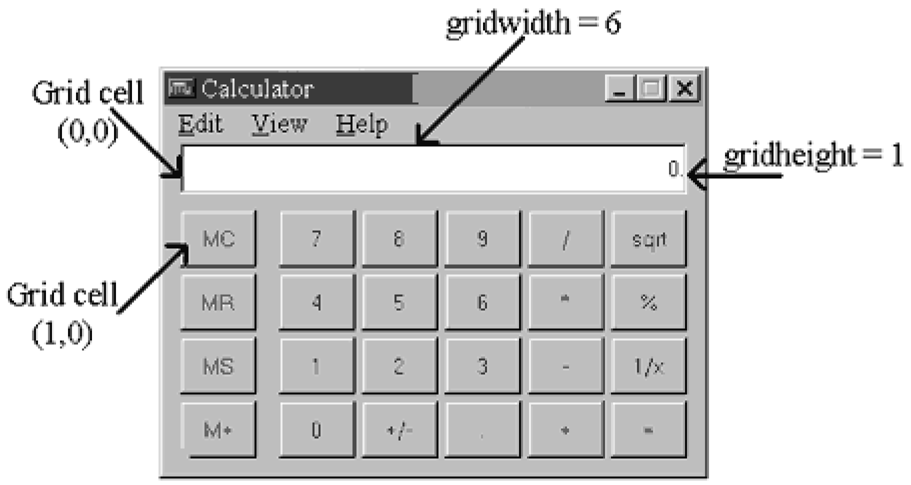
Grid Bag Layout

In this section I'll show you yet another way of creating the calculator window using `java.awt.GridBagLayout` manager instead of combining layouts and panels.

Our calculator has rows and columns, but in a grid layout, all of the components must have the same size. This does not work for our calculator because there is a text field on the top that is as wide as tree numeric buttons.

The `GridBagLayout` is an advanced grid, that allows you to have a grid with cells of different sizes. Class `GridBagLayout` works together with another class called `GridBagConstraints`. Constrains is nothing else but attributes of the cell, and you have to set them for each cell separately. All constraints for a cell have to be set *before* placing a component in the cell. For example, one of the constraint's attributes is called `gridwidth`. It allows you to make a cell as wide as several other cells.

When working with the grid layout you should create an instance of the constraint object first, and then set the values to its properties. After this is done, you can add the component to the cell in your container.

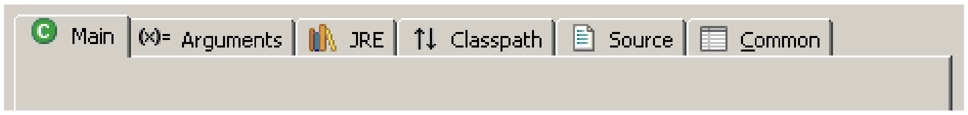


The next code sample is heavily sprinkled with comments to help you understand how to use `GridBagLayout`.

```
// Set the GridBagLayout for the window's content pane
 GridBagLayout gb = new GridBagLayout();
 this.setLayout(gb);

// Create an instance of the GridBagConstraints
// You'll have to repeat these lines for each component
// that you'd like to add to the grid cell
 GridBagConstraints constr = new GridBagConstraints();

//setting constraints for the Calculator's displayField:

// x coordinate in the grid
 constr.x=0;
// y coordinate in the grid
 constr.y=0;

// this cell has the same height as other cells
 constr.gridheight =1;

// this cell is as wide as 6 other ones
 constr.gridwidth= 6;

// fill all space in the cell
 constr.fill= constr.BOTH;
// proportion of horizontal space  taken by  this
// component
 constr.weightx = 1.0;

// proportion of  vertical space taken by  this component
 constr.weighty = 1.0;
// position of the component within the cell
 constr.anchor=constr.CENTER;

 displayField = new JTextField();
// set constrains for this field
 gb.setConstraints(displayField,constr);

// add the text field to the window
 windowContent.add(displayField);
```

Card Layout

Think of a deck of cards laying on top of each other, where you can only see the top card. The `java.awt.CardLayout` manager can be used if you need to create a component that looks like a tab folder.



When you click on a tab, the content of the screen changes. In fact, all of the panels needed for this screen are already pre-loaded and lay on top of each other. When the user clicks on a tab, the program just “brings this card" on top and makes the rest of the cards invisible.

Most likely you won’t use this layout, because the Swing library includes a better component for windows with tabs. This component is called `JTabbedPane`.

Can I Create Windows Without Using Layouts?

Sure you can! You may set screen coordinates of each component while adding them to the window. In this case, your class has to explicitly state that it won’t use any layout manager. Java has a special keyword `null` that actually means “has no value”. We’ll use this keyword quite often in the future, and in the following example it means that there is no layout manager:

```
windowContent.setLayout(null);
```

But if you do this, your code has to assign the coordinates of the left upper corner, the width, and the height of each window component. The next example shows how you can set a button’s width to 40 pixels, height to 20, and plae it 100 pixels to the right and 200 pixels down from the top left corner of the window:

```
JButton myButton = new Button("New Game");
myButton.setBounds(100,200,40,20);
```

Window Components

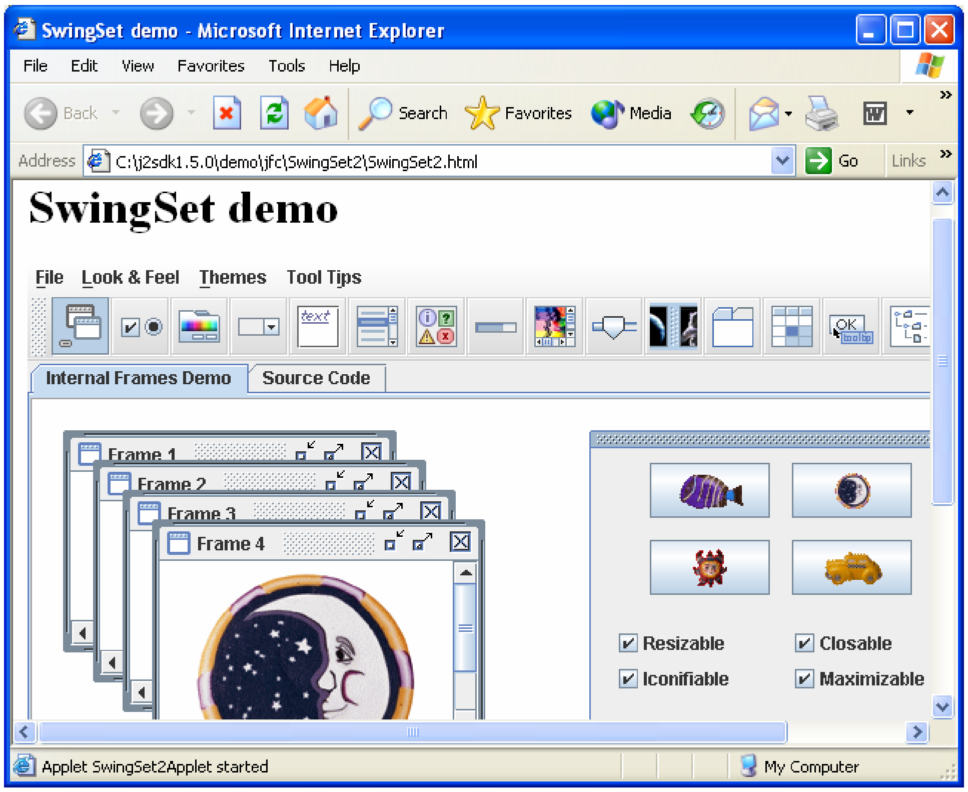
I’m not going to describe all Swing components in this book, but you can find references to Swing online tutorial in the section *Additional Reading*. This tutorial has detail explanations of all

Swing components. Our calculators use only `JButton`, `JLabel` and `JTextField`, and here's the list of what else is available:

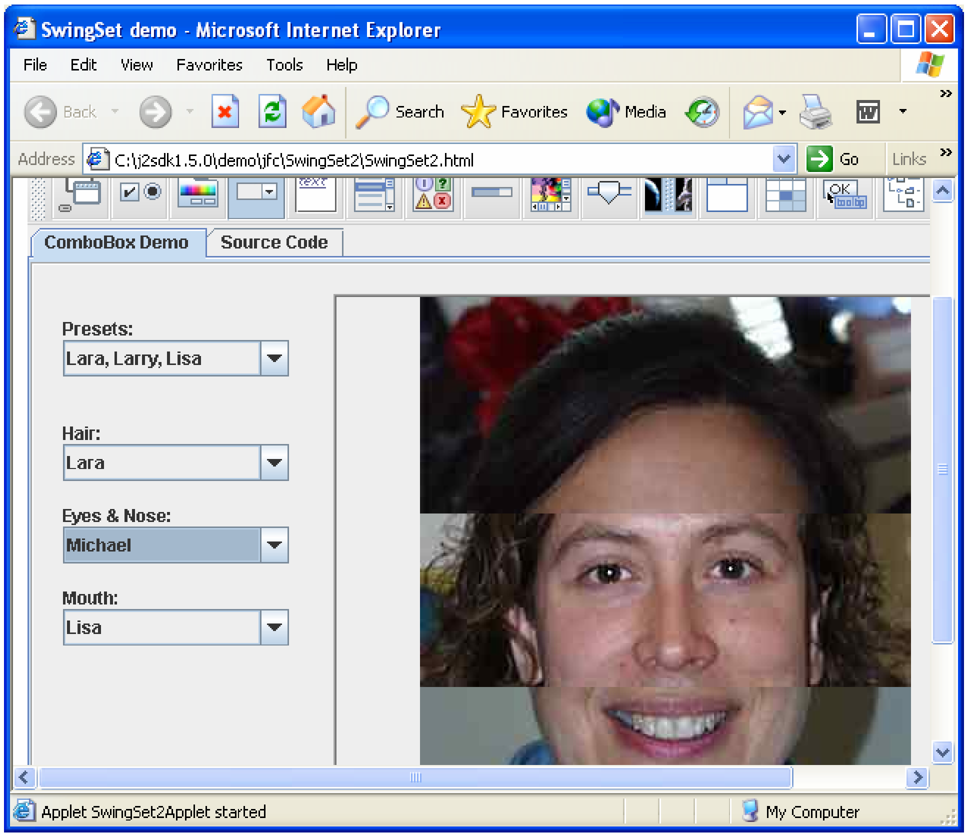
- ✓ `JButton`
- ✓ `JLabel`
- ✓ `JCheckBox`
- ✓ `JRadioButton`
- ✓ `JToggleButton`
- ✓ `JScrollPane`
- ✓ `JSpinner`
- ✓ `JTextField`
- ✓ `JTextArea`
- ✓ `JPasswordField`
- ✓ `JFormattedTextField`
- ✓ `JEditorPane`
- ✓ `JScrollBar`
- ✓ `JSlider`
- ✓ `JProgressBar`
- ✓ `JComboBox`
- ✓ `JList`
- ✓ `JTabbedPane`
- ✓ `JTable`
- ✓ `JToolTip`
- ✓ `JTree`
- ✓ `JViewPort`
- ✓ `ImageIcon`

You can also create menus (`JMenu` and `JPopupMenu`), popup windows, frames inside other frames (`JInternalFrame`), use the standard-looking windows (`JFileChooser`, `JColorChooser` and `JOptionPane`).

Java comes with an excellent demo application that shows all available Swing components in action. It's located in the J2SDK folder under `demo\jfc\SwingSet2`. Just open the file SwingSet2.html, and you'll see a screen similar to the next one.


SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\j2sdk1.5.0\demo\jfc\SwingSet2\SwingSet2.html Go Links
SwingSet demo
File Look & Feel Themes Tool Tips
Internal Frames Demo
Source Code
Frame 1
Frame 2
Frame 3
Frame 4
Resizable
Closable
Iconifiable
Maximizable
Applet SwingSet2Applet started
My Computer

Click on any image on the toolbar to see how this particular Swing component works. You can also find Java code that was used to create each window by selecting the tab *Source Code*. For example, if you click on the fourth icon from the left (so-called *combobox*), you'll see a window that looks like this:

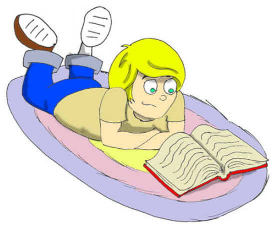

SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\j2sdk1.5.0\demo\jfc\SwingSet2\SwingSet2.html
Go Links
ComboBox Demo
Source Code
Presets:
Lara, Larry, Lisa
Hair:
Lara
Eyes & Nose:
Michael
Mouth:
Lisa
Applet SwingSet2Applet started
My Computer

Swing has so many different components to make your windows pretty!

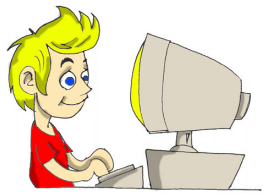
In this chapter we were creating creating Swing components simply by typing the code without using any special tools. But there are tools that allow you to select a component from a toolbar and drop it on the window. These tools will automatically generate proper Java code for Swing components. One of the free Graphic User Interface (GUI) designers that allow easy creation of Swing and SWT components is called *jigloo* from CloudGarden, and you can find a reference to a Web page of this product in the section *Additional Reading*.

In the next chapter you'll learn how a window can respond to the user's actions.

Additional Reading

	1.Swing Tutorial: http://java.sun.com/docs/books/tutorial/uiswing/ 2. Class `JFormattedTextField`: http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/JFormattedTextField.html 3.SWT tutorial and articles: http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-swt-home/SWT_Resources.html 4.Jigloo GUI builder: http://www.cloudgarden.com/jigloo/index.html

Practice



1.Modify the class `Calculator.java` to add the buttons +, -, /, and *. Add these buttons to the panel p2, and place the panel in the East area of the content pane.

2. Read about the class `JFormattedTextField` on the web and change the code of the calculator to use this class instead of the `JTextField`. The goal is to create a right-aligned field like real calculators have.

Practice for Smarty Pants

Modify the class `Calculator.java` to keep all numeric buttons in the 10-element array declared as follows:

```
Buttons[] numButtons= new Buttons[10];
```

Replace 10 lines that start from

```
button0=new JButton("0");
```

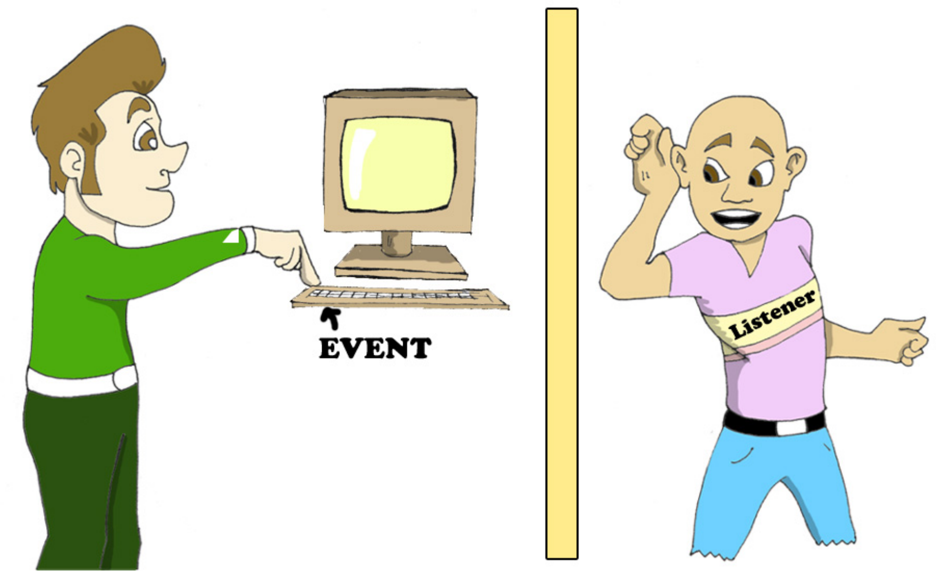
with a loop that creates the buttons and store them in this array.
Hint: peek into the code of the Tic-Tac-Toe game in Chapter 7.

Chapter 6. Window Events

Various events may happen to a running program: a user clicks on a button in a window, the Web browser decides to re-paint the window, and so on. I'm sure you've tried to click on the buttons of our calculator from Chapter 5, but these buttons were not ready to respond to your actions yet.

Each window component can process a number of events, or as we say, *listen* to these events. Your program has to register window components with Java classes called *listeners*. You should make components listen to only those events they are interested in. For example, when a person moves the mouse cursor over the calculator button, it's not important where exactly the mouse pointer was when the person pressed the button as long as it was on the button's surface. That's why you do not need to register the button with the `MouseMotionListener`. On the other hand, this listener is handy for all kinds of drawing programs.

Calculator's buttons should register themselves with the `ActionListener` that can process button-click events. All these listeners are special Java classes called *interfaces*.



Interfaces

Most of the classes define methods that perform various actions, for example *will react to button clicks*, *will react to mouse movements*, and so on. A combination of such actions is called a *class behavior*.

Interfaces are special classes that just name a set of particular actions without writing actual code that implements these actions, for example:

```
interface MouseMotionListener {
    void mouseDragged(MouseEvent e);
    void mouseMoved(MouseEvent e);
}
```

As you can see, the methods `mouseDragged()` and `mouseMoved()` do not have any code – they are just declared in the interface called `MouseMotionListener`. But if your class needs to react when the mouse is being moved or dragged, it has *to implement* this interface. The word *implements* means that this class will definitely include methods that might have been declared in this interface, for example:

```
import java.awt.event.MouseMotionListener;

class myDrawingPad implements MouseMotionListener{

      // your code that can draw goes here

      mouseDragged(MouseEvent e){
      // your code that has to be performed when
      // the mouse is being dragged goes here
      }
      mouseMoved(MouseEvent e){
      // your code that has to be performed when
      // the mouse is being moved goes here
      }
}
```

You may be wondering, why even bother creating interfaces without writing code there? The reason is that once the interface is created, it could be reused by many classes. For example, when other classes (or JVM itself) see that the class `myDrawingPad` implements the interface `MouseMotionListener`, they know for sure that this class will definitely have methods `mouseDragged`() and `mouseMoved()`. Every time when a user moves the mouse, JVM will call the method `mouseMoved()`and execute the code that you wrote there. Imagine if a programmer Joe decides to name such method `mouseMoved()`, Mary calls it `movedMouse()`, and Pete prefers `mouseCrawling()`? In this case the JVM would be confused and wouldn't know which method to call on your class to signal about the mouse movement.

A Java class can implement multiple interfaces, for example it may need to respond to mouse movements and to a button click:

```
class myDrawingProgram implements
                     MouseMotionListener, ActionListener
{

   //You have to write the code for each method that
   // has been defined in both interfaces here

}
```

After getting comfortable with the interfaces that come with Java, you'll be able to create your own interfaces, but this is an advanced topic and let's not even go there at this time.

Action Listener

Let's get back to our calculator. If you've completed assignments from the previous chapter, the visual part is done. Now we'll create another class-listener that will perform some actions when the user clicks on one of the buttons. Actually, we could have added the code processing click events to the class `Calculator.java`, but good programmers always keep visual and processing parts in separate classes.

We'll name a second class `CalculatorEngine`, and it must implement a `java.awt.ActionListener` interface that declares only one method - `actionPerformed(ActionEvent)`. JVM calls this method on the class that implements this interface whenever the person clicks on the button.

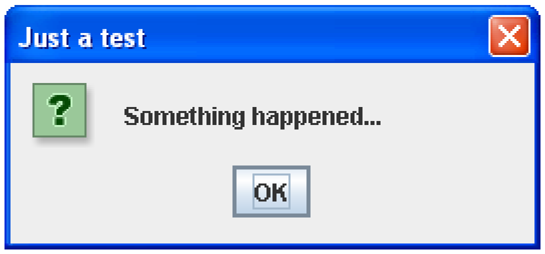
Please create the following simple class:

```
import java.awt.event.ActionListener;
public class CalculatorEngine implements ActionListener {

}
```

If you try to compile this class (or just save it in Eclipse), you'll get an error message saying that the class must implement the method `actionPerformed(ActionEvent e)`. Let's fix this error:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class CalculatorEngine implements
                                     ActionListener {

   public void actionPerformed(ActionEvent e){
      // An empty method is also allowed here,
      // even though nothing is going to happen when
      // the JVM calls it
   }
}
```

The next version of this class will display a *message box* from the method `actionPerformed()`. You can display any messages using the class `JOptionPane` and its method `showConfirmDialog()`. For example, the class `CalculatorEngine` displays the following message box:



There are different versions of the method `showConfirmDialog()`, and we are going to use the one with four arguments. In the code below, `null` means that this message box does not have the parent window, the second argument contains the title of the message box, then goes the message itself, and the fourth argument allows you to select a button(s) to be included in the box (PLAIN_MESSAGE means that only a single button OK will be displayed in the message box).

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JOptionPane;
public class CalculatorEngine implements
ActionListener {

   public void actionPerformed(ActionEvent e){
      JOptionPane.showConfirmDialog(null,
                  "Something happened...",
                  "Just a test",
                  JOptionPane.PLAIN_MESSAGE);
   }
}
```

In the next section I'll explain you how to compile and run the next version of our calculator that will display the *Something Happened* message box.

Registering Components with ActionListeneter

Who and when will call the code that we wrote in the method `actionPerformed()`? The JVM itself will call this method *if you register* (or link) the calculator's buttons with the class `CalculatorEngine`! Just add the following two lines at the end of the constructor of the class `Calculator.java` to register the button zero with our action listener:

```
CalculatorEngine calcEngine = new CalculatorEngine();
button0.addActionListener(calcEngine);
```

From now on, every time when the user clicks on the `button0`, JVM calls the method `actionPerformed()` on the object `CalculatorEngine`. Compile and run the class `Calculator` now, and click on the button zero – it'll display the *Something happened* message box! Other buttons remain silent because they are not registered yet with our action listener. Keep adding similar lines to bring all buttons to life:

```
button1.addActionListener(calcEngine);
button2.addActionListener(calcEngine);
button3.addActionListener(calcEngine);
button4.addActionListener(calcEngine);
…
```

What's the Source of an Event?

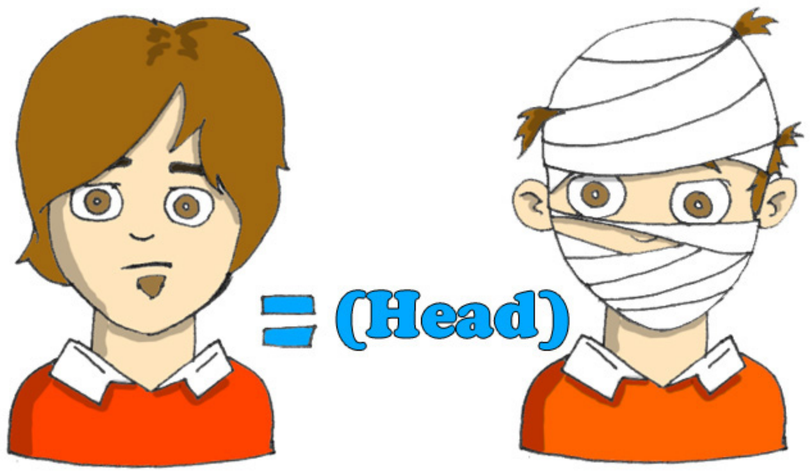
The next step is to make our listener a little smarter – it'll display different message boxes, depending on which button was pressed. When an *action event* happens, JVM calls the method `actionPerformed(ActionEvent)` on your listener class, and it provides a valuable information about the event in the argument `ActionEvent`. You can get this information by calling appropriate methods on this object.

Casting

In the next example we are finding out which button has been pressed by calling the method `getSource()` of the class `ActionEvent` – the variable `e` is a reference to this object that lives somewhere in computer's memory. But according to Java documentation, this method returns the source of the event as an instance of type `Object`, which is a superclass of all Java classes including window components. It's done this way to make a universal method that works for all components. But we know for sure, that in our window only buttons can possibly be the reason of the action event! That's why we *cast* the returned `Object` to the shape of a `JButton` by placing a type `(JButton)` in parentheses in front of the method call:

```
JButton clickedButton =  (JButton) evt.getSource();
```

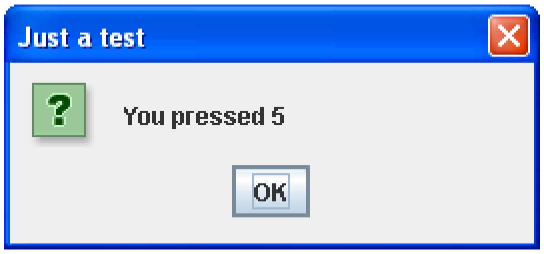
We declare a variable of type `JButton` on the left of the equal sign, and even though the method `getSource()` returns the data of type `Object`, we say to JVM: *Don't worry, I know for sure that I'm getting an instance of a `JButton`.*



Only after performing casting from `Object` to `JButton` we are allowed to call the method `getSource()` that belongs to a class `JButton`.

```java
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JOptionPane;
import javax.swing.JButton;
public class CalculatorEngine implements ActionListener
{
   public void actionPerformed(ActionEvent e){
      // Get the source of this action
     JButton clickedButton=(JButton) e.getSource();
      // Get the button's label
      String clickedButtonLabel =
                           clickedButton.getText();

      // Concatenate the button's label
      // to the text of the message box
      JOptionPane.showConfirmDialog(null,
               "You pressed " + clickedButtonLabel,
               "Just a test",
               JOptionPane.PLAIN_MESSAGE);
   }
}
```

For example, if you press the button five, you'll see the following message box:


Just a test

You pressed 5

But what if window events are produced not only by buttons, but by some other components as well? We do not want to cast every object to JButton! For these cases you should use a special Java operator called instanceof to perform the proper casting. The next example first checks what type of object caused the event, and then performs casting to either JButton or JTextField:

```
public void actionPerformed(ActionEvent evt){

 JTextField myDisplayField=null;
 JButton clickedButton=null;

 Object eventSource = evt.getSource();

 if (eventSource instanceof JButton){
      clickedButton = (JButton) eventSource;
 }else if (eventSource instanceof JTextField){
      myDisplayField = (JTextField)eventSource;
 }
}
```

Our calculator has to execute different portions of the code for each button, and the next code snippet shows you how to do this.

```
public void actionPerformed(ActionEvent e){

     Object src = e.getSource();

     if (src == buttonPlus){
      // Code that adds numbers goes here
     } else if (src == buttonMinus){
      // Code that subtracts numbers goes here
     }else if (src == buttonDivide){
      // Code that divides numbers goes here
     } else if (src == buttonMultiply){
      // Code that multiplies numbers goes here
     }

}
```

How to Pass Data Between Classes

Actually, when you press a numeric button on the real calculator, it does not show a message box, but rather displays the number in the text field on top. Here's the a new challenge – we need to be able to reach the attribute displayField of the class Calculator from the method actionPerformed() of the class CalculatorEngine. This can be done if we define in the class CalculatorEngine a variable that will store a *reference to the instance* of the object Calculator.

We are going to declare a constructor in the next version of the class CalculatorEngine. This constructor will have one argument of type `Calculator`. Don't be surprised, method arguments can have data types of the classes that were created by you!

JVM executes the constructor of the `CalculatorEngine` instance during creation of this class in memory. The class `Calculator` instantiates the `CalculatorEngine`, and passes to the engine's constructor *the reference to itself*:

```
CalculatorEngine calcEngine = new CalculatorEngine(this);
```

This reference contains a location of the calculator's instance in memory. The engine's constructor stores this value in the member variable `parent`, and eventually will use it in the method `actionPerformed()` to access the calculator's display field.

```
parent.displayField.getText();
…
parent.displayField.setText(dispFieldText +
                                              clickedButtonLabel);
```

These two lines where taken from the next code sample.

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JButton;

public class CalculatorEngine implements ActionListener {

 Calculator parent; // a reference to the Calculator

 // Constructor stores the reference to the
 // Calculator window in  the member variable parent
 CalculatorEngine(Calculator parent){
   this.parent = parent;
 }

 public void actionPerformed(ActionEvent e){
   // Get the source of this action
   JButton clickedButton =  (JButton) e.getSource();

   // Get the existing text from the Calculator's
   // display field
   String dispFieldText = parent.displayField.getText();

   // Get the button's label
   String clickedButtonLabel = clickedButton.getText();

   parent.displayField.setText(dispFieldText +
                                          clickedButtonLabel);
 }
}
```

When you declare a variable for storing a reference to the instance of a particular class, this variable has to have either the data type of this class or of one of its superclasses.

Every class in Java is inherited from the class `Object`, and if the class `Fish` is a subclass of a `Pet`, each of these lines is correct:

```
Fish myFish   = new Fish();
Pet  myFish   = new Fish();
Object myFish = new Fish()
```

Finishing Calculator

Let's come up with some rules (*an algorithm*) of how our calculator should work:

1. The user enters all the digits of the first number.

2. If the user hits one of the action buttons +, -, / or * , then store the first number and selected action in member variables, and erase the number from the display text field.

3. The user enters the second number and hits the button *equals* .

4. Convert the `String` value from the text field into a numeric type double to be able to store large numbers with a decimal point. Perform selected action using this value and the number stored in the variable from step 2.

5. Display the result from step 4 in the text field and store this value in the variable that was used in step 2.

We'll program all these actions in the class `CalculatorEngine`. While reading the code below, remember that the method `actionPerformed()` will be called after each button click and the data between these method calls will be stored in the variables `selectedAction` and `currentResult`.

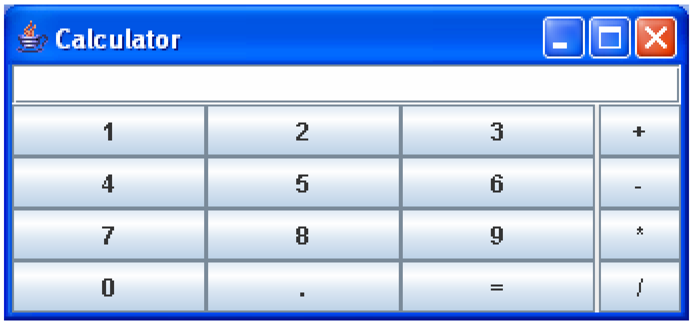
```java
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JButton;

public class CalculatorEngine
                           implements ActionListener {

 Calculator parent;  //a reference to Calculator window
 char selectedAction = ' ';  // +, -, /, or *

 double currentResult =0;

// Constructor stores the reference to the Calculator
// window in the member variable parent
 CalculatorEngine(Calculator parent){
   this.parent = parent;
 }

 public void actionPerformed(ActionEvent e){

   // Get the source of this action
   JButton clickedButton =  (JButton) e.getSource();
   String dispFieldText=parent.displayField.getText();

   double displayValue=0;

   //Get the number from  the text field
   // if it's not empty
   if (!"".equals(dispFieldText)){
     displayValue= Double.parseDouble(dispFieldText);
   }
   Object src = e.getSource();

   // For each action button memorize  selected
   // action +, -, /, or *, store the current value
   // in the currentResult, and  clean up the display
   //  field for entering the next number
```

Class `CalculatorEngine` (part 1 of 2)

```
    if (src == parent.buttonPlus){
       selectedAction = '+';
       currentResult=displayValue;
       parent.displayField.setText("");
    } else if (src == parent.buttonMinus){
       selectedAction = '-';
       currentResult=displayValue;
       parent.displayField.setText("");
     }else if (src == parent.buttonDivide){
       selectedAction = '/';
       currentResult=displayValue;
       parent.displayField.setText("");
     } else if (src == parent.buttonMultiply){
       selectedAction = '*';
       currentResult=displayValue;
       parent.displayField.setText("");
     } else if (src == parent.buttonEqual){
     // Perform the calculations based on selectedAction
     // update the value of the variable currentResult
     // and display the result
        if (selectedAction=='+'){
          currentResult+=displayValue;
       // Convert the result to String by concatenating
       // to an empty string and display it
          parent.displayField.setText(""+currentResult);
        }else if (selectedAction=='-'){
          currentResult -=displayValue;
          parent.displayField.setText(""+currentResult);
        }else if (selectedAction=='/'){
          currentResult /=displayValue;
          parent.displayField.setText(""+currentResult);
        }else if (selectedAction=='*'){
          currentResult*=displayValue;
          parent.displayField.setText(""+currentResult);
        }
     } else{
       // For all numeric buttons append the button's
       // label to the text field
       String clickedButtonLabel=
                                clickedButton.getText();
       parent.displayField.setText(dispFieldText +
                                     clickedButtonLabel);
     }
    }
 }
```

Class `CalculatorEngine` (part 2 of 2)

The final version of the calculator window will look like this:



The class `Calculator` performs the following steps:

1. Create and displays all window components.
2. Create an instance the event listener `CalculatorEngine`.
3. Pass to the engine a reference to the itself .
4. Registers with this listener all components that can generate events.

Here's the final version of the class `Calculator`:

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.BorderLayout;

public class Calculator {
      // Declare and instantiate window components
      JButton button0=new JButton("0");
      JButton button1=new JButton("1");
      JButton button2=new JButton("2");
      JButton button3=new JButton("3");
      JButton button4=new JButton("4");
      JButton button5=new JButton("5");
      JButton button6=new JButton("6");
      JButton button7=new JButton("7");
      JButton button8=new JButton("8");
      JButton button9=new JButton("9");
      JButton buttonPoint = new JButton(".");
      JButton buttonEqual=new JButton("=");
      JButton buttonPlus=new JButton("+");
      JButton buttonMinus=new JButton("-");
```

`Class Calculator (part 1 of 3)`

```
    JButton buttonDivide=new JButton("/");
    JButton buttonMultiply=new JButton("*");
    JPanel windowContent = new JPanel();
    JTextField displayField = new JTextField(30);

   // Constructor
   Calculator(){
    // Set the layout manager for this panel
    BorderLayout bl = new BorderLayout();
    windowContent.setLayout(bl);

    // Add the display field to the top od the window
    windowContent.add("North",displayField);


    // Create the panel with the GridLayout
    // that will contain 12 buttons - 10 numeric ones, and
    // buttons with the point and the equal sign

       JPanel p1 = new JPanel();
       GridLayout gl =new GridLayout(4,3);
       p1.setLayout(gl);

       p1.add(button1);
       p1.add(button2);
       p1.add(button3);
       p1.add(button4);
       p1.add(button5);
       p1.add(button6);
       p1.add(button7);
       p1.add(button8);
       p1.add(button9);
       p1.add(button0);
       p1.add(buttonPoint);
       p1.add(buttonEqual);

 // Add the panel p1 to the center area of the window
       windowContent.add("Center",p1);
       // Create the panel with the GridLayout
       // that will contain 4 action buttons -
       // Plus, Minus, Divide and Multiply
       JPanel p2 = new JPanel();
       GridLayout gl2 =new GridLayout(4,1);
       p2.setLayout(gl2);
       p2.add(buttonPlus);
       p2.add(buttonMinus);
       p2.add(buttonMultiply);
```

Class Calculator (part 2 of 3)

```
      p2.add(buttonDivide);

  // Add the panel p2 to the east area of the window
      windowContent.add("East",p2);

  // Create the frame and add the content pane to it
      JFrame frame = new JFrame("Calculator");
      frame.setContentPane(windowContent);

  // set the size of the window to be big enough to
  // accomodate all window controls
      frame.pack();

   // Display the window
     frame.setVisible(true);

   // Instantiate the event listener and
   // register each button with it
     CalculatorEngine calcEngine = new
                               CalculatorEngine(this);

      button0.addActionListener(calcEngine);
      button1.addActionListener(calcEngine);
      button2.addActionListener(calcEngine);
      button3.addActionListener(calcEngine);
      button4.addActionListener(calcEngine);
      button5.addActionListener(calcEngine);
      button6.addActionListener(calcEngine);
      button7.addActionListener(calcEngine);
      button8.addActionListener(calcEngine);
      button9.addActionListener(calcEngine);

      buttonPoint.addActionListener(calcEngine);
      buttonPlus.addActionListener(calcEngine);
      buttonMinus.addActionListener(calcEngine);
      buttonDivide.addActionListener(calcEngine);

      buttonMultiply.addActionListener(calcEngine);
            buttonEqual.addActionListener(calcEngine);
        }

        public static void main(String[] args) {
            // Instantiate the class Calculator
            Calculator calc = new Calculator();
        }
      }
```

Class Calculator (part 3 of 3)

Now compile the project and run the class `Calculator`. It works almost the same as the real world calculators.

Congratulations! This is your first program that can be used by many people – give it as a gift to your friends.

For better understanding of how this program works, I recommend you to get familiar with *debugging* of programs. Please read about debugger in Appendix B, and then come back again.

Some Other Event Listeners

These are some other Java listeners from the package `java.awt` that are good to know:

- Focus listener will send a signal to your class when a component gains or loses *focus*. For example, we say that the text field has focus, if it has a blinking cursor.
- Item listener reacts on selection of items in a list or a combobox (dropdown box).
- Key listener responds to keyboard buttons.
- Mouse listener responds when mouse is clicked, or it enters/leaves a component's area on the window.
- Mouse movement listener tells you if the mouse is being moved or dragged. *To drag* means moving the mouse while pressing its button.
- Window listener gives you a chance to catch the moments when the user opens, closes, minimizes or activates the window.

In the next table you'll see the name of the listener interfaces, and the methods that these interfaces declare.

Interface	**Methods to implement**

`FocusListener`	`focusGained(FocusEvent)`
	`focusLost(FocusEvent)`

`ItemListener`	`itemStateChanged(ItemEvent)`

`KeyListener`	`keyPressed(KeyEvent)`
	`keyReleased(KeyEvent)`
	`keyTyped(KeyEvent)`

`MouseListener`	`mouseClicked(MouseEvent)`
	`mouseEntered(MouseEvent)`
	`mouseExited(MouseEvent)`
	`mousePressed(MouseEvent)`
	`mouseReleased(MouseEvent)`

`MouseMotionListener`	`mouseDragged(MouseEvent)`
	`mouseMoved(MouseEvent)`

`WindowListener`	`windowActivated (WindowEvent)`
	`windowClosed(WindowEvent)`
	`windowClosing(WindowEvent)`
	`windowDeactivated (WindowEvent)`
	`windowDeiconified(WindowEvent)`
	`windowIconified(WindowEvent)`
	`windowOpened(WindowEvent)`

For example, the `FocusListener` interface declares two methods: `focusGained()` and `focusLost()`. This means that even if your class is interested only in processing of the events when a particular field gains the focus, you also must include the empty method `focusLost()`. This may be annoying, and Java provides special *adapter classes* for each listener to make event processing easier.

How to Use Adapters

Let's say you need to save some data on the disk when the user closes the window. According to the table above, the class that implements `WindowsListener` interface has to include seven methods. This means that you'll have write code in the method `windowClosing()` and also include six empty methods.

The package `java.awt` has adapters, which are classes that have already implemented all required methods (these methods are empty inside). One of such classes is called `WindowAdapter`. You can extend the class that has to process events from `WindowAdapter` and just override the methods you are interested in, for example the method `windowClosing()`.

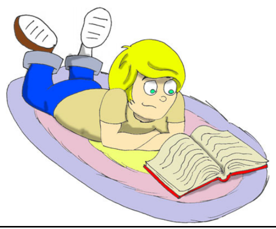
```
class MyEventProcessor extends java.awt.WindowsAdapter {
      public void windowClosing(WindowEvent e) {
      // your code that saves the data on a disk
      // goes here.
      }
}
```

The rest is easy – just register this class as an event listener in the window class:

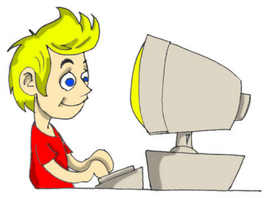
```
MyEventProcessor myListener =  new MyEventProcessor();
addWindowListener(myListener);
```

We can achieve the same result using so-called *anonymous inner classes*, but this topic is a little too complicated for this book.

Additional Reading

	Writing Event Listeners: http://java.sun.com/docs/books/tutorial/uiswing/events/

Practice



Try to divide a number by zero using our calculator - the display field shows the word `Infinity`. Modify the class `CalculatorEngine` to display a message *Can't divide by zero* if the user clicks on the button *Divide* when display field is empty.

Practice for Smarty Pants

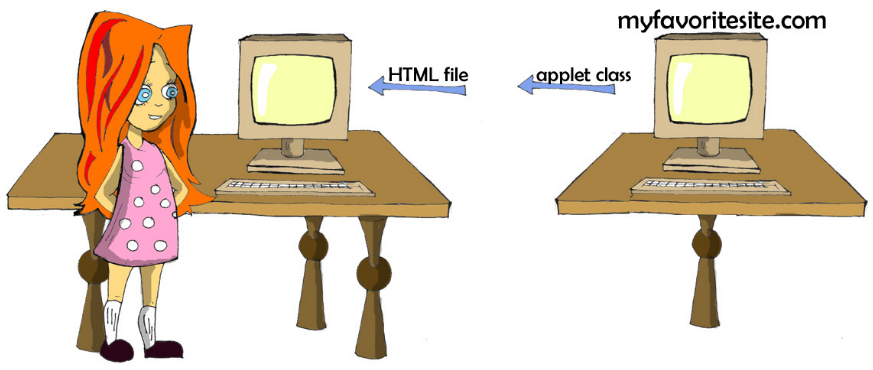
Modify the class `CalculatorEngine` to not allow entering more than one period in the number.

Hint: read about the method `indexOf()` of the class `String` to find out if the display field already has one period.

Chapter 7. The Tic-Tac-Toe Applet

When you go online to your favorite Web site, the chances are that some of these games or other programs were written in Java using so-called *applets.* These special applications live and run inside the Web browser's window. Web browsers understand a simple mark-up language called HTML, which allows you to insert special *tags* (marks) in the text files to display them nicely in the browsers. Besides the text, you can include in HTML file a special tag `<applet>` that will tell the browser where to find and how to display a Java applet.

Java applets are downloaded to your computer from the Internet as a part of a Web page, and the browser is smart enough to start its own JVM to run these applets.



In this chapter you'll learn how to create applets on your computer, and Appendix C explains how to publish your Web pages on the Internet so other people can also use them.

People browse the Internet without knowing if web pages contain Java applets or not, but they want to be sure that their computers will not be harmed by some bad guys who added a nasty applet to the page. That's why the applets were designed with the following restrictions:

- Applets can not access files on your disk unless you have a special *certificate* file on your disk that gives them such permission.

- Applets can only connect to the computer they where downloaded from.

- Applets can not start any other program located in your computer.

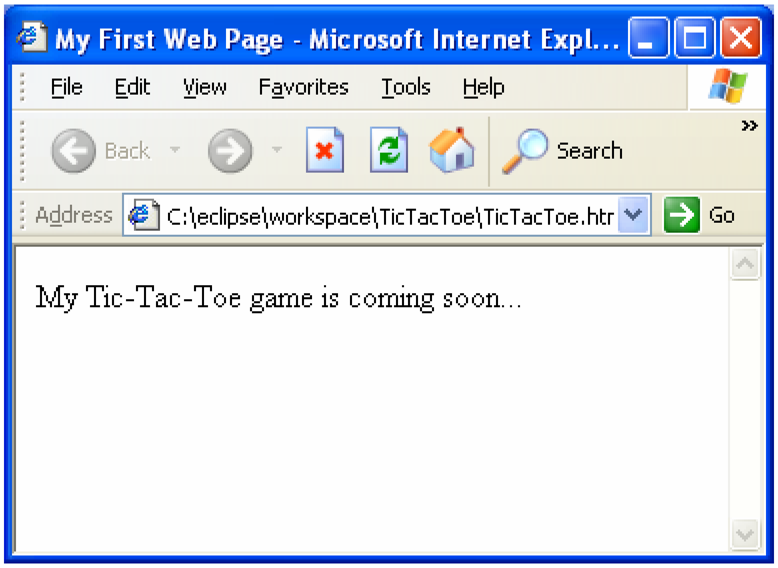
To run an applet you'll need a Java class written in a special way, an HTML text file that contains the tag `<applet>` pointing to this class, and a Web browser that supports Java. You can also test applets in Eclipse or using a special program called *appletviewer*. But before learning how to create applets, let's spend 15 minutes to get familiar with some HTML tags.

Learning HTML in 15 Minutes

Imagine for a moment that you've written and compiled the game applet called `TicTacToe`. Now you need create the HTML file with information about it. First create the text file called `TicTacToe.html` (by the way, Eclipse can create text files also). HTML files have names that end either with *.html* or *.htm*. Inside, they usually have the sections *header* and *body*. Most of the HTML tags have the matching closing tags that start with a forward slash, for example `<Head>` and `</Head>`. This is how the file `TicTacToe.html` can look like:

```
<HTML>
<Head>
<Title>My First Web Page</Title>
</Head>
<BODY>
  My Tic-Tac-Toe game is coming soon…
</BODY>
</HTML>
```

You can place the tags either in the same line like we did with the tags <Title> and </Title>, or in separate lines. Open this file in your Web browser using its menus *File* and *Open*. The blue title bar of the window will read *My First Web Page…*, and inside the page you'll see the words *My Tic-Tac-Toe game is coming soon…*:

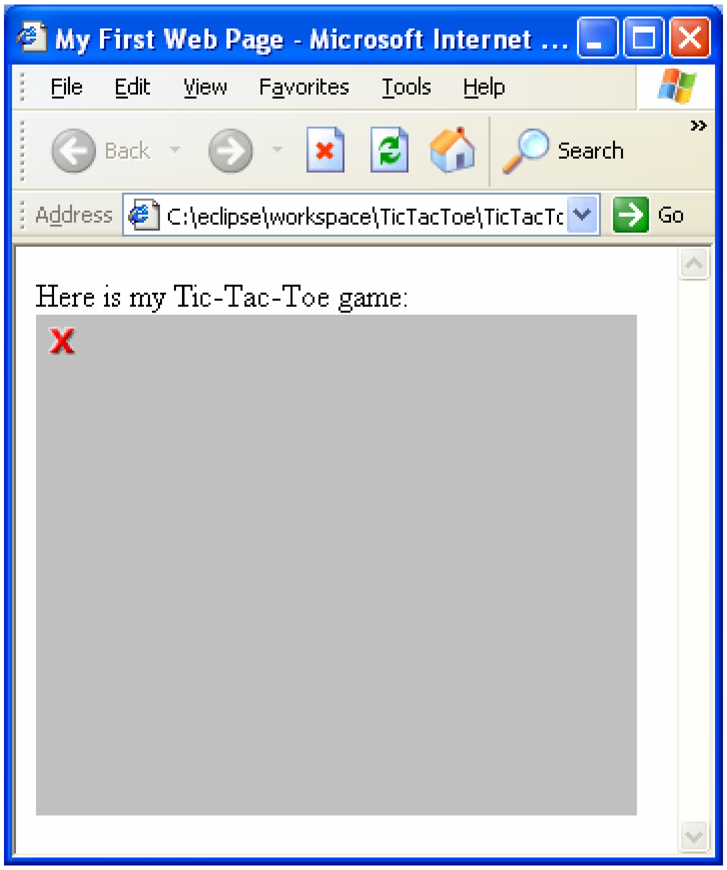

My First Web Page - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back
Search
Address
C:\eclipse\workspace\TicTacToe\TicTacToe.htr
Go

My Tic-Tac-Toe game is coming soon...

Now change this file to add the tag for the Tic-Tac-Toe applet to this file:

```
<HTML>
<BODY>
  Here is my Tic-Tac-Toe game:
  <APPLET code="TicTacToe.class" width=300
                             height=250>
  </APPLET>
</BODY>
</HTML>
```

Now the screen looks different:


My First Web Page - Microsoft Internet ...
File Edit View Favorites Tools Help
Back
Search
Address C:\eclipse\workspace\TicTacToe\TicTacTo
Go

Here is my Tic-Tac-Toe game:

No wonder, since the Web browser could not find the TicTacToe.class, it just shows a gray rectangle. We'll create this class a little later in this chapter.

HTML tags are surrounded by angle brackets, and some of the tags may have additional *attributes*. The tag `<APPLET>` in our example uses the following attributes:

- `code` - it's the name of the applet's Java class.

- `width` – has the width in *pixels* of the rectangular area on the screen that will be used by the applet. Images on the computer screen are made out of tiny dots that are called pixels.

- `height` - has the height of the area to be used by the applet.

If a Java applet consists of multiple classes, put all of them into one archive file using the *jar* program that comes with JDK. And if you do so, the attribute archive must have the name of this archive. You can read about jars in Appendix A.

Writing Applets Using AWT

Why use AWT for writing applets if the Swing library is better? Can we write applets using Swing classes? Yes we can, but there is something you need to know about.

Web browsers come with their own JVMs, which support AWT, but might not support the Swing classes that are included in your applet. Of course the users may download and install the latest JVM, and there are special HTML converters that will change the HTML file to point their browsers to this new JVM, but do you really want to ask users to do this? After your Web page is published on the Internet, you do not know who might be using it. Imagine an old guy somewhere in a desert with a 10 year old computer – he'll just leave your page instead of going through all these installation troubles. Imagine that our applet helps to sell games online, and we do not want to loose this guy – he might be our potential customer (people in deserts also have credit cards). ☺

Use AWT if you're not sure what kind of Web browsers your users have.

Actually, the other choice is to ask your users to download special *Java plugin*, and configure their browsers to use the plugin instead of JVM that came with their browser. You can read more about this option ot the following Web site: http://java.sun.com/j2se/1.5.0/docs/guide/plugin/.

How to Write AWT Applets

Java AWT applets have to be inherited from the class `java.applet.Applet`, for example:

```
class TicTacToe extends java.applet.Applet {

}
```

Unlike Java applications, applets do not need the method `main()` because the Web browser will *download* and run them as soon as they see the tag <applet> in the Web page. The browser also sends signals to applets when important events happen, for example the applet is starting, re-painting, and so on. To make sure that the applet reacts to these events, you should program special *callback methods*: `init()`, `start()`, `paint()`, stop(), and `destroy()`. The browser's JVM will call these methods in the following cases:

- `init()` is called when the applet is loaded by the browser. It's called only once, so it plays a role similar to constructors in regular Java classes.

- `start()` is called right after the `init()`. It is also called if a user returns to a Web page after visiting another page.

- `paint()` is called when the applet's window needs to be displayed or refreshed after some activity on the screen. For example, the applet is overlapped with some other window and the browser needs to repaint it.

- `stop()` is called when a user leaves the Web page containing the applet.

- `destroy()` – is called when the browser destroys the applet. You'd write code in this method only if the applet uses some other resources, for example it holds a connection to the computer it was downloaded from.

Even though you do not have to program all of these methods, each applet must have at least `init()` or `paint()`. Here's a code of the applet that displays the words *Hello World*. This applet has only one method `paint()` that receives an instance of the object `Graphics` from the browser's JVM. This object has a whole bunch of methods for painting. The next example uses the method `drawString()` to draw the text *Hello World.*

```
public class HelloApplet extends java.applet.Applet {
      public void paint(java.awt.Graphics graphics) {
      graphics.drawString("Hello World!", 70, 40);
      }
}
```

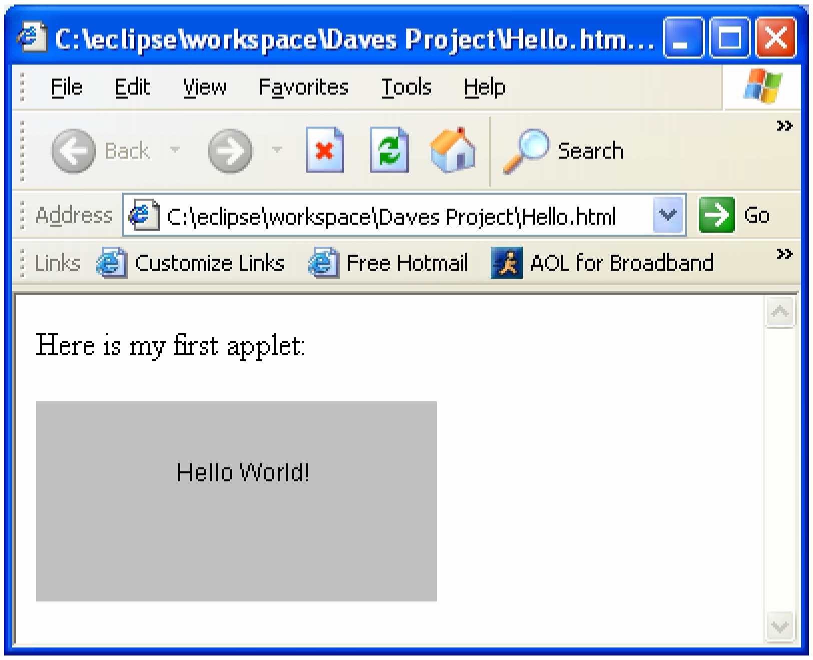
Create this class in Eclipse. Then in the *Run* window select *Java Applet* in the top left corner, press the button *New*, and enter `HelloApplet` in the field *Applet Class*.

To test this applet in the Web browser, create the file `Hello.html` in the same folder where you applet class is located:

```
<HTML>
 <BODY>
  Here is my first applet:<P>
 <APPLET code="HelloApplet.class" width=200 height=100>
 </APPLET>
 </BODY>
</HTML>
```

Now start you Web browser and open the file `Hello.html` using the menus *File* and *Open*.

The screen should look like this:

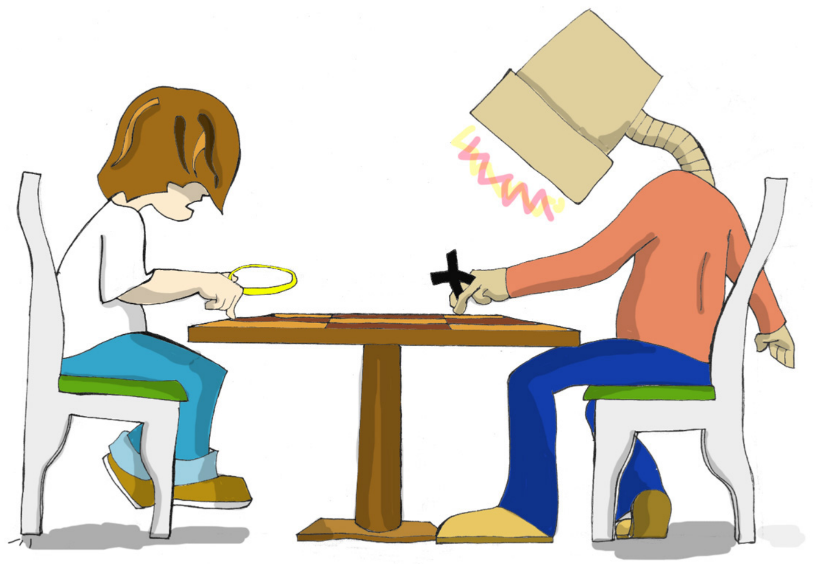

C:\eclipse\workspace\Daves Project\Hello.htm...
File Edit View Favorites Tools Help
Back
Search
Address
C:\eclipse\workspace\Daves Project\Hello.html
Go
Links
Customize Links
Free Hotmail
AOL for Broadband

Here is my first applet:

Hello World!

Do you think that after this simple example we are ready for writing a game program? You bet! Just fasten your seat belts…

Writing a Tic-Tac-Toe Game



The Strategy

Every game uses some algorithm – a set of rules or a strategy that have to be applied depending on the player's actions. The algorithms for the same game can be simple or very complicated. When you hear that the world chess champion Gary Kasparov plays against a computer, he actually plays against a program. Teams of experts are trying to invent sophisticated algorithms to beat him. The tic-tac-toe game can also be programmed using different strategies, and we'll be using the simple one:

1. We are going to use a 3x3 board.
2. The user will play with the symbol *X*, and the computer will use *O*.
3. The winner must have a full row, column, or a diagonal with the same symbols.
4. After each move, the program has to check if there is a winner.
5. If there is a winner, the winning combination has to be highlighted and the game has to end.
6. The game should also end if there is no more empty squares left.
7. The player has to press the button *New Game* to play again.
8. When computer makes a decision where to put the next *O*, it has to try to find a row, a column or a

diagonal that has already two *O*'s, and put the third row accordingly.

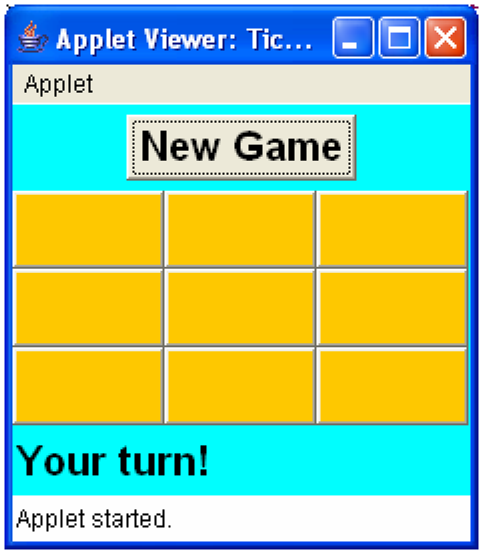
9. If there is no two *O*'s, the computer has to try to find the two *X*'s and place an *O* to block the person's winning move.
10. If no winning or blocking move was found, the computer has to try to occupy the central square, or pick the next empty square *randomly*.

The Code

I'll give you just a short description of the program here because there is lot of comments in the applet's code that will help you to understand how it works.

The applet will use a `BorderLayout` manager, and the `North` portion of the window will have the button *New Game*.

The center part will show nine buttons representing squares, and the `South` part will display messages:



Your turn!

Applet started.

All window components will be created in the applet's method `init()`. All events will be processed by the `ActionListener` in the method `actionPerformed()`. The method `lookForWinner()` is called after every move to check if the game is over.

Rules 8, 9, and 10 from our strategy are coded in the method `computerMove()` that might need to generate a *random number*. This is done using the Java class `Math` and its method `random()`.

You'll also find somewhat unusual syntax when several method calls are perform in one *expression*, for example:

```
if(squares[0].getLabel().equals(squares[1].getLabel())){…}
```

This line makes the code shorter because it actually performs that same actions that could have been done in the following lines:

```
String label0 = squares[0].getLabel();
String label1 = squares[1].getLabel();
if(label0.equals(label1)){…}
```

In complex expressions Java evaluates the code in parentheses before doing any other calculations. The short version of this code gets the result of the expression in parentheses first, and immediately uses it as an argument for the method `equals()`, which is applied to the result of the first call to `getLabel()`.

Even though the game code occupies several pages, it should not be to difficult to understand. Just read all program comments.

```
/**
 * A tic-tac-toe game on a 3x3 board
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class TicTacToe extends Applet implements
                                        ActionListener{
Button squares[];
Button newGameButton;
Label score;
int emptySquaresLeft=9;
/**
 * init method is the applet's constructor
 */
 public void init(){
 //Set the applet's layout manager, font and color
  this.setLayout(new BorderLayout());
  this.setBackground(Color.CYAN);

  // Change the applet's font to be bold
  // of size 20 points
  Font appletFont=new Font("Monospased",Font.BOLD, 20);
  this.setFont(appletFont);
  // Create the button New Game and register it
  // with the action listener
  newGameButton=new Button("New Game");
  newGameButton.addActionListener(this);

  Panel topPanel=new Panel();
  topPanel.add(newGameButton);
```

Class `TicTacToe` (part 1 of 7)

```
      this.add(topPanel,"North");

      Panel centerPanel=new Panel();
      centerPanel.setLayout(new GridLayout(3,3));
      this.add(centerPanel,"Center");

      score=new Label("Your turn!");
      this.add(score,"South");

   // create an array to hold references to 9 buttons
      squares=new Button[9];

  // Instantiate the buttons, store the references
  // to them in the array, register them with the
  // listeners, paint them in orange and add to panel
      for(int i=0;i<9;i++){
                  squares[i]=new Button();
                  squares[i].addActionListener(this);

      squares[i].setBackground(Color.ORANGE);
                  centerPanel.add(squares[i]);
      }
 }
 /**
 * This method will process all action events
 * @param ActionEvent object
 */
 public void actionPerformed(ActionEvent e) {

  Button theButton = (Button) e.getSource();
  // Is this a New Game button?
  if (theButton ==newGameButton){
    for(int i=0;i<9;i++){
                  squares[i].setEnabled(true);
                  squares[i].setLabel("");

      squares[i].setBackground(Color.ORANGE);
    }

   emptySquaresLeft=9;
   score.setText("Your turn!");
   newGameButton.setEnabled(false);

   return;  // exit the method here
 }

 String winner = "";
 // Is this one of the squares?
 for ( int i=0; i<9; i++ ) {
  if (  theButton == squares[i] ) {
    squares[i].setLabel("X");
    winner = lookForWinner();

    if(!"".equals(winner)){
       endTheGame();
    } else {
       computerMove();
       winner = lookForWinner();
       if ( !"".equals(winner)){
          endTheGame();
       }
    }
    break;
  }
 } // end for

 if ( winner.equals("X") ) {
      score.setText("You won!");
 } else if (winner.equals("O")){
      score.setText("You lost!");
 } else if (winner.equals("T")){
      score.setText("It's a tie!");
 }
} // end actionPerformed

/**
 *  This method is called after every move to see
 *  if we have a winner. It checks every row, column
 *  and diagonal to find out three squares with the
 *  same label (other than blank)
 *  @return "X", "O", "T" for tie or "" for no winner
 */
 String lookForWinner() {

  String theWinner = "";
  emptySquaresLeft--;

  if (emptySquaresLeft==0){
   return "T";  // it's a tie
  }
```

```
// Check the row 1 - array elements 0,1,2
if (!squares[0].getLabel().equals("") &&
 squares[0].getLabel().equals(squares[1].getLabel()) &&
 squares[0].getLabel().equals(squares[2].getLabel())) {

         theWinner = squares[0].getLabel();
         highlightWinner(0,1,2);
// Check the row 2  - array elements 3,4,5
} else if (!squares[3].getLabel().equals("")  &&
 squares[3].getLabel().equals(squares[4].getLabel()) &&
 squares[3].getLabel().equals(squares[5].getLabel())) {

         theWinner = squares[3].getLabel();
         highlightWinner(3,4,5);
// Check the row 3 -  - array elements 6,7,8
} else if ( ! squares[6].getLabel().equals("") &&
 squares[6].getLabel().equals(squares[7].getLabel()) &&
 squares[6].getLabel().equals(squares[8].getLabel())) {

         theWinner = squares[6].getLabel();
         highlightWinner(6,7,8);
// Check the column 1  - array elements 0,3,6
} else if ( ! squares[0].getLabel().equals("") &&
 squares[0].getLabel().equals(squares[3].getLabel()) &&
 squares[0].getLabel().equals(squares[6].getLabel())) {

         theWinner = squares[0].getLabel();
         highlightWinner(0,3,6);
// Check the column 2 - array elements 1,4,7
} else if ( ! squares[1].getLabel().equals("") &&
 squares[1].getLabel().equals(squares[4].getLabel()) &&
 squares[1].getLabel().equals(squares[7].getLabel())) {

         theWinner = squares[1].getLabel();
         highlightWinner(1,4,7);
// Check the column 3 - array elements 2,5,8
} else if (  ! squares[2].getLabel().equals("") &&
 squares[2].getLabel().equals(squares[5].getLabel()) &&
 squares[2].getLabel().equals(squares[8].getLabel())) {
         theWinner = squares[2].getLabel();
         highlightWinner(2,5,8);
// Check the first diagonal  - array elements 0,4,8
} else if ( ! squares[0].getLabel().equals("") &&
 squares[0].getLabel().equals(squares[4].getLabel()) &&
 squares[0].getLabel().equals(squares[8].getLabel())) {

         theWinner = squares[0].getLabel();
         highlightWinner(0,4,8);
// Check the second diagonal - array elements 2,4,6
} else if ( ! squares[2].getLabel().equals("") &&
 squares[2].getLabel().equals(squares[4].getLabel()) &&
 squares[2].getLabel().equals(squares[6].getLabel())) {

         theWinner = squares[2].getLabel();
         highlightWinner(2,4,6);
 }
 return theWinner;
}
/**
 * This method applies a set of rules to find
 * the best computer's move. If a good move
 * can't be found, it picks a random square.
 */
 void computerMove() {
   int selectedSquare;
      // Computer first tries to find an empty
      // square next the two squares with "O" to win
     selectedSquare = findEmptySquare("O");
    // if can't find two "O", at least try to stop the
    // opponent from making 3 in a row by placing
    // "O" next to 2 "X".
    if ( selectedSquare == -1 )
        selectedSquare =  findEmptySquare("X");
    }
    // if the selectedSquare is still -1, at least
    // try to occupy the central square
    if ( (selectedSquare == -1)
            &&(squares[4].getLabel().equals("")) ){
       selectedSquare=4;
    }
    // no luck with the central either...
    // just get a random square
    if ( selectedSquare == -1 ){
       selectedSquare = getRandomSquare();
    }
    squares[selectedSquare].setLabel("O");
 }
```

```java
/**
 * This method checks every row, column and diagonal
 * to see if there are two squares with the same label
 * and an empty square.
 * @param   give X - for user, and O for computer
 * @return  the number of the empty square to use,
 *          or the negative 1 could not find 2 square
 *          with the same label
 */
 int findEmptySquare(String player) {

   int weight[] = new int[9];

   for ( int i = 0; i < 9; i++ ) {
        if ( squares[i].getLabel().equals("O") )
            weight[i] = -1;
        else if ( squares[i].getLabel().equals("X") )
            weight[i] = 1;
        else
            weight[i] = 0;
   }

   int twoWeights = player.equals("O") ? -2 : 2;

    // See if row 1 has the same 2 squares and a blank
   if ( weight[0] + weight[1] + weight[2] == twoWeights
) {
        if ( weight[0] == 0 )
            return 0;
        else if ( weight[1] == 0 )
            return 1;
        else
            return 2;
    }
   // See if row 2 has the same 2 squares and a blank
  if (weight[3] +weight[4] + weight[5] == twoWeights) {
        if ( weight[3] == 0 )
            return 3;
        else if ( weight[4] == 0 )
            return 4;
        else
            return 5;
    }
```

```
// See if row 3 has the same 2 squares and a blank
if (weight[6] + weight[7] +weight[8] == twoWeights ) {
      if ( weight[6] == 0 )
          return 6;
      else if ( weight[7] == 0 )
          return 7;
      else
          return 8;
}
 // See if column 1 has the same 2 squares and a blank
if (weight[0] + weight[3] + weight[6] == twoWeights) {
      if ( weight[0] == 0 )
          return 0;
      else if ( weight[3] == 0 )
          return 3;
      else
          return 6;
}
// See if column 2 has the same 2 squares and a blank
if (weight[1] +weight[4] + weight[7] == twoWeights ) {
      if ( weight[1] == 0 )
          return 1;
      else if ( weight[4] == 0 )
          return 4;
      else
          return 7;
}
 // See if column 3 has the same 2 squares and a blank
if (weight[2] + weight[5] + weight[8] == twoWeights ){
      if ( weight[2] == 0 )
          return 2;
      else if ( weight[5] == 0 )
          return 5;
      else
          return 8;
}
//See if diagonal 1 has the same 2 squares and a blank
if (weight[0] + weight[4] + weight[8] == twoWeights ){
      if ( weight[0] == 0 )
          return 0;
      else if ( weight[4] == 0 )
          return 4;
      else
          return 8;
}
```

```
 // See if diagonal has the same 2 squares and a blank
 if (weight[2] + weight[4] + weight[6] == twoWeights ){
      if ( weight[2] == 0 )
           return 2;
      else if ( weight[4] == 0 )
           return 4;
      else
           return 6;
 }
 // There are no two neighbors  that are the same
 return -1;
} // end of findEmptySquare()
/**
 * This method selects any empty square
 * @return a randomly selected square number
 */
 int getRandomSquare() {
  boolean gotEmptySquare = false;
  int selectedSquare = -1;

  do {
    selectedSquare = (int) (Math.random() * 9 );
    if (squares[selectedSquare].getLabel().equals("")){
       gotEmptySquare = true; // to end the loop
    }
  } while (!gotEmptySquare );

    return selectedSquare;
 } // end getRandomSquare()
/**
 * This method highlights the winning line.
 * @param first,second and third squares to highlight
 */
 void highlightWinner(int win1, int win2, int win3) {
    squares[win1].setBackground(Color.CYAN);
    squares[win2].setBackground(Color.CYAN);
    squares[win3].setBackground(Color.CYAN);
 }
// Disables squares and enable New Game button
void endTheGame(){
    newGameButton.setEnabled(true);
    for(int i=0;i<9;i++){
            squares[i].setEnabled(false);
    }
  }
} // end of class
```

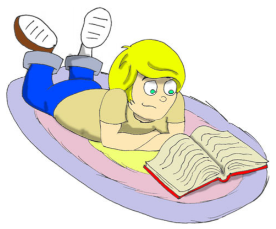
Class `TicTacToe` (part 7 of 7)

Congratulations! You've completed your first game in Java.

You can run this applet either directly from Eclipse, or by opening the file `TicTacToe.html` that we created in the beginning of this chapter, just copy HTML file and the `TicTacToe.class` in the same folder. Our class `TicTacToe` has a small bug – you might not even notice it, but I'm sure it'll be gone after you complete the second assignment below.

Our `TicTacToe` class uses a simple strategy because our goal is just learn how to program, but if you'd like to improve this game, learn so-called *minimax strategy* that allows to select the best move for the computer. Description of the minimax strategy does not belong to this book, but is available online.

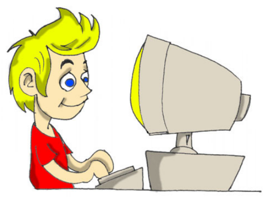
Additional Reading



Java Applets:
http://java.sun.com/docs/books/tutorial/applet/

Java Class `Math`
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html

Practice



1. Add to the top panel of the class `TicTacToe` two labels to count wins and loses. Declare two class variables for this and increment the corresponding variable each time you have a winner or a loser. The score has to be refreshed right after the program prints a message *You won* or *You lost*.

2. Our program allows click on the square that already has an *X* or *O*. This is a bug! The program continues as if you've made a valid move. Modify the code to ignore clicks on such squares

3. Add the method `main()` to the class `TicTacToe` to allow starting the game not as an applet, but as a Java application.

Practice for Smarty Pants

1. Rewrite the `TicTacToe` to replace one-dimensional array that stores nine buttons

```
JButton squares[]
```

with two dimensional 3x3 array:

```
JButton squares[][]
```

Read about multi-dimentional arrays on the Web.

Chapter 8. Program Errors - Exceptions

Say you forget a closing curly brace in your Java code. This will result in compilation error that can be fixed easily. But there are so called *run-time errors*, when all of a sudden your program stops working properly. For example, a Java class reads a file with the game scores. What's going to happen if someone will delete this file? Will the program crash with that scary long error message, or will it stay alive displaying a user friendly message like this one: *Dear friend, for some reason I could not read the file scores.txt. Please make sure that the file exists*? You should make your programs ready for unusual situation. In many programming languages error processing depends on the good will of a programmer. But Java forces you to include error processing code, otherwise the programs will not even compile.

Run-time errors in Java are called *exceptions*, and error processing is called *exception handling*. You have to place code that may produce errors in so-called `try/catch` block. It's as if you're saying to JVM the following: *Try to read the file with scores, but if something happens, catch the error and execute the code that will deal with it*:

```
try{
  fileScores.read();
}
catch (IOException e){
  System.out.println(
 "Dear friend,I could not read the file cores.txt");
}
```

We'll learn how to work with files in Chapter 9, but at this point get familiar with a new term *I/O* or *input/output*. Read and write operations (to disk or other device) are called input/output and hence the `IOException` is a class that contains information about input/output errors.

A method *throws an exception* in case of an error. Different exceptions will be thrown for different type of errors. If the `catch` block exists in the program for this particular type of an error, it will be caught and the program will jump into the `catch` block to execute the code located there. The program will stay alive, and this exception is considered to be taken care of.

The print statement from the code above will be executed only in case of the file read error.

Reading the Stack Trace

If an unexpected exception occurs that is not handled by the program, it prints a multi-line error message on the screen. Such message is called stack trace. If your program has called several methods before it ran into a problem, the *stack trace* can help you to trace the program, and find the line that have caused the error.

Let's write a program `TestStackTrace` that divides by zero on purpose (line numbers are not the part of the code).

```
1 class TestStackTrace{
2     TestStackTrace()
3     {
4       divideByZero();
5     }
6
7     int divideByZero()
8     {
9      return 25/0;
10    }
11
12    static void main(String[]args)
13    {
14       new TestStackTrace();
15    }
16 }
```

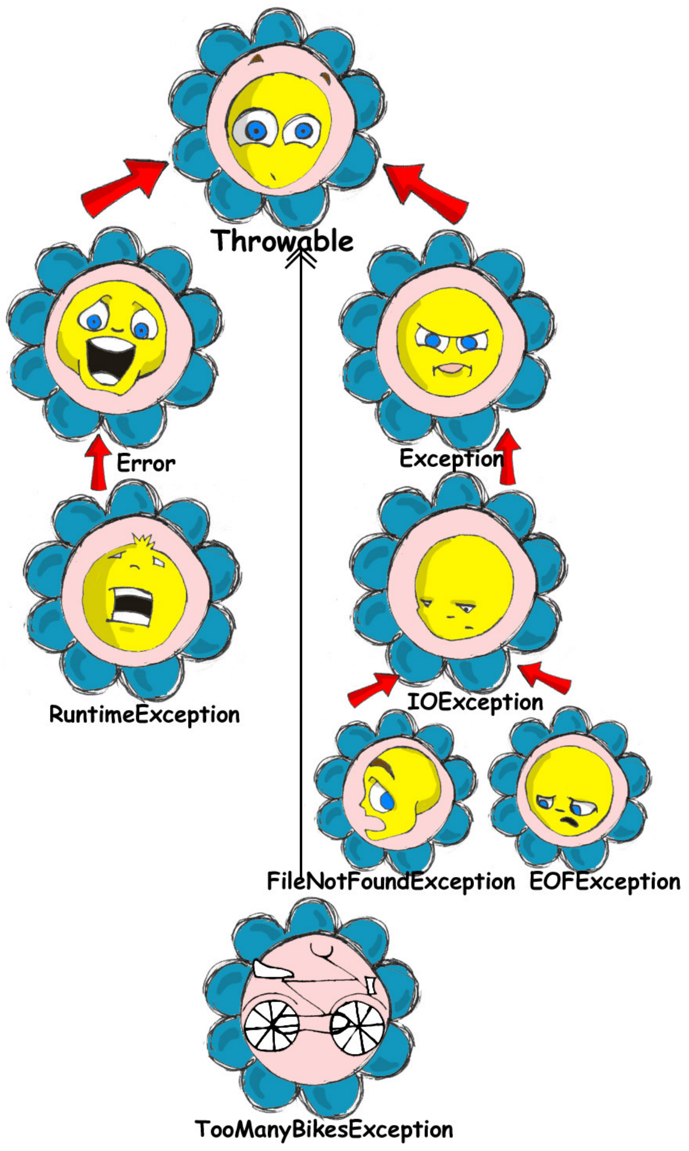
The output of this program shows the sequence of method calls that were made up to the moment when the run-time error had happened. Start reading this output from the last line going up.

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
   at TestStackTrace.divideByZero(TestStackTrace.java:9)
   at TestStackTrace.<init>(TestStackTrace.java:4)
   at TestStackTrace.main(TestStackTrace.java:14)
```

This means that the program started in the method `main()`, then went to `init()` which is a constructor, and then called the method `divideByZero()`. The numbers 14, 4 and 9 show in which lines of the program these methods were called. After that, an `ArithmeticException` was thrown –the line number nine tried to divide by zero.

Genealogical Tree of Exceptions

Exceptions in Java are also classes, and some of them are shown in the following inheritance tree:



Subclasses of the class `Exception` are called *checked exceptions* and you must process them in your code.

Subclasses of the class `Error` are fatal JVM errors and the running program can't handle them.

The `TooManyBikesException` is an example of exception that can be created by a programmer.

How a programmer is supposed to know in advance if some Java method may throw an exception and that a `try/catch` block should be used? Not to worry, if you call a method that may throw an exception, Java compiler will print an error message similar to this one:

```
"ScoreReader.java":  unreported exception: java.io.IOException;
must be caught or declared to be thrown at line 57
```

Of course you are welcome to read Java documentation that describes exceptions that may be thrown by any particular method. The rest of this chapter will explain how to deal with these exceptions.

Try/Catch Block

There are five Java keywords that can be used for error handling: `try`, `catch`, `finally`, `throw`, and `throws`.

After one `try` block you may put several `catch` blocks, if you believe that more than one error may happen. For example, when a program tries to read a file, the file may not be there, and you'll get the `FileNotFoundException`, or the file is there, but the code keeps reading the file after reaching the end of file – this generates `EOFException`. The next code fragment will print messages in plain English if the program can't find a file with game scores or reached the end of the file. For any other read errors it'll print the message *Problem reading file* and a technical description of the error.

```
public void getScores(){
 try{
    fileScores.read();
    System.out.println("Scores loaded successfully");
 }catch(FileNotFoundException e){
     System.out.println("Can not find file Scores");
 }catch(EOFException e1){
      System.out.println("Reached end of file");
 }catch(IOException e2){
      System.out.println("Problem reading  file " +
                                          e2.getMessage());
 }
}
```

If the method `read()` fails, the program jumps over the line `println()` and tries to land in the `catch` block that matches the error. If it finds such block, the appropriate `println()` will be executed, but if the matching `catch` block is not found, the method `getScores()` will re-throw this exception to its caller.

If you write several `catch` blocks, you may need to place them in a particular order if these exceptions are inherited from each other. For example, since the `EOFException` is a subclass of the

IOException, you have to put the catch block for the subclass first. If you would put the catch for IOException first, the program would never reach the FileNotFound or EOFException, because the first catch would intercept them.

Lazybones would program the method getScores() just like this:

```
public void getScores(){
 try{
  fileScores.read();
 }catch(Exception e){
  System.out.println("Problem reading  file "+
                                          e.getMessage());
   }
}
```

This is an example of a bad style of Java coding. When you write a program, always remember that someone else may read it, and you don't want to be ashamed of your code.

Catch blocks receive an instance of the object Exception that contains a short explanation of a problem, and its method getMessage() will return this info. Sometimes, if the description of an error is not clear, try the method toString() instead:

```
catch(Exception e){
 System.out.println("Problem reading file "+ e.toString());
}
```

If you need more detailed information about the exception, use the method printStackTrace(). It will print the sequence of method calls that lead to this exception similar to an example from the section *Reading Stack Trace*.

Let's try to "kill" the calculator program from Chapter 6. Run the class Calculator and enter from the keyboard the charactes *abc*. Press any of the action buttons, and you'll see on the console screen something like this:

```
java.lang.NumberFormatException: For input string: "abc"
      at
java.lang.NumberFormatException.forInputString(NumberFormatExcept
ion.java:48)
      at
java.lang.FloatingDecimal.readJavaFormatString(FloatingDecimal.ja
va:1213)
      at java.lang.Double.parseDouble(Double.java:202)
      at
CalculatorEngine.actionPerformed(CalculatorEngine.java:27)
      at
javax.swing.AbstractButton.fireActionPerformed(AbstractButton.jav
a:1764)
```

This was an example of a non-handled exception. The class `CalculatorEngine` has the following line in its method `actionPerformed()`:

```
displayValue= Double.parseDouble(dispFieldText);
```

If the variable `dispFieldTest` has not a numeric value, the method `parseDouble()` will not be anle to convert it to the `double` data type and will throw a `NumberFormatException`.

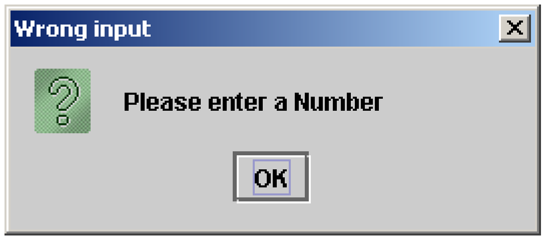
Let's handle this exception and display an error message that will explain the problem to the user. The line with `parseDouble()` has to be placed in a `try/catch` block, and Eclipse will help you with this. Highlight this line and right-click on it with the mouse. In the popup menu select the items *Source* and *Surround with try/catch block*. Voila! The code is changed:

```
try {
   displayValue= Double.parseDouble(dispFieldText);
} catch (NumberFormatException e1) {
   // TODO Auto-generated catch block
   e1.printStackTrace();
}
```

Replace the `printStackTrace()` line with the following:

```
javax.swing.JOptionPane.showConfirmDialog(null,
   "Please enter a Number", "Wrong input",
   javax.swing.JOptionPane.PLAIN_MESSAGE);
return;
```

We've got rid of the scary stack trace error messages, and displayed a simple to understand message *Please enter a Number*:


Wrong input

Please enter a Number

Now the `NumberFormatException` is handled.

The keyword `throws`

In some cases, it makes more sense to handle the exception not in the method where it happened, but in the method's caller.
In such cases the method signature has to declare (warn) that it may throw a particular exception. This is done using a special

keyword throws. Let's use the same example that reads a file. Since the method read() may throw an IOException, you should either handle or declare it. In the next example we are going to declare that the method getAllScores() may throw an IOException:

```
class MySuperGame{

  void getAllScores() throws IOException{
   // …
   // Do not use try/catch  if you are
   // not handling exceptions in this method
   file.read();
  }

  public static void main(String[] args){
    MySuperGame msg = new MySuperGame();
    System.out.println("List of Scores");

    try{
     // Since the  getAllScores()declares exception,
     // we handle  it over here
       msg.getAllScores();

    }catch(IOException e){
       System.out.println(
       "Sorry, the list of scores is not available");
    }
 }
```

Since we are not even trying to catch exceptions here, the IOException will be *propagated* from the getAllScores() to its caller - the method main(). Now the main method has to handle this exception.

The Keyword finally

Any code within a try/catch block can end in one of the following ways:

- The code inside the try block successfully ended and the program continues.
- The code inside the try block runs into a return statement and the method is exited.
- The code in the try block throws an exception and control goes to the matching catch block, which either handles the error

and the method execution continues, or it re-throws the exception to the caller of this method.

If there is a piece of code that must be executed no matter what, put it under the keyword `finally`:

```
try{
      file.read();
}catch(Exception e){
      printStackTrace();
}finally{
  // the code that must always be executed
  // goes here, for example file.close();
}
```

The code above has to close the file regardless of success or failure of the read operation. Usually, you can find the code that releases some computer resources in the block `finally`, for example, disconnection from a network or file closing.

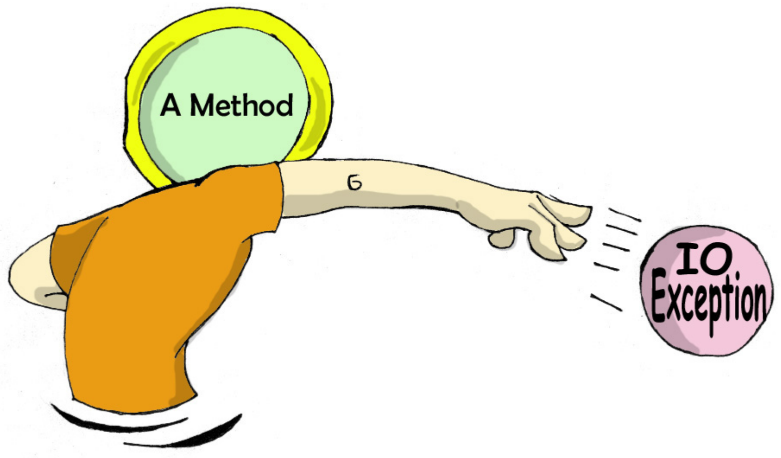
If you are not planning to handle exceptions in the current method, they will be propagated to the caller. In this case, you can use the `finally` even without a `catch` block:

```
void myMethod () throws IOException{
 try{
   // your code that reads a file goes here
 }
 finally{
   // your code that closes the file  goes here
 }
}
```

The Keyword `throw`

If an exception has happened in a method, but you believe that the caller should handle it, just re-throw it to the method's caller. Sometimes, you might want to catch one exception but re-throw another one with a different description of the error like in the code snippet below.

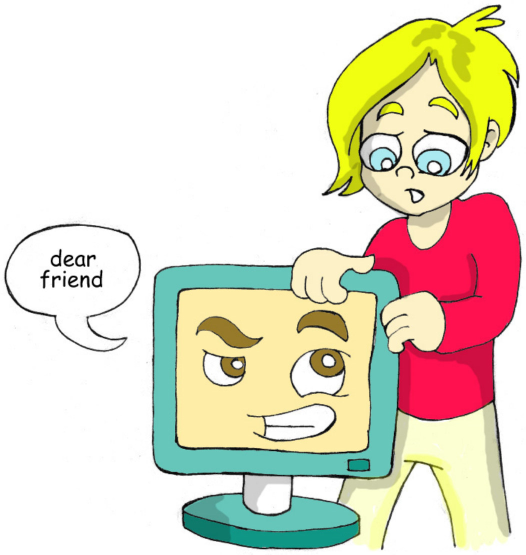
The statement `throw` is used to throw Java objects. The object that a program throws must be *throwable*. This means that you can only throw objects that are direct or indirect subclasses of the class `Throwable`, and all Java exceptions are its subclasses.



The next code fragment shows how the method `getAllScores()` catches an `IOException` and creates a new `Exception` object with a more friendly description of the error, and re-throws it to the method `main()`. Now the method `main()` won't compile unless you put the line that calls `getAllScores()` in the `try/catch` block, because this method may throw an `Exception` and it should be either handled or re-thrown again. The method `main()` should not throw any exceptions, that's why it should handle it.

```
class ScoreList{
   // Additional code is needed to compile this class

   static void getAllScores() throws Exception{
     try{
      file.read();//this line may throw an exception
     } catch (IOException e) {
         throw new  Exception (
         "Dear Friend, the file Scores has problems");
     }
   }

  public static void main(String[] args){
    System.out.println("Scores");

      try{
             getAllScores();
      }
      catch(Exception e1){
              System.out.println(e1.getMessage());
      }
  }
}
```

In case of a file error, the main method will handle it, and the `e1.getMessage()` will return the *Dear Friend…* message.



Creating New Exceptions

Programmers could also create new exception classes that did not exist in Java before. Such classes have to derive from one of the Java exception classes. Let's say you are in business of selling bikes and need to *validate* customer orders. Different number of bikes can fit in your small truck depending on the model. For example, you can fit no more than three FireBird bikes in your truck. Create a new subclass of `Exception` called `TooManyBikesException`, and if someone tries to order more than three of these bikes, throw the this exception:

```
class TooManyBikesException extends Exception{

  // Constructor
  TooManyBikesException (){
  // Just call the constructor of the superclass
  // and pass to it the error message to display
  super("Can't ship this many bikes in one shipment.");
  }
}
```

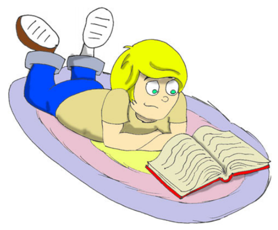
This class has only a constructor that takes the message describing this error and gives it to its superclass for storage. When some `catch` block receives this exception it can find out what exactly has happened by calling the method `getMessage()`.

Imagine that a user selects on the `OrderWindow` several bicycles of some model and hits the button *Place Order*. As you know from Chapter 6, this action will result in call to `actionPerformed()` that will check if the order can be delivered. The next code example shows how the method `checkOrder()` of this window declares that it can throw `TooManyBikesException`. If the order won't fit in the truck, this method throws the exception, the catch block intercepts it and displays an error message in the text field on the window.

```
class OrderWindow implements ActionListener{
// The code to create window components is needed here.

// The user clicked on the button Place Order
   String selectedModel = txtFieldModel.getText();
   String selectedQuantity =
                              txtFieldQuantity.getText();
   int quantity = Integer.parseInt(selectedQuantity);

  void actionPerformed(ActionEvent e){
    try{
      bikeOrder.checkOrder("FireBird", quantity);
  //the next line will be skipped in case of exception
      txtFieldOrderConfirmation.setText(
                           "Your order is complete");
   } catch(TooManyBikesException e){
    txtFieldOrderConfirmation.setText(e.getMessage());
   }
 }

 void checkOrder(String bikeModel, int quantity)
                         throws TooManyBikesException{

//Write the code that checks if the requested
//quantity of bikes of selected model will fit in the
//truck. If they won't fit, do  the following:

  throw new TooManyBikesException("Can not ship" +
      quantity + " bikes of the model " + bikeModel +
                              " in one shipment" );
 }
}
```

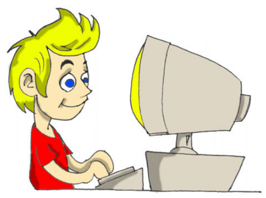
In a perfect world, every program would work properly, but realistically we have to be ready for the unexpected situations. It really helps that Java forces you to write code that is prepared for these situations.

Additional Reading



Handling Errors With Exceptions:
http://java.sun.com/docs/books/tutorial/essential/exceptions/

Practice



Create a Swing application for placing bike orders. It has to have two text fields *Bike Model* and *Quantity*, a button *Place Order*, and the label for order confirmation.

Use the code in the examples `OrderWindow` and `TooManyBikesException`. Make up several combinations of bike models and quantities that will throw an exception.

Practice for Smarty Pants

Modify the application from the previous assignment to replace the text field *Bike Model* with a dropdown list box that will contain several models, so the user can select from the list rather then type them.

You'll have to read online about the Swing component `JComboBox` and the `ItemListener` to process events when the user picks the bike model.

Chapter 9. Saving the Game Score

After a program ends it gets erased from memory. This means that all the classes, methods and variables do not exist until you run this program again. If you'd like to save some results of the program execution, they must be saved in files on a disk, tape, a memory stick, or other device that can store the data for a long time. In this chapter you'll learn how to save data on disks using Java *streams*. Basically, you open a stream between your program and a file on disk. If you need to read data from disk, it has to be an *input stream*, and if you write data on the disk, open an *output stream*. For example, if a player wins a game and you want to save the score, you can save it in a file called scores.txt using an output stream.

A program reads or writes data from/to a stream *serially* – byte after byte, character after character, etc. Since your program may use different data types like `String`, `int`, `double`, and so on, you should use an appropriate Java stream, for example a byte stream, a character stream, or a data stream.

Classes that work with file streams are located in packages `java.io.` and `java.nio`.

No matter what type of a file stream you are going to use, the following three steps should be done in your program:

- Open a stream that points at some file.
- Read or write some data from/to this stream.
- Close the stream.

Byte Streams

If you create a program that reads a file, and then displays its content on the screen, you need to know what type of data is stored in this file. On the other hand, a program that just copies files from one place to another, does not even need to know if it's an image, text or a file with music. Such program reads the original file in memory as a set of bytes, and then write them into a destination folder byte after byte using Java classes `FileInputStream` or `FileOutputStream`.

The next example shows how to use the class `FileInputStream` to read a graphic file named `abc.gif` that is located in the folder `c:\practice`. If you use a computer with Microsoft Windos, to avoid a confusion with special Java characters that start with a backslash, use double slashes in your code to separate folders and files: *c:\\practice*. This little program does not display the image, but rather prints some numbers , which is how this image is stored on a disk. Each byte has a positive integer value from 0 to 255, and the class `ByteReader` prints these values separated by a space character.

Please note that the class `ByteRader` closes the stream in the block `finally`. Never call the method `close()` inside of the `try/catch` block right after finishing reading the file, do it in the `finally` block. In case of exception during the file read, the program would jump over the crossed-out `close()` statement and the stream would not be closed! The reading ends when the method `FileInputStream.read()` returns the value of a negative one.

```
import java.io.FileInputStream;
import java.io.IOException;

public class ByteReader {

  public static void main(String[] args) {

   FileInputStream myFile = null;

   try {
      // Open a byte stream pointing at the file
       myFile = new
              FileInputStream("c:\\temp\\abc.gif");

       while (true) {
         int intValueOfByte = myFile.read();
         System.out.print(" " + intValueOfByte);

           if (intValueOfByte  == -1){
            // we've reached the end of file
            // let's exit out of the loop
              break;
           }
         } // end of while loop
         // ~~myFile.close();~~ don't do it here
     } catch (IOException e) {
              System.out.println("Could not read file: "
                                                   + e.toString());
     } finally{
         try{
              myFile.close();
         } catch (Exception e1){
                   e1.printStackTrace();
         }
         System.out.println(
                  " Finished reading the file");
       }
  }
}
```

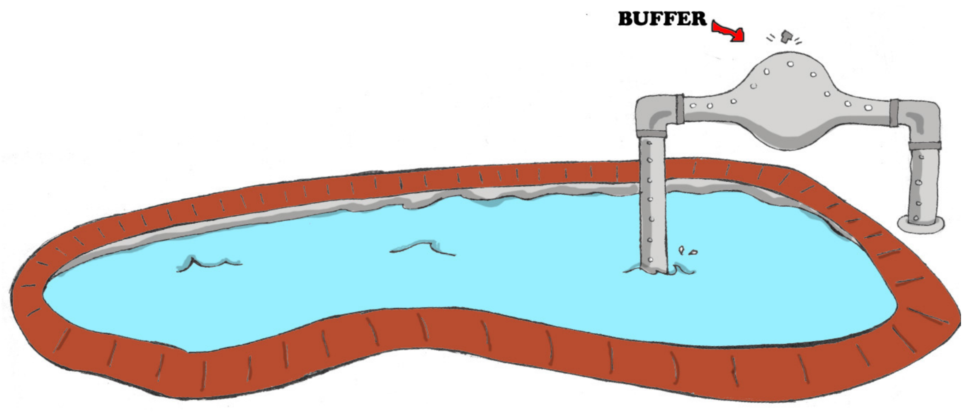
The next code fragment writes several bytes that are represented by integer numbers into a file called `xyz.dat` using the class `FileOutputStream`:

```
int somedata[]= {56,230,123,43,11,37};

  FileOutputStream myFile = null;

  try {
       // Open the file xyz.dat and save
       // there data from the array
       myFile = new  FileOutputStream("xyz.dat");
       for (int i = 0; i <some data.length; i++){
            file.write(data[i]);
       }
 } catch (IOException e) {
    System.out.println("Could not write to a  file: "+
                                         e.toString());
  } finally{
       try{
        myFile.close();
       } catch (Exception e1){
             e1.printStackTrace();
       }
  }
```

Buffered Streams

So far we were reading and writing data one byte at a time, which means that the program ByteReader will have to access disk 1000 times for reading a file of 1000 bytes. But accessing data on disks is much slower than data manipulation in memory. To minimize the number of times a program tries to access the disk, Java provides so-called buffers which are sort of "reservoirs of data".

The class BufferedInputStream helps quickly fill the memory buffer with data from the FileInputStream. A buffered stream reads a big chunk of bytes from a file in one shot into a buffer in memory, and, then the method read() gets the single bytes from the buffer a lot faster.



Your program can connect streams like a plumber connect two pipes. Let’s modify the example that reads a file. First the data is being poured from the `FileInputStream` to the `BufferedInputStream`, and then to the method `read()`:

```
FileInputStream myFile = null;
  BufferedInputStream buff =null;

  try {
      myFile = new  FileInputStream("abc.dat");
      // connect the streams
      buff = new BufferedInputStream(myFile);
      while (true) {
            int byteValue = buff.read();
            System.out.print(byteValue + " ");
            if (byteValue  == -1)
                  break;
    }
  } catch (IOException e) {
      e.printStackTrace();
  }finally{
     try{
      buff.close();
      myFile.close();
     } catch(IOException e1){
         e1.printStackTrace();
     }
  }
```

How big is the buffer? It depends on the JVM, but you can set its size and see if it makes the file reading a little faster. For example, to set the buffer size to 5000 bytes use the two-argument constructor:

```
BufferedInputStream buff =
                     new BufferedInputStream(myFile, 5000);
```

Buffered streams do not change the type of reading – they just make it faster.

The `BufferedOutputStream` works in the same fashion, but it uses the class `FileOutputStream`.

```
int somedata[]= {56,230,123,43,11,37};
  FileOutputStream myFile = null;
  BufferedOutputStream buff =null;

  try {
      myFile = new  FileOutputStream("abc.dat");
      // connect the streams
      buff = new BufferedOutputStream(myFile);
      for (int i = 0; i <somedata.length; i++){
         buff.write(somedata[i]);
      }
  } catch (IOException e) {
      e.printStackTrace();
  }finally{
       try{
            buff.flush();
            buff.close();
            myFile.close();
       } catch(IOException e1){
             e1.printStackTrace();
       }
  }
```

To make sure that all bytes from the butter are pushed out to the file stream, call the method `flush()` when the writing into a `BufferedOutputStream` is finished.

Command-Line Arguments

Our `ByteReader` program stores the name of the file abc.gif right in its code, or as programmers say, the file name is *hard-coded* in the program. This means that to create a similar program that reads the file xyz.gif, you'll have to modify the code and recompile the program, which is not nice. It would be much better to pass the name of the file from a command line, when you run the program.

You can run any Java program with *command-line arguments*, for example:

```
java ByteReader xyz.gif
```

In this example we are passing to the method main() of ByteReader just one argument - xyz.gif. If you remember, the method `main()` has an argument :

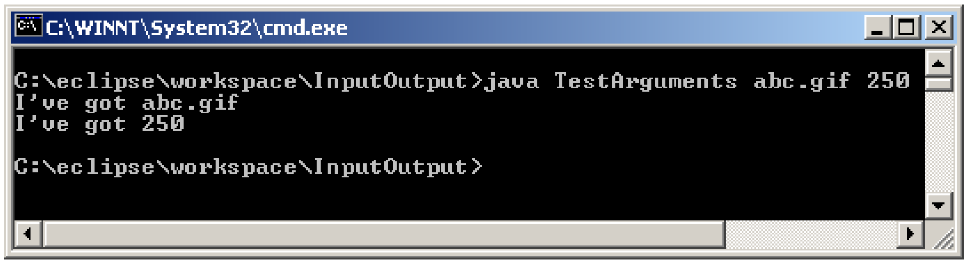
```
public static void main(String[] args) {
```

Yes, it's a `String` array that JVM passes to the main method, and if you start a program without any command line arguments, this array remains empty. Otherwise, this array will have exactly as many elements as the number of command-line arguments passed to the program.

Let's see how we can use these command line arguments in a very simple class that will just print them:

```java
public class TestArguments {

       public static void main(String[] args) {

         // How many arguments we've got?
         int numberOfArgs = args.length;

         for (int i=0; i<numberOfArgs; i++){
           System.out.println("I've got " + args[i]);
         }
       }
}
```

The next screenshot shows you what happens if you run this program with two arguments – `xyz.gif` and `250`. The value xyz.gif is placed by JVM into the element `args[0]`, and the second one goes into `args[1]`.



```
C:\eclipse\workspace\InputOutput>java TestArguments abc.gif 250
I've got abc.gif
I've got 250

C:\eclipse\workspace\InputOutput>
```

Command-line arguments are always being passed to a program as `Strings`. It's the responsibility of a program to convert the data to the appropriate data type, for example:

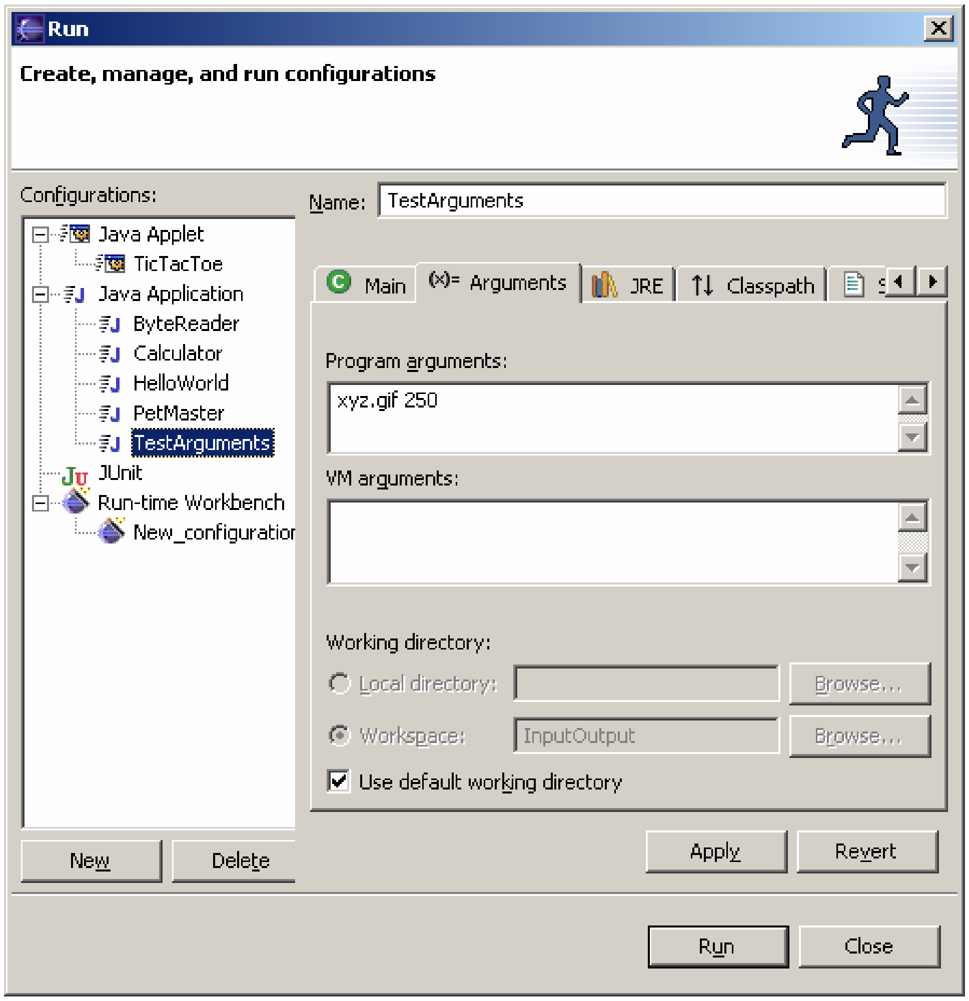
```java
int myScore = Integer.parseInt(args[1]);
```

It's always a good idea to check if the command line contains correct number of arguments. Do this right in the beginning of the method `main()`. If the program doesn't receive expected arguments, it should print a brief message about it and immediately stop by using a special method `System.exit()`:

```
public static void main(String[] args) {
 if (args.length != 2){
  System.out.println(
             "Please provide arguments, for example:");
  System.out.println("java TestArguments xyz.gif 250");

  // Exit the program
  System.exit(0);
 }
}
```

In the end of this chapter you'll have to write a program to copy files. To make this program working with any files, the names of the original and destination files have to be passed to this program as command-line arguments.

You can test your programs in Eclipse that also has a place to provide command-line arguments to each of your programs. In the *Run* window, select the tab that reads `(x)=Arguments` and enter required values in the box *Program Arguments*.



Name: TestArguments

Program arguments: xyz.gif 250

VM arguments:

Working directory:

Workspace: InputOutput

[✓] Use default working directory

VM arguments box allows you to pass parameters to your JVM. Such parameters could request more memory for your program, fine-tune performance of the JVM, etc. The section *Additional Reading* has a reference to a Web site that that describes these parameters in details.

Reading Text Files

Java uses two-byte characters to store letters, and the classes `FileReader` and `FileWriter` are handy for working with text files. These classes can read text files either one character at a time with the method `read()`, or entire lines with `readLine()`. Classes `FileReader` and `FileWriter` also have their counterparts `BufferedReader` and `BufferedWriter` that will speed up the work with files.

Thr next class `ScoreReader` reads the file scores.txt line by line, and the program ends when the method `readLine()` returns `null` which means end of file.

Use any plain text editor and create a file c:\scores.txt with the following content:

```
David 235
Brian 190
Anna  225
Zachary 160
```

Run the program `ScoreReader`, and it'll print the content of this file. Add several more lines to the file with scores and re-run the program to see that the new lines are also printed.

```
import java. io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class ScoreReader {

  public static void main(String[] args) {
      FileReader myFile = null;
      BufferedReader buff = null;

      try {
         myFile=new FileReader("c:\\scores.txt");
         buff = new BufferedReader(myFile);

         while (true) {
          // read a line from scores.txt
          String line = buff.readLine();
          // check for the end of file
          if (line == null)
              break;
          System.out.println(line);
        } // end while
      }catch (IOException e){
                e.printStackTrace();
      } finally {
            try{
               buff.close();
               myFile.close();
            }catch(IOException e1){
                    e1.printStackTrace();
            }
      }
  } // end main
}
```

If your program needs to write a text file on a disk, use one of the several overloaded methods `write()` of the class `FileWriter` . These methods will allow you to write a character, a `String` or an entire array of characters.

`FileWriter` has more than one overloaded constructor, and if you open a file for writing providing just the file name, this file will be replaced by the new one every time you run the program:

```
FileWriter fOut = new FileWriter("Scores.txt");
```

If you need to add data to an existing file, use the two-argument constructor (`true` means append mode):

```
FileWriter fOut = new FileWriter("Scores.txt", true);
```

The next class `ScoreWriter` writes three lines from the array `scores` into the file c:\scores.txt.

```
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class ScoreWriter {

      public static void main(String[] args) {

      FileWriter myFile = null;
      BufferedWriter buff = null;
      String[] scores = new String[3];

      // Populate array with scores
      scores[0] = "Mr. Smith 240";
      scores[1] = "Ms. Lee 300";
      scores[2] = "Mr. Dolittle 190";

      try {
        myFile = new
        FileWriter("c:\\scores2.txt");
        buff = new BufferedWriter(myFile);

        for (int i=0; i < scores.length; i++) {
         // write strings array into scores2.txt
         buff.write(scores[i]);

         System.out.println("Writing  " + scores[i] );
        }
       System.out.println("File writing is complete");

      }catch (IOException e){
                e.printStackTrace();
      } finally {
            try{
                buff.flush();
                buff.close();
                myFile.close();
            }catch(IOException e1){
                   e1.printStackTrace();
            }
      }
  } // end of main
}
```

Output of this program will look like this:

```
Writing  Mr. Smith 240
Writing  Ms. Lee 300
Writing  Mr. Dolittle 190
File writing is complete
```

Class `File`

Class `java.io.File` has a number of handy methods, which allow to rename a file, delete a file, check if the class exists, etc. Say your program saves some data in a file, and it needs to display a message to the user warning if such file already exists. To do this, you have to create an instance of the object `File` giving the name of the file, and then call the method `exists()`. If this method returns `true`, the file abc.txt is found and you should display a warning, otherwise there is no such file:

```
File aFile = new File("abc.txt");

if (aFile.exists()){
   // Print a message or use a JOptionPane
   // to display a warning
}
```

Constructor of the class `File` *does not actually create a file* – it just creates an instance of this object in memory that points at the actual file. If you really need to create a file, use the method `createNewFile()` instead.

Some of the useful methods of the class `File` are listed next.

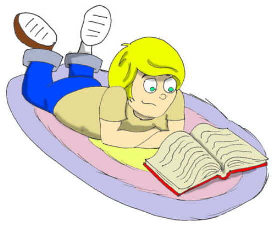
Method name	**What it does**
`createNewFile()`	Creates a new, empty file named according to the file name used during the *File* instantiation. It creates a new file only if a file with this name does not exist.
`delete()`	Deletes a file or a directory
`renameTo()`	Renames a file
`length()`	Returns the length of the file in bytes
`exists()`	Returns `true` if the file exists
`list()`	Returns an array of strings with names of files/directories located within a particular directory
`lastModified()`	Returns the time when the file was last modified
`mkDir()`	Creates a directory

The next code snippet below renames a file `customers.txt` to `customers.txt.bak`. If the `.bak` file already exists, it will be overridden.

```java
File file = new File("customers.txt");
File backup = new File("customers.txt.bak");

if (backup.exists()){
         backup.delete();
}
file.renameTo(backup);
```

Even though this chapter was about working with files located on your computer's disk, Java allows you to create streams pointing to remote machines on the computer network. Such computers can be located pretty far apart from each other. For example, NASA uses Java to control Mars rovers, and I'm sure that they just pointed their streams at Mars. ☺

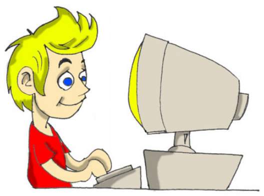
Additional Reading



1.JVM command line options
http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/java.html

2. How to use File Streams:
http://java.sun.com/docs/books/tutorial/essential/io/filestreams.html

Practice



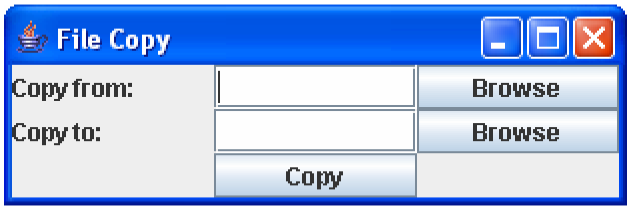
Write a file copy program called `FileCopy` by combining the code fragments from the section on byte streams.

Open two streams (input and output) and call the methods `read()` and `write()` *in the same loop*. Use command line arguments to pass the names of the original file and its new destination to the program, for example :

```
java FileCopy  c:\\temp\\scores.txt
                 c:\\backup\\scores2.txt
```

Practice for Smarty Pants

Create a Swing program that will allow users select file names to copy using the class `JFileChooser`, which creates a standard file selection window. This window should pop up when the user clicks one of the *Browse* buttons. You'll have to write a couple of lines of code to display selected file name in the appropriate text field.



When the user clicks on the button Copy, the code in the method `actionPerformed()` should copy selected file. Try to reuse the code from the previous assignment without doing copy/paste.

Chapter 10. More Java Building Blocks

We've had a chance to use quite a bit of different Java elements in the previous chapters, and even created a Tic-Tac-Toe game. But I've skipped some of the important Java elements and techniques, and it's time to catch up on them.

Working with Date and Time Values

Each computer has an internal clock. Any Java program can find out what's the current date and time, and display it in different formats, for example 06/15/2004 or June 15, 2004. Java has multiple classes that deal with dates, but two of them - `java.util.Date` and `java.text.SimpleDateFormat` will take care of most of your dat/time needs.

It's easy to create an object that stores the current system date and time up to the millisecond:

```
Date today = new Date();
System.out.println( "The date is " + today );
```

The output of these lines may look similar to this one:

```
The date is Fri Feb 27 07:18:51 EST 2004
```

The class `SimpleDateFormat` allows you to display date and time in different formats. First, you have to create an instance of this class with format that you need, and then call its method `format()` passing a `Date` object as an argument. The next program formats and prints the current date in several different formats.

```
import java.util.Date;
import java.text.SimpleDateFormat;

public class MyDateFormat {

 public static void main( String [] args ) {
  // Create an object Date
  // and print it in a default format
  Date today = new Date();
  System.out.println( "The date is " + today );

  // Format that prints dates like 02-27-04
  SimpleDateFormat sdf=
            new SimpleDateFormat("MM-dd-yy");
  String formattedDate=sdf.format(today);
  System.out.println( "The date(dd-mm-yy) is "
                    + formattedDate );

  // Format that prints dates like 27-02-2004
  sdf = new SimpleDateFormat("dd-MM-yyyy");
  formattedDate=sdf.format(today);
  System.out.println( "The date(dd-mm-yyyy) is "
                    + formattedDate );

  // Format that prints dates like Fri, Feb 27, '04
  sdf = new SimpleDateFormat("EEE, MMM d, ''yy");
  formattedDate=sdf.format(today);
  System.out.println(
  "The date(EEE, MMM d, ''yy) is "+ formattedDate);

  // Format that prints time  like 07:18:51 AM
  sdf = new SimpleDateFormat("hh:mm:ss a");
  formattedDate=sdf.format(today);
  System.out.println( "The time(hh:mm:ss a) is "
                              + formattedDate );
 }
}
```

Compile and run the class `MyDateFormat, and it will print something like this:`

```
The date is Fri Feb 27 07:34:41 EST 2004
The date(dd-mm-yy) is 02-27-04
The date(dd-mm-yyyy) is 27-02-2004
The date(EEE, MMM d, ''yy) is Fri, Feb 27, '04
The time(hh:mm:ss a) is 07:34:41 AM
```

Java documentation for the class `SimpleDateFormat` describes more formats. You can also find more methods that deal with dates in other Java class called `java.util.Calendar.`

Method Overloading

A class may have more than one method with the same name, but with different argument lists. This is called *method overloading.*

For example, a method `println()` of the class `System` can be called with different types of arguments: `String`, `int`, `char`, and others:

```
System.out.println("Hello");

System.out.println(250);

System.out.println('A');
```

Even though it looks like we're calling the same method `println()` three times, in fact, we are calling different ones. You may say why don't create methods with different names, for example `printString()`, `printInt()`, `printChar()`?. One of the reasons is that it's easier to remember one name of a print method than several ones. There are other reasons as well for using method overloading, but those reasons are a bit complicated to explain and should be discussed in more advanced books.

If you remember, our class `Fish` from Chapter 4, which has a method `dive()` that expects one argument:

```
public int dive(int howDeep)
```

Let's create yet another version of this method that does not need any arguments. This method will force a fish to dive for five feet, unless the current depth becomes more than 100 feet. The new version of the class `Fish` has a new `final` variable `DEFAULT_DIVING` that has a value five feet.

Now the class Fish has two overloaded methods `dive()`.

```
public class Fish extends Pet {
 int currentDepth=0;
 final int DEFAULT_DIVING = 5;

  public int dive(){
    currentDepth=currentDepth + DEFAULT_DIVING;
    if (currentDepth > 100){
        System.out.println("I am a little fish and " +
                        " can't dive below 100 feet");
        currentDepth=currentDepth - DEFAULT_DIVING;
    }else{
        System.out.println("Diving for " +
                              DEFAULT_DIVING + " feet");
        System.out.println("I'm at " + currentDepth +
                        " feet below the sea level");
    }
            return currentDepth;
  }
  public int dive(int howDeep){
    currentDepth=currentDepth + howDeep;
    if (currentDepth > 100){
         System.out.println("I am a little fish and " +
                          " can't dive below 100 feet");
         currentDepth=currentDepth - howDeep;
    }else{
       System.out.println("Diving for " + howDeep +
                                          " feet");
       System.out.println("I'm at " + currentDepth +
                            " feet below the sea level");
    }
       return currentDepth;
  }

      public String say(String something){
        return "Don't you know that fishes do not talk?";
      }
      // constructor
      Fish(int startingPosition){
            currentDepth=startingPosition;
      }
}
```

The `FishMaster` can now call any of the overloaded methods `dive()`:

```
public class FishMaster {

      public static void main(String[] args) {

            Fish myFish = new Fish(20);

            myFish.dive(2);

            myFish.dive();   // a new overloaded method

            myFish.dive(97);
            myFish.dive(3);

            myFish.sleep();
      }
}
```

Constructors can also be overloaded, but only one of them will be used when an object is being created. JVM will call the constructor that has a matching argument list. For example, if you add a no-arguments constructor to the class `Fish`, the `FishMaster` can create its instance using one of the following ways:

```
Fish myFish = new Fish(20);
```

or

```
Fish myFish = new Fish();
```

Reading Keyboard Input

In this section you'll learn how a program can print questions in the command window and understand the responses that a user enters from the keyboard. This time we'll remove from the class `FishMaster` all hard-coded values that it passes to the class `Fish`. Now the program will ask the question *How Deep?*, and the fish will dive according to the user's responses.

You should be pretty comfortable by now with using *standard output* `System.out. By the way, the variable out` is of a data type `java.io.OutputStream`. Now I'll explain you how to deal with *standard input* `System.in`, and as you can guess, the type of the variable `in` is `java.io.InputStream`.

The next version of the class `FishMaster` displays a *prompt* on the system console and waits for the user's response. After the user types one or more characters and presses the button *Enter*, JVM places these characters into the object `InputStream` and pass them to the program.

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class FishMaster {

  public static void main(String[] args) {

   Fish myFish = new Fish(20);
   String feetString="";
   int feets;
   // create a input stream reader connected to
   // System.in, and pass it to the buffered reader
      BufferedReader stdin = new BufferedReader
                   (new InputStreamReader(System.in));

   // Keep diving until the user presses "Q"
      while (true) {
        System.out.println("Ready to dive.How deep?");
        try {
            feetString = stdin.readLine();
            if (feetString.equals("Q")){
            // Exit the program
              System.out.println("Good bye!");
              System.exit(0);
            }else {
        // Convert the feetString into an integer and
        // Dive according to the value of variable feet
             feets = Integer.parseInt(feetString);
             myFish.dive(feets);
            }
         } catch (IOException e) {
                  e.printStackTrace();
         }
       } // End while
  } // End main
}
```

A dialog between the user and the program FishMaster can look like this:

```
Ready to dive.How deep?
14
Diving for 14 feet
I'm at 34 feet below the sea level
Ready to dive.How deep?
30
Diving for 30 feet
I'm at 64 feet below the sea level
Ready to dive.How deep?
Q
Good bye!
```

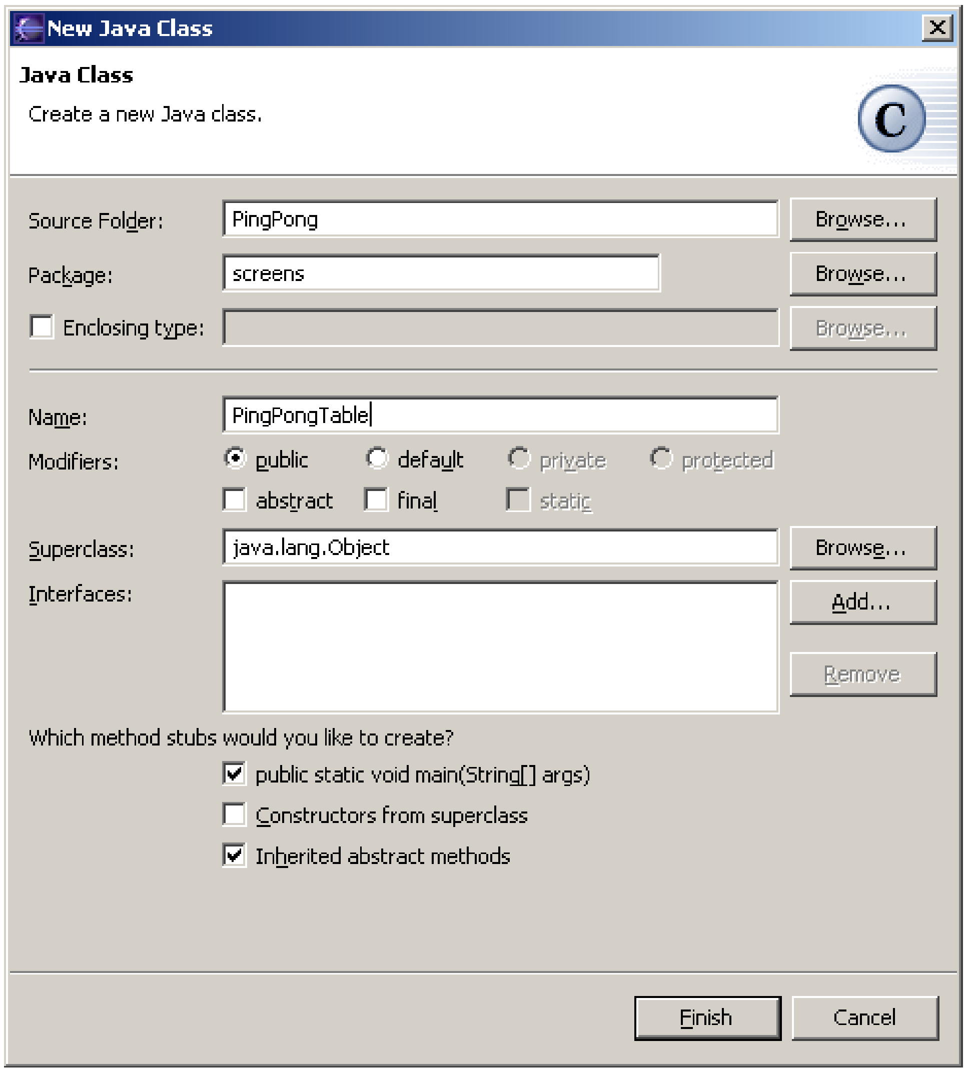
First, the `FishMaster` creates the `BufferedReader` stream that is connected to the standard input `System.in`. After that it displays the message *Ready to dive. How deep?* and the method readLine() pauses the program until the user hits the *Enter* button. The entered value comes as a `String`, that's why the `FishMaster` converts it to an integer and calls the method `dive()` on the class `Fish`. These action repeat in a loop until the user types the letter *Q* to exit the program. The line `feetString.equals("Q")` compares the value of the String variable `feetString` with the letter *Q*.

We were using the method `readLine()` to get the entire line entered by the user at once, but there is another method `System.in.read()` that allows you to process user's input one character at a time.

More on Java Packages

When programmers work on large projects that have lots of classes, they usually organize them in different *packages*. For example, one package can have all classes that display windows (screens), while another one can contain event listeners. Java also keeps its classes in packages, for example `java.io` for classes responsible for input/output operation, or `javax.swing` for Swing classes.

Let's create a new project in Eclipse called *PingPong*. This project will have classes in two packages: `screens` and `engine`. Now create a new class `PingPongTable` and enter the word `screens` in the field *Package*:


New Java Class

Java Class

Create a new Java class.

Source Folder:	PingPong	Browse...
Package:	screens	Browse...
☐ Enclosing type:		Browse...
Name:	PingPongTable	
Modifiers:	● public ○ default ○ private ○ protected	
	☐ abstract ☐ final ☐ static	
Superclass:	java.lang.Object	Browse...
Interfaces:		Add... Remove

Which method stubs would you like to create?

- ☑ public static void main(String[] args)
- ☐ Constructors from superclass
- ☑ Inherited abstract methods

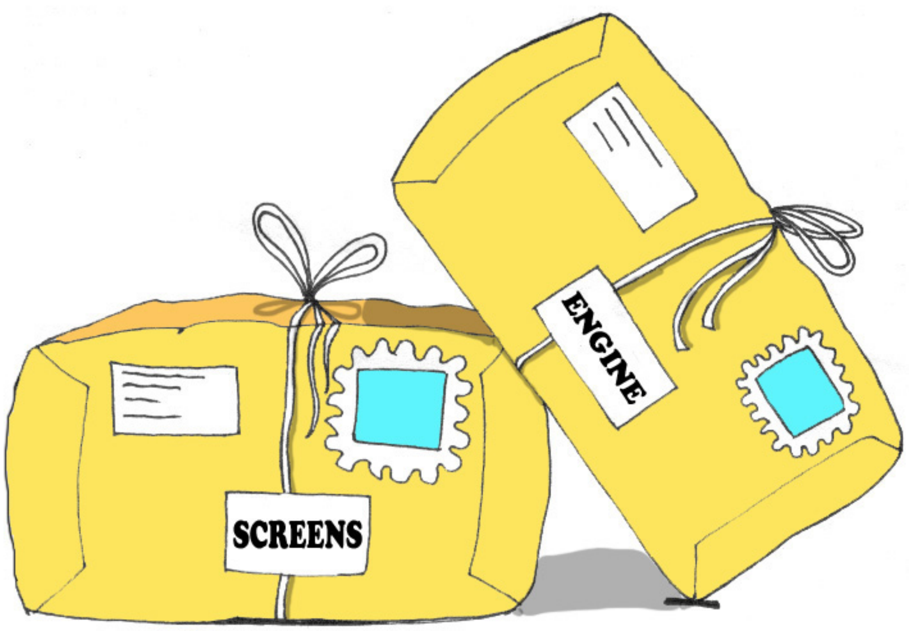
Finish Cancel

Press the button *Finish* and Eclipse will generate code that include the line with the package name.

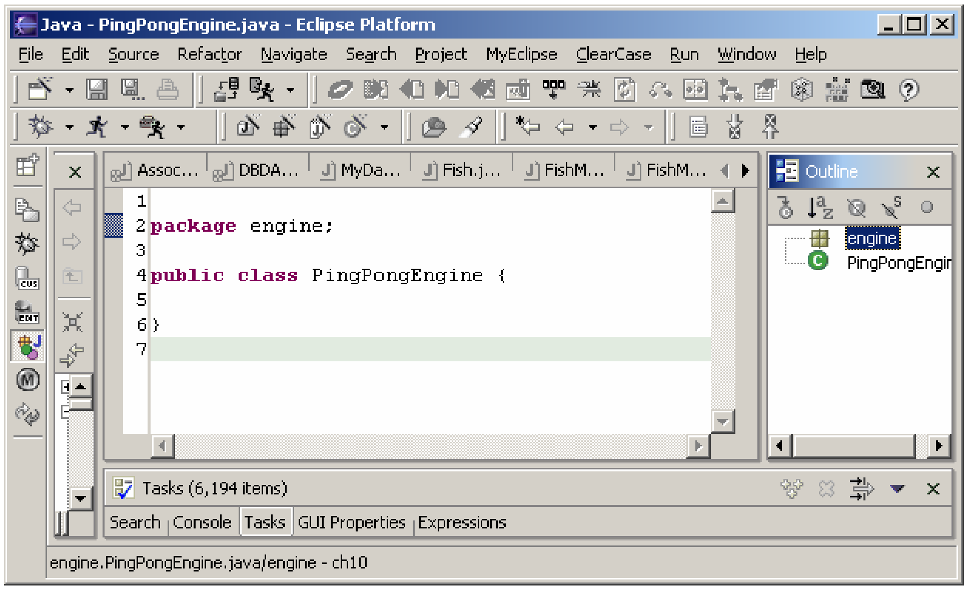
```
package screens;

public class PingPongTable {

        public static void main(String[] args) {
        }
}
```

By the way, if your class includes the line with the keyword `package`, you are not allowed to write anything but comments above this line.

Since each package is stored in a different folder on the disk, Eclipse creates the folder `screens` and put the file `PinPongTable.java` there. Check it out – there should be a folder c:\eclipse\workspace\PingPong\screens on your disk with files `PingPongTable.java` and `PingPongTable.class`.



Now create another class called `PingPongEngine` and enter the word `engine` as the package name. The `PingPong` project has two packages now:


Java - PingPongEngine.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project MyEclipse ClearCase Run Window Help
Assoc... DBDA... MyDa... Fish.j... FishM... FishM...
Outline
engine
PingPongEngir
Tasks (6,194 items)
Search Console Tasks GUI Properties Expressions
engine.PingPongEngine.java/engine - ch10

```java

package engine;

public class PingPongEngine {

}

```

Since our two classes are located in two different packages (and folders), the class `PingPongTable` won't see the class `PingPongEngine` unless you add the `import` statement.

```
package screens;

import engine.PingPongEngine;

public class PingPongTable {

 public static void main(String[] args) {
  PingPongEngine gameEngine = new PingPongEngine();
 }
}
```

Java packages not only help better organize your classes, but they can be also used to restrict access to their classes from the "foreigners" sitting in other packages.

Access Levels

Java classes, methods and member variables could have `public`, `private`, `protected` and package access levels. Our class `PingPongEngine` has `public` access level. This means than any class can access it. Let's make a simple experiment – remove the keyword `public` from the declaration of the class `PingPongEngine`. Now the class `PingPongTable` won't even compile giving an error *PingPongEngine can not be resolved or is not a type*. This means that the class `PingPongTable` *does not see* the class `PingPongEngine` anymore.

If no access level is specified, the class will have a package access level. This means that it will be available only for the classes located in the same package.

Similarly, if you forget to give a public access to methods of the class `PingPongEngine`, the `PingPongTable` will complain saying that these methods are not visible. You'll see how the access levels are used in the next chapter while creating a ping pong game.

The `private` access level is used to hide class variables or methods from the outside world. Think of a car – most of the people have no clue how many parts are there under the hood, and what actually happens when a driver pushes the brake pedal.

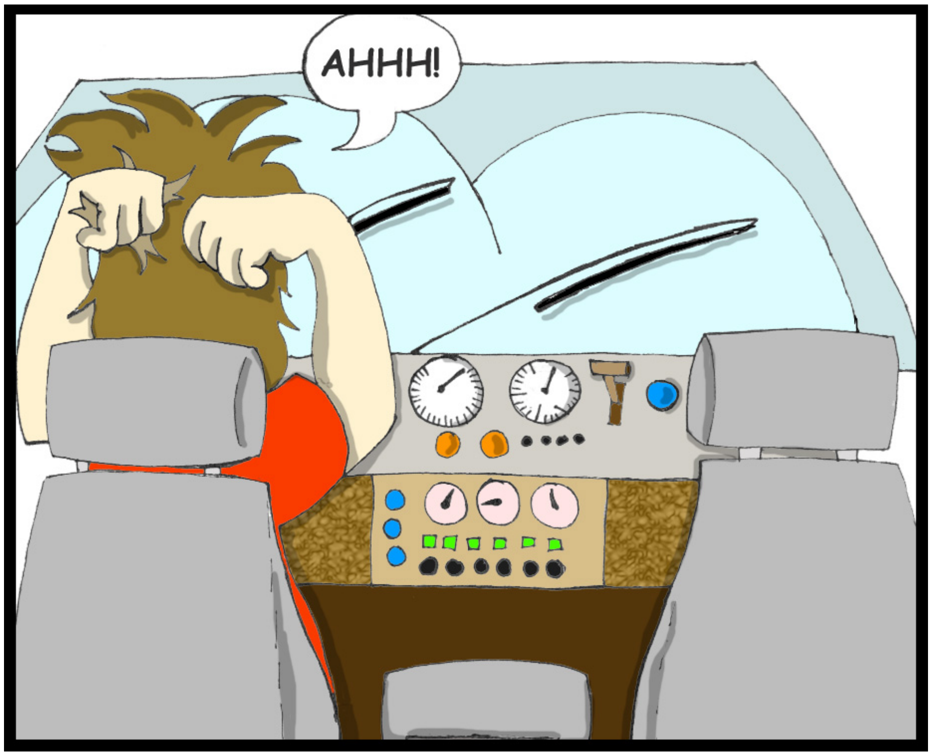
Look at the next code sample - in Java, we can say that the object `Car` *exposes* only one public method – `brake()`, which internally may call several other methods that a driver does not need to know about. For example, if the driver pushes the brake pedal too hard, the car's computer may apply special anti-lock brakes. I already mentioned before that Java programs control such complicated robots as Mars rovers, let alone simple cars.

```
public class Car {

 // This private variable can be used inside
 // this class only
  private String brakesCondition;

 // A public method brake() calls private methods
 // to decide which brakes to use
  public void brake(int pedalPressure){
    boolean useRegularBrakes;
    useRegularBrakes=
              checkForAntiLockBrakes (pedalPressure);

    if (useRegularBrakes==true){
      useRegularBrakes();
    }else{
      useAntiLockBrakes();
    }
  }

  // This private method can be called inside
  // this class only
  private boolean checkForAntiLockBrakes(int pressure){
    if (pressure > 100){
        return true;
    }else {
        return false;
    }
  }

   // This private method can be called inside this
   // class only
  private void useRegularBrakes(){
   // code that sends a signal to regular brakes
  }

   // This private method can be called inside this
   // class only
  private void useAntiLockBrakes(){
   // code that sends a signal to anti-lock brakes
  }
}
```

There is one more Java keyword `protected` that controls access level. If you use this keyword in a method signature, this method will be visible inside the class, from its subclasses, and from other classes located in the same package. But it won't be available for independent classes located in other packages.

One of the main features of object-oriented languages is called *encapsulation*, which is an ability to hide and protect elements of a class.

When you design a class, hide methods and member variables that should not be visible from outside. If car designers would not hide control of some of the under-the-hood operations, the driver would have to deal with hundreds of buttons, switches and gauges.



In the next section you can find a class `Score` that hides its attributes in `private` variables.

Getting Back to Arrays

In Chapter 9 the program `ScoreWriter` has created an array of `String` objects that stored names and scores of players in a file. It's about time to learn how to use arrays for storing any objects.

This time we'll create an object to represent a game score, and it will have such attributes as player's first and last names, score, and the last date when the game was played.

The class `Score` is next. It has *getters* and *setters* for each of its attributes, which are declared private. Well, it might not be obvious why the caller class can not just set the value of the attribute score just like this:

```
Score.score = 250;
```
instead of
```
Score.setScore(250);
```

Try to think out of the box. What if later on we decide that our program has to play some music whenever a player reaches the score of 500. If the class `Score` have a method `setScore()`, you just need to modify only this method to add code that checks the score and plays music if needed. The caller class will keep calling the musical version of the method setScore() the same way. If the caller class would set the value directly, the musical changes had to be implemented in this caller. And what if you'd like to re-use the class Score in two different game programs? In case of direct attribute changes you'd have to implement these changes in two caller classes, but if you have a setter method, the changes are *encapsulated* there and will immediately start working for each caller class.

```java
import java.util.Date;

public class Score {
  private String firstName;
  private String lastName;
  private int score;
  private Date playDate;

  public String getFirstName(){
      return firstName;
  }
  public void setFirstName(String firstName){
      this.firstName = firstName;
  }
  public String getLastName(){
        return lastName;
  }
  public void setLastName(String lastName){
        this.lastName = lastName;
  }
  public int getScore(){
        return score;
  }
  public void setScore(int score){
      this.score=score;
  }
  public Date getPlayDate(){
        return playDate;
  }
  public void setPlayDate(Date playDate){
      this.playDate=playDate;
  }
// Concatenate all attributes into a String
// and add a new line character at the end.
// This method is handy if the caller class needs
// to print all values in one shot, for example
// System.out.println(myScore.toString());
  public String toString(){
      String scoreString = firstName + " " +
         lastName + " " +  score + " " + playDate +
        System.getProperty("line.separator");
      return scoreString;
  }
}
```

The program `ScoreWriter2` will create instances of the object `Score` and assign the values to their attributes.

```java
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;
import java.util.Date;

public class ScoreWriter2 {

/**
 The method main performs the following actions:
 1. Create an instance of array
 2. Create Score objects and populate array with them
 3. Write the scores data into a file
*/
      public static void main(String[] args) {

      FileWriter myFile = null;
      BufferedWriter buff = null;

      Date today = new Date();
      Score scores[] = new Score[3];

      // The player #1
      scores[0]=new Score();
      scores[0].setFirstName("John");
      scores[0].setLastName("Smith");
      scores[0].setScore(250);
      scores[0].setPlayDate(today);

      // The player #2
      scores[1]=new Score();
      scores[1].setFirstName("Anna");
      scores[1].setLastName("Lee");
      scores[1].setScore(300);
      scores[1].setPlayDate(today);

      // The player #3
      scores[2]=new Score();
      scores[2].setFirstName("David");
      scores[2].setLastName("Dolittle");
      scores[2].setScore(190);
      scores[2].setPlayDate(today);
```

Class `ScoreWriter2 (part 1 of 2)`

```
   try {
      myFile = new FileWriter("c:\\scores2.txt");
      buff = new BufferedWriter(myFile);

      for (int i=0; i < scores.length; i++) {
       // Convert each of the scores to a String
       // and write it into scores2.txt
       buff.write(scores[i].toString());
       System.out.println("Writing  " +
                         scores[i].getLastName() );
      }
    System.out.println("File writing is complete");

  }catch (IOException e){
     e.printStackTrace();
  } finally {
     try{
        buff.flush();
        buff.close();
        myFile.close();
     }catch(IOException e1){
        e1.printStackTrace();
     }
  }
 }
}
```

Class `ScoreWriter2` (part 2 of 2)

If a program tries to access an array element that is beyond the arrays length, i.e. `scores[5].getLastName()`, Java throws the `ArrayIndexOutOfBoundsException`.

Class `ArrayList`

The package `java.util` includes classes that are quite handy when a program needs to store several instances (*a collection*) of some objects in memory. Some of the popular collection classes from this package are `ArrayList`, `Vector`, `HashTable`, `HashMap` and `List`. I'll show you how to use the class `java.util.ArrayList`.

The drawback of regular arrays is that you have to know the number of array elements in advance. Remember, to create an instance of an array you have to put a number between the brackets:

```
String[] myFriends = new String[5];
```

Class ArrayList does not have this restriction – you can create an instance of this collection without knowing how many objects will be there – just add more elements as needed.

Why use arrays, let's just always use ArrayList! Unfortunately, nothing comes for free, and you have to pay the price for a convenience – ArrayList is a little slower than a regular array, and you could only store objects there, for example you can not just store a bunch of int numbers in an ArrayList.

To create and populate an ArrayList object, you should instantiate it first, create instances of the objects you are planning to store there, and add them to the ArrayList by calling its method add(). The next little program will *populate* an ArrayList with String objects and print the content of this collection.

```
import java.util.ArrayList;

public class ArrayListDemo {
  public static void main(String[] args) {
      // Create and populate an ArrayList
      ArrayList friends = new ArrayList();
      friends.add("Mary");
      friends.add("Ann");
      friends.add("David");
      friends.add("Roy");

      // How many friends are there?
      int friendsCount = friends.size();

    // Print the content of the ArrayList
    for (int i=0; i<friendsCount; i++){
      System.out.println("Friend #" + i + " is "
          + friends.get(i));
    }
  }
}
```

This program will print the following lines:

```
Friend #0 is Mary
Friend #1 is Ann
Friend #2 is David
Friend #3 is Roy
```

The method get() *extracts* from an ArrayList the element located at a particular position. Since you can store any objects in a collection, the method get() returns each element as a Java Object, and it's a responsibility of the program to cast this object to a proper data type. We did not have to do it in the previous example only because we stored String objects in the collection friends, and Java converts an Object to a String automatically.

But if you decide to store in ArrayList some other objects, for example instances of the class Fish, the proper code to add and extract a particular Fish may look as in the program FishTank that comes next. First, this program creates a couple of instances of the class Fish, assigns some value to color, weight and current depth and stores them in the ArrayList called fishTank. Then, the program gets the objects from this collection, casts them to the class Fish and prints their values.

```java
import java.util.ArrayList;

public class FishTank {
  public static void main(String[] args) {
      ArrayList fishTank = new ArrayList();
      Fish theFish;

      Fish aFish = new Fish(20);

      aFish.color = "Red";
      aFish.weight = 2;
      fishTank.add(aFish);

      aFish = new Fish(10);
      aFish.color = "Green";
      aFish.weight = 5;
      fishTank.add(aFish);

      int fishCount = fishTank.size();

      for (int i=0;i<fishCount; i++){
        theFish = (Fish) fishTank.get(i);
        System.out.println("Got the " +
         theFish.color + " fish that weighs " +
           theFish.weight + " pounds. Depth:" +
                        theFish.currentDepth);
      }
  }
}
```

Here's an output of the program FishTank:

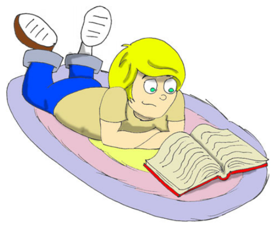
```
Got the Red fish that weighs 2.0 pounds. Depth:20
Got the Green fish that weighs 5.0 pounds. Depth:10
```

Now that you've read about the Java access levels, classes Pet and Fish can be modified a bit. Such variables as age, color, weight and height should be declared as protected, and the variable currentDepth should be private. You should add new public methods such as getAge() to return the value of the variable age, and setAge() has to set the value of this variable, an so on.

Programmers with good manners do not allow one class directly modify properties of another one – the class should provide methods that modify its internals. That's why the class `Score` from the previous section was designed with `private` variables, which could be changed with setters and getters.

In this chapter I've shown you different Java elements and techniques that seem to be unrelated to each other. But all these elements are often used by professional Java programmers. After completion of the practical assignments for this chapter you should have a better understanding of how these elements work together.

Additional Reading



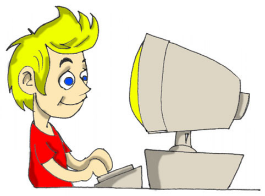
1. Java Collections:
http://java.sun.com/docs/books/tutorial/collections/intro/

2. Class `ArrayList`:
http://java.sun.com/j2se/1.5.0/docs/api/java/util/ArrayList.html

3. Class `Vector`:
http://java.sun.com/j2se/1.5.0/docs/api/java/util/Vector.html

4. Class `Calendar`:
http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html

Practice



1.Add an overloaded no-argument constructor to the class `Fish`. This constructor should set the starting position to 10 feet. The class `FishMaster` will create an instance of the object `Fish` just like this:

```
Fish myFish = new Fish();
```

2. Add a four-argument constructor to the class `Score`. Create a program `ScoreWriter3` that will populate the instances of the `Score` objects not by using setters, but rather at the time when the score are created, for example

```
Score aScore =
  new Score("John", "Smith", 250, today);
```

Practice for Smarty Pants

Learn online how to use the class `Vector` and try to create a program `VectorDemo` that is similar to the program `ArrayLiastDemo`.

Chapter 11. Back to Graphics – the Ping Pong Game

In chapters 5, 6 and 7 we've used some of AWT and Swing components. Now I'll show you how you can draw and move such objects as ovals, rectangles and lines in a window. You'll also learn how to process mouse and keyboard events. To add a little fun to these boring subjects, in this chapter we'll be learning all these thing while creating a ping pong game. This game will have two players and I call them *the kid* and *computer*.

The Strategy

Let's come up with some rules of the game:

1. The game lasts until one of the players (the kid or computer) will reach the score of 21.
2. Kid's racket movements will be controlled by the computer mouse.
3. Game score has to be displayed at the bottom of the window.
4. A new game starts when a player presses the *N* key on the keyboard, *Q* ends the game, and *S* serves the ball.
5. Only the kid can serve the ball.
6. To win a point the ball should go beyond the racket's vertical line when the racket is not blocking the ball.
7. When computer bounces the ball, it can move only horizontally to the right.
8. If the ball contacts the kid's racket in the upper half of the table, the ball should be moving in the up-and-left direction. If the ball was located in the bottom part of the table, it should move in the down-and-left direction.

You must be thinking that it's going to be too difficult to program. The trick is to split a complicated task into a set of smaller and simpler tasks, and try to solve each of them one at a time.

Instead of just saying "My computer does not work" (a big problem), try to see what exactly does not work (find the smaller one).

1.Is computer plugged into the power outlet (yes/no)? *Yes.*
2. When I start the computer, do I see the screen with all these icons (yes/no)? *Yes.*
3. Can I move the mouse on the screen (yes/no)? *No.*
4. Is the mouse cable plugged in properly (yes/no)? *No.*

Just plug in the mouse and computer will start working again! A big problem came down to fixing a loose mouse cable.

This trick is called *analytical thinking*, and it helps not only in programming, but everywhere in your life – do not get frustrated if you can't achieve a big goal, split it in a set of the smaller ones an reach them one at time!

That's why the first version of the game will have only some of these rules implemented – it'll just paint the table, move the racket and display coordinates of the mouse pointer when you click the mouse button.

The Code

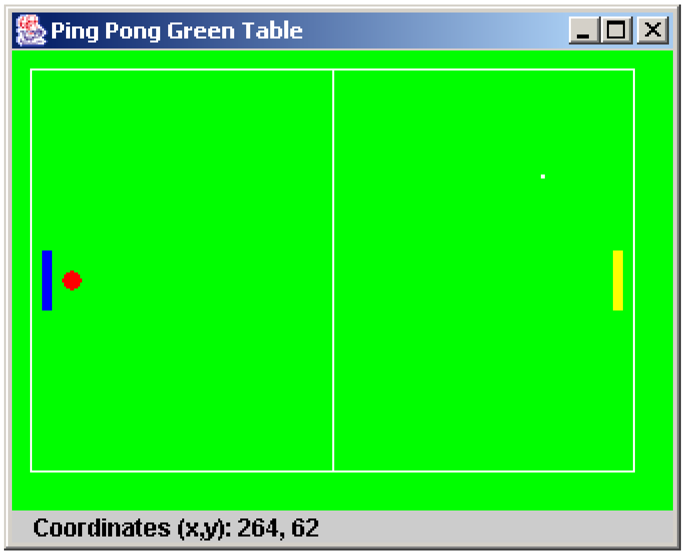
This game will consist of the following three classes:

- Class `PingPongGreenTable` will take care of the visual part. During the game it'll be displaying the table, rackets and the ball.

- Class `PingPongGameEngine` will be responsible for calculations of the ball and rackets' coordinates, starting and ending the game, and serving the ball. The engine class will pass the current coordinates of components to `PingPongGreenTable`, which will repaint itself accordingly.

- Interface `GameConstants` will contain declarations of all constants that the game needs, for example width and height of the table, starting positions of the rackets and so on.

The ping pong table will look like this:



The first version of this game will do only three things:

- Display a green ping pong table.
- Display coordinates of the mouse pointer when you click on the mouse.
- Move the kid's racket up and down.

Two pages later you can see our `PingPongGreenTable` class that is a subclass of the Swing's `JPanel`. Look at the code while reading the text below.

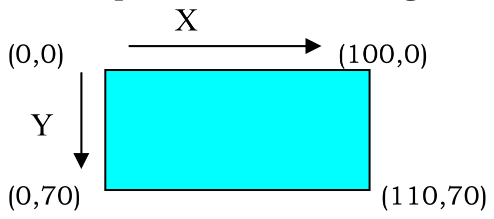
Since our game needs to know exact coordinates of the mouse pointer, constructor of the `PingPongGreenTable` class will create an instance of the event listener class `PingPongGameEngine`. This class will perform some actions when the kid clicks on the mouse button or just moves the mouse.

The method `addPaneltoFrame()` creates a label that will display coordinates of the mouse.

This class is not an applet, and that's why instead of the method `paint()` it uses the method `paintComponent()`. This method is called either by JVM when it needs to refresh the window, or when our program calls a method `repaint()`. You've read it right, method `repaint()` internally calls `paintComponent()` and provides your class with an object `Graphics` so you can paint on the window. We'll call this method every time after recalculating coordinates of the rackets or the ball to display them in the proper position.

To paint a racket, set the color first, and then fill a rectangle with this paint using the method `fillRect()`. This method needs to know `X` and `Y` coordinates of the top left corner of the rectangle and its width and height in pixels. The ball is painted using the method `fillOval()`, and it needs to know coordinates of the center of the oval, its height and width. When the height and width of the oval are the same, it looks like a circle.

`X` coordinate in a window grows from left to right, and `Y` coordinate increases from top to bottom. For example, the width of this rectangle is 100 pixels, and the height is 70:



`X` and `Y` coordinates of the corners of this rectangle are shown in parentheses.

Another interesting method is `getPreferredSize()`. We create an instance of a Swing class `Dimension` to set the size of the table. JVM needs to know dimensions of the window, that's why it calls the method `getPreferredSize()` of the `PingPongGreenTable` object. This method returns to JVM an object `Dimension` that we've created in the code according to the size of our table.

Both table and engine classes use some constant values that do not change. For example, class `PingPongGreenTable` uses the width and height of the table, and `PingPongGameEngine` needs to know ball movement increments – the smaller the increment, the smoother the movement. It's convenient to keep all the constants (`final` variables) in an interface. In our game the name of the interface is `GameConstants`. If a class needs these values, just add `implements GameConstants` to the class declaration and use any of the `final` variables from this interface as if they were declared in the class itself! That's why both table and engine classes implement `GameConstants` interface.

If you decide to change the size of the table, ball, or racket you'll need to do it only in one place – in the `GameConstants` interface. Let's look at the code of the class `PingPongGreenTable` and the interface `GameConstants`.

```java
package screens;

import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.BoxLayout;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import java.awt.Point;
import java.awt.Dimension;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.Color;
import engine.PingPongGameEngine;
/**
*  This class paints a green ping pong table
*  and displays coordinates of the point
*  where the user clicked the mouse
*/
public class PingPongGreenTable extends JPanel
                            implements GameConstants{
    JLabel label;
    public Point point = new Point(0,0);

    public int ComputerRacket_X =15;
    private int kidRacket_Y =KID_RACKET_Y_START;

    Dimension preferredSize = new
              Dimension(TABLE_WIDTH,TABLE_HEIGHT);

    // This method sets the size of the frame.
    // It's called by JVM
    public Dimension getPreferredSize() {
      return preferredSize;
    }
```

Class `PingPongGreenTable` (part 1 of 3)

```
     // Constructor. Creates a listener for mouse events
     PingPongGreenTable(){

       PingPongGameEngine gameEngine =
                            new PingPongGameEngine(this);
     // Listen to mouse clicks to show its coordinates
       addMouseListener(gameEngine);
     // Listen to mouse movements to move the rackets
       addMouseMotionListener(gameEngine);
     }

    // Add a panel with a JLabel to the frame
     void addPaneltoFrame(Container container) {
       container.setLayout(new BoxLayout(container,
                                        BoxLayout.Y_AXIS));
       container.add(this);
       label = new JLabel("Click to see coordinates");
       container.add(label);
    }

    // repaint the window. This method is called by JVM
    // when it needs to refresh the screen or when a
    // method repaint() is called from PingPointGameEngine
     public void paintComponent(Graphics g) {

       super.paintComponent(g);
       g.setColor(Color.GREEN);
       // paint the table green
       g.fillRect(0,0,TABLE_WIDTH,TABLE_HEIGHT);

       g.setColor(Color.yellow);

       // paint the right racket
       g.fillRect(KID_RACKET_X_START,kidRacket_Y,5,30);
       g.setColor(Color.blue);

       // paint the left racket
       g.fillRect(ComputerRacket_X,100,5,30);

       g.setColor(Color.red);
       g.fillOval(25,110,10,10); //paint the ball

       g.setColor(Color.white);
       g.drawRect(10,10,300,200);
       g.drawLine(160,10,160,210);
```

Class `PingPongGreenTable` (part 2 of 3)

```
 // Display a point as a small 2x2 pixels rectangle
      if (point != null) {
         label.setText("Coordinates (x,y): " +
                       point.x + ", " + point.y);
         g.fillRect(point.x, point.y, 2, 2);
      }
    }

 // Set the current position of the kid's racket
    public void setKidRacket_Y(int xCoordinate){
      this.kidRacket_Y = xCoordinate;
    }

 // Return the current position of the kid's racket
    public int getKidRacket_Y(int xCoordinate){
       return kidRacket_Y;
    }

    public static void main(String[] args) {
      // Create an instance of the frame
      JFrame f = new JFrame("Ping Pong Green Table");
      // Ensure that the window can be closed
      // by pressing a little cross in the corner
      f.setDefaultCloseOperation(
                  WindowConstants.EXIT_ON_CLOSE);

      PingPongGreenTable table =
                             new PingPongGreenTable();
      table.addPaneltoFrame(f.getContentPane());
      // Set the size and make the frame visible
      f.pack();
      f.setVisible(true);
    }
}
```

Class PingPongGreenTable (part 3 of 3)

The next is the interface GameConstants. All values of the variables are in pixels. Use capital letters to name final variables:

```
package screens;

public interface GameConstants {
      public final int TABLE_WIDTH =  320;
      public final int TABLE_HEIGHT = 220;
      public final int KID_RACKET_Y_START = 100;
      public final int KID_RACKET_X_START = 300;
      public final int TABLE_TOP = 12;
      public final int TABLE_BOTTOM = 180;

      public final int RACKET_INCREMENT = 4;
}
```

A running program can not change vales of these variables, because they were declared as final. But if, for example you decide to increase the size of the table, you'll need to change the values of TABLE_WIDTH and TABLE_HEIGHT and then recompile the GameConstants interface.

Decision-maker in this game is the class PingPongGameEngine, which implements two mouse-related interfaces. The MouseListener will have code only in the method mousePressed(). On every mouse click this method will draw a small white point on the table and display its coordinates. Actually, this code is useless for our game, but it'll show you in a simple way how to get coordinates of the mouse from the MouseEvent object that is given to the program by JVM.

A method mousePressed() sets the coordinates of the variable point depending on where the mouse poiner was when the player pressed its button. After coordinates are set, it asks JVM to repaint the table.

The MouseMotionListener reacts on movements of the mouse over the table, and we'll use its method mouseMoved() to move the kid's racket up or down.

A method mouseMoved() calculates the next position of the kid's racket. If the mouse pointer is above the racket (the Y coordinate of the mouse is less then Y coordinate of the racket), it ensures that the racket will not go over the top of the table.

When constructor of the table creates the engine object, it passes to the engine a reference to the table's instance (the keyword this represents a reference to memory location of the object PingPongGreenTable). Now the engine can "talk" to the table, for example set new coordinates of the ball or repaint the table if needed. If this part is not clear, you may want to re-read a section about passing data between classes in Chapter 6.

In our game rackets move vertically from one position to another using four pixel increment as defined in the interface `GameConstants` (the engine class implements this interface). For example, the next line subtracts four from the value of the variable `kidRacket_Y`:

```
kidRacket_Y -= RACKET_INCREMENT;
```

For example, if the `Y` coordinate of the racket was 100, after this line of code its value becomes 96, which means that the racket has to be moved up. You can get the same result using the following syntax:

```
kidRacket_Y = kidRacket_Y - RACKET_INCREMENT;
```

If you remember, we've talked about different ways of changing variable values in Chapter 3.

The class `PingPongGameEngine` is next.

```
package engine;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import screens.*;
public class PingPongGameEngine  implements
       MouseListener, MouseMotionListener, GameConstants{

 PingPongGreenTable table;
 public int kidRacket_Y = KID_RACKET_Y_START;
 // Constructor. Stores a reference to the table
 public PingPongGameEngine(PingPongGreenTable
                                         greenTable){
      table = greenTable;
 }
// Methods required by the MouseListener interface
 public void mousePressed(MouseEvent e) {
 // Get X and Y coordinates of the mouse pointer
 // and set it to the "white point" on the table
      table.point.x = e.getX();
      table.point.y = e.getY();
//The method repaint internally calls the table's
// method paintComponent() that refreshes the window
      table.repaint();
 }
 public void mouseReleased(MouseEvent e) {};
 public void mouseEntered(MouseEvent e) {};
 public void mouseExited(MouseEvent e) {};
 public void mouseClicked(MouseEvent e) {};

// Methods required by the MouseMotionListener interface
 public void mouseDragged(MouseEvent e) {}

 public void mouseMoved(MouseEvent e) {
  int mouse_Y = e.getY();
  // If a mouse is above the kid's racket
  // and the racket did not go over the table top
  // move it up, otherwise move it down
  if (mouse_Y < kidRacket_Y && kidRacket_Y > TABLE_TOP){
      kidRacket_Y -= RACKET_INCREMENT;
  }else if (kidRacket_Y < TABLE_BOTTOM) {
                  kidRacket_Y += RACKET_INCREMENT;
  }
  // Set the new position of the racket table class
   table.setKidRacket_Y(kidRacket_Y);
   table.repaint();
  }
}
```

Java Threads Basics

So far, all our programs perform actions in a sequence – one after another. If a program calls two methods, the second method waits until the first one completes. In other words, each of our programs has only one *thread of execution*.

In a real life though, we can do several things at the same time, for example eat, talk on the phone, watch TV, and do the homework. To do all these actions *in parallel* we use several *processors*: hands, eyes, and mouth.

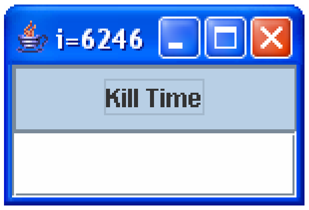
Some of the more expensive computers also have two or more processors. But most likely your computer has only one processor that performs calculations, sends commands to the monitor, disk, remote computers, and so on.

But even one processor can perform several actions at once if a program uses *multiple threads*. One Java class can start several *threads of execution* that will take turns in getting slices of the processor's time.

A good example of a program that creates multiple threads is a Web browser. You can browse the Internet while downloading some files – one program runs two threads of execution.

The next version of our ping pong game will have one thread that displays the table. The second thread will calculate coordinates of the ball and rackets and will send commands to the first thread to repaint the window. But first, I'll show you two very simple programs to give you a better feeling of why threads are needed.

Each of these sample programs will
display a button and a text field.



When you press the button *Kill Time*, the program will start a loop that will increment a variable thirty thousand times. The current value of the variable-counter will be displayed on the title bar of the window. The class `NoThreadsSample` has only one thread of execution, and *you won't be able to type anything in the text field until the loop is done*. This loop takes all processor's time, that's why the window is locked.

Compile and run this class and see for yourself that the window is locked for some time. Note that this class creates an instance of `JTextField` and passes it to the content pane without declaring a variable for this instance. If you are not planning to get or set attributes of this object in this program, you do not need such reference variable.

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class NoThreadsSample extends JFrame
                            implements ActionListener{
  // Constructor
  NoThreadsSample(){
  // Create a frame with a button and a text field
      GridLayout gl =new GridLayout(2,1);
      this.getContentPane().setLayout(gl);
      JButton myButton = new JButton("Kill Time");
      myButton.addActionListener(this);
      this.getContentPane().add(myButton);
      this.getContentPane().add(new JTextField());
  }
  // Process button clicks
  public void actionPerformed(ActionEvent e){
  // Just  kill some time to show
  // that window controls are locked
      for (int i=0; i<30000;i++){
        this.setTitle("i="+i);
      }
    }

  public static void main(String[] args) {
    // Create an instance of the frame
    NoThreadsSample myWindow = new NoThreadsSample();
    // Ensure that the window can be closed
    // by pressing a little cross in its corner
    myWindow.setDefaultCloseOperation(
                      WindowConstants.EXIT_ON_CLOSE);

    // Set the frame's size - top left corner
    // coordinates, width and height
    myWindow.setBounds(0,0,150, 100);
    //Make the window visible
    myWindow.setVisible(true);
  }
}
```

The next version of this little window will create and start a separate thread for the loop, and the main window's thread will allow you to type in the text field while the loop is running.

You can create a thread in Java using one of the following ways:

1. Create an instance of the Java class `Thread` and pass to this instance an object that implements `Runnable` interface. If your class implements `Runnable` interface the code will look like this:

```
Thread worker = new Thread(this);
```

This interface requires you to write in the method `run()` the code that must be running as a separate thread. But to start the thread, you have to call the method `start()`, that will actually call your method `run()`. I agree, it's a bit confusing, but this is how you start the thread:

```
worker.start();
```

2. Make a subclass of the class `Thread` and implement the method `run()` there. To start the thread call the method `start()`.

```
public class MyThread extends Thread{

  public static void main(String[] args) {
      MyThread worker = new MyThread();
      worker.start();
  }
  public void run(){
    // your code goes here
  }
}
```

I'll be using the first method in the class `ThreadsSample` because this class already extends `JFrame`, and you can't extend more than one class in Java.

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ThreadsSample extends JFrame
             implements ActionListener, Runnable{

  // Constructor
  ThreadsSample(){
   // Create a frame with a button and a text field
      GridLayout gl =new GridLayout(2,1);
      this.getContentPane().setLayout(gl);
      JButton myButton = new JButton("Kill Time");
      myButton.addActionListener(this);
      this.getContentPane().add(myButton);
      this.getContentPane().add(new JTextField());
  }

  public void actionPerformed(ActionEvent e){
  // Create a thread and execute the kill-time-code
  // without blockiing the window
    Thread worker = new Thread(this);
    worker.start();  // this calls the method run()
  }

  public void run(){
  // Just  kill some time to show that
  // window controls are NOT locked
      for (int i=0; i<30000;i++){
        this.setTitle("i="+i);
      }
   }

  public static void main(String[] args) {

      ThreadsSample myWindow = new ThreadsSample();
  // Ensure that the window can be closed
  // by pressing a little cross in the corner
      myWindow.setDefaultCloseOperation(
                    WindowConstants.EXIT_ON_CLOSE);

  // Set the frame's size and make it visible
      myWindow.setBounds(0,0,150, 100);
      myWindow.setVisible(true);
  }
}
```

Class `ThreadsSample` starts a new thread when you click on the button *Kill Time*. After this, the thread with a loop and the main thread take turn in getting slices of the processor's time. Now you can type in the text field (the main thread), while the other thread runs the loop!

Threads deserve much better study that these couple of pages, and I encourage you to do some additional reading on this topic.

Finishing Ping Pong Game

After a brief introduction of threads, we are ready to modify the code of our ping pong game classes.

Let's start with the class `PingPongGreenTable`. We do not need to display a white point when the user clicks the mouse – this was just an exercise to learn how to display coordinates of the mouse pointer. That's why we'll remove the declaration of the variable `point` and the lines that paint the white point from the method `paintComponent()`. Also, constructor does not need to add `MouseListener` anymore, because it only displays the point's coordinates.

On the other hand, this class should react to some of the keyboard buttons (*N* for new game, *S* for serving the ball, and *Q* to quit the game). The method `addKeyListener()` will take care of this.

To make our code a little more encapsulated, I've also moved the `repaint()` calls from the engine class to `PingPongGreenTable`. Now this will be responsible for repainting itself when needed.

I've also added methods to change positions of the ball, computer's racket and to display messages.

```
package screens;

import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.BoxLayout;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import java.awt.Dimension;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.Color;
import engine.PingPongGameEngine;
/**
*  This class paints the green ping pong table,
*  ball, rackets and displays the score
*/
public class PingPongGreenTable extends JPanel
                            implements GameConstants{
  private JLabel label;

  private int computerRacket_Y =
                            COMPUTER_RACKET_Y_START;
  private int kidRacket_Y = KID_RACKET_Y_START;
  private int ballX = BALL_START_X;
  private int ballY = BALL_START_Y;

  Dimension preferredSize = new
                Dimension(TABLE_WIDTH,TABLE_HEIGHT);

  // This method sets the size of the frame.
  // It's called by JVM
  public Dimension getPreferredSize() {
    return preferredSize;
  }

  // Constructor. Creates a listener for mouse events
  PingPongGreenTable(){

    PingPongGameEngine gameEngine =
                        new PingPongGameEngine(this);
    // Listen to mouse movements to move the rackets
    addMouseMotionListener(gameEngine);
    //Listen to the keyboard events
    addKeyListener(gameEngine);
  }
```

Class `PingPongGreenTable` (part 1 of 3)

```
// Add a panel with a JLabel to the frame
void addPaneltoFrame(Container container) {
   container.setLayout(new BoxLayout(container,
                                  BoxLayout.Y_AXIS));
   container.add(this);
   label = new JLabel(
     "Press N for a new game, S to serve or Q to quit");
   container.add(label);
}

// repaint the window. This method is called by JVM
// when it needs to refresh the screen or when a
// method repaint() is called from PingPointGameEngine
public void paintComponent(Graphics g) {

   super.paintComponent(g);
   g.setColor(Color.GREEN);
   // paint the table green
   g.fillRect(0,0,TABLE_WIDTH,TABLE_HEIGHT);

   g.setColor(Color.yellow);
   // paint the right racket
   g.fillRect(KID_RACKET_X, kidRacket_Y,
                            RACKET_WIDTH, RACKET_LENGTH);
   g.setColor(Color.blue);
   // paint the left racket
   g.fillRect(COMPUTER_RACKET_X, computerRacket_Y,
                             RACKET_WIDTH,RACKET_LENGTH);
   // paint the ball
   g.setColor(Color.red);
   g.fillOval(ballX,ballY,10,10);
   //draw the white lines
   g.setColor(Color.white);
   g.drawRect(10,10,300,200);
   g.drawLine(160,10,160,210);
   // Set the focus to the table, so the  key
   // listenerwill send commands to  the table
   requestFocus();
}

// Set the current position of the kid's racket
 public void setKidRacket_Y(int yCoordinate){
   this.kidRacket_Y = yCoordinate;
   repaint();
 }
```

Class `PingPongGreenTable` (part 2 of 3)

```
 // Return current posiition of the kid's racket
 public int getKidRacket_Y(){
    return kidRacket_Y;
 }

// Set the current position of the computer's racket
 public void setComputerRacket_Y(int yCoordinate){
    this.computerRacket_Y = yCoordinate;
    repaint();
 }

 // Set the game's message
 public void setMessageText(String text){
    label.setText(text);
    repaint();
 }

 // Set the game's message
 public void setBallPosition(int xPos, int yPos){
    ballX=xPos;
    ballY=yPos;
    repaint();
 }

 public static void main(String[] args) {

 // Create an instance of the frame
    JFrame f = new JFrame("Ping Pong Green Table");

 // Ensure that the window can be closed
 // by pressing a little cross in the corner
    f.setDefaultCloseOperation(
                              WindowConstants.EXIT_ON_CLOSE);
    PingPongGreenTable table = new PingPongGreenTable();
    table.addPaneltoFrame(f.getContentPane());

 // Set the frame's size and make it visible
    f.setBounds(0,0,TABLE_WIDTH+5, TABLE_HEIGHT+40);
    f.setVisible(true);
 }
}
```

Class `PingPongGreenTable` (part 3 of 3)

I've added some more `final` variables to the interface `GameConstants`, and you should be able to guess what they are for just by looking at the variable names.

```java
package screens;
/**
 * This interface contains all definitions of the
 * final variables that are used in the game
 */
public interface GameConstants {
  // Size of the ping pong table
  public final int TABLE_WIDTH =  320;
  public final int TABLE_HEIGHT = 220;
  public final int TABLE_TOP = 12;
  public final int TABLE_BOTTOM = 180;

  // Ball movement increment in pixels
  public final int BALL_INCREMENT = 4;

  // Maximum and minimum allowed ball coordinates
  public final int BALL_MIN_X = 1+ BALL_INCREMENT;
  public final int BALL_MIN_Y = 1 + BALL_INCREMENT;
  public final int BALL_MAX_X =
                          TABLE_WIDTH - BALL_INCREMENT;
  public final int BALL_MAX_Y =
                          TABLE_HEIGHT - BALL_INCREMENT;

  // Starting coordinates of the ball
  public final int BALL_START_X = TABLE_WIDTH/2;
  public final int BALL_START_Y = TABLE_HEIGHT/2;

  //Rackets' sizes, positions and movement increments
  public final int KID_RACKET_X = 300;
  public final int KID_RACKET_Y_START = 100;
  public final int COMPUTER_RACKET_X = 15;
  public final int COMPUTER_RACKET_Y_START = 100;
  public final int RACKET_INCREMENT = 2;
  public final int RACKET_LENGTH = 30;
  public final int RACKET_WIDTH = 5;

  public final int WINNING_SCORE = 21;

//Slow down fast computers - change the value if needed
  public final int SLEEP_TIME = 10; //time in miliseconds

}
```

Below are the highlights of the changes I've made in the class `PingPongGameEngine`:

- ✓ I have removed the interface `MouseListener` and all its methods, because we're not processing mouse clicks anymore. `MouseMotionListener` will take care of all mouse movements.
- ✓ This class now implements `Runnable` interface, and you can find decision-making code in the method `run()`. Look at the constructor – I create and start a new thread there. The method `run()` applies game strategy rules in several steps,

and all these steps are programmed inside the `if` statement `if(ballServed)`. It's a short version of `if(ballServed==true)`.

- ✓ Please note the use of conditional `if` statement that assigns a value to the variable `canBounce` in step 1. Depending on the highlighted expression, this variable will get the value of either `true`, or `false`.
- ✓ The class implements `KeyListener` interface, and the method `keyPressed()` checks what letter was keyed in to start/quit the game, or to serve the ball. The code of this method allows the user to type both capital and small letters, for example `N` and `n`.
- ✓ I've added several `private` methods like `displayScore()`, `kidServe()` and `isBallOnTheTable()`. These methods are declared private because they are used within this class only, and other classes do not even have to know about them. This is an example of *encapsulation* in action.
- ✓ Some computers are too fast, and this makes the ball movements difficult to control. That's why I've slowed the game down by calling a method `Thread.sleep()`. A static method `sleep()` will pause this particular thread for a number of milliseconds given as an argument of this method.
- ✓ To add a little fun to the game, when the kid's racket hits the ball it moves diagonally. That's why code changes not only the `X` coordinate of the ball, but `Y` as well.

```
package engine;

import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import screens.*;
/**
 * This class is a mouse and keyboard listener.
 * It calculates ball and racket movements,
 * changes their coordinates.
 */
public class PingPongGameEngine implements Runnable,
       MouseMotionListener, KeyListener, GameConstants{

  private PingPongGreenTable table;//reference to table
  private int kidRacket_Y = KID_RACKET_Y_START;
  private int computerRacket_Y=COMPUTER_RACKET_Y_START;
  private int kidScore;
  private int computerScore;
  private int ballX;    // ball's X position
  private int ballY;    // ball's Y position
  private boolean movingLeft = true;
  private boolean ballServed = false;

  //Value in pixels of the vertical ball movement
  private int verticalSlide;

   // Constructor. Stores a reference to the table
  public PingPongGameEngine(
                              PingPongGreenTable greenTable){
   table = greenTable;
   Thread worker = new Thread(this);
   worker.start();
  }
 // Methods required by MouseMotionListener
 // interface (some of them are empty, but must be
 // included in the class anyway)

 public void mouseDragged(MouseEvent e) {
 }
```

Class `PingPongGameEngine` (part 1 of 5)

```
public void mouseMoved(MouseEvent e) {

 int mouse_Y = e.getY();

  // If a mouse is above the kid's racket
  // and the racket did not go over the table top
  // move it up, otherwise move it down
  if (mouse_Y<kidRacket_Y && kidRacket_Y>TABLE_TOP){
     kidRacket_Y -= RACKET_INCREMENT;
  }else if (kidRacket_Y < TABLE_BOTTOM) {
     kidRacket_Y += RACKET_INCREMENT;
  }

  // Set the new position of the racket  on the table
  table.setKidRacket_Y(kidRacket_Y);
}

  // Methods required by KeyListener interface
public void keyPressed(KeyEvent e){
  char key = e.getKeyChar();
  if ('n' == key || 'N' == key){
     startNewGame();
  } else if ('q' == key || 'Q' == key){
     endGame();
  } else if ('s' == key || 'S' == key){
     kidServe();
  }
}

public void keyReleased(KeyEvent e){}

public void keyTyped(KeyEvent e){}

// Start a new Game
public void startNewGame(){
  computerScore=0;
  kidScore=0;
  table.setMessageText("Score Computer: 0  Kid: 0");
  kidServe();
}

// End the game
public void endGame(){
  System.exit(0);
}
```

Class `PingPongGameEngine` (part 2 of 5)

```
// Method run() is required by Runnable interface
public void run(){

boolean canBounce=false;
while (true) {

 if(ballServed){ // if ball is moving

  //Step 1. Is ball moving o the left?
  if ( movingLeft && ballX > BALL_MIN_X){
    canBounce = (ballY >= computerRacket_Y &&
       ballY < (computerRacket_Y + RACKET_LENGTH)?
                                             true: false);
    ballX-=BALL_INCREMENT;

    // Add up or down slide to any left/right ball
    // movement
    ballY-=verticalSlide;

    table.setBallPosition(ballX,ballY);
    // Can bounce?
    if (ballX <= COMPUTER_RACKET_X  && canBounce){
      movingLeft=false;
    }
  }

  // Step 2. Is ball moving to the right?
  if ( !movingLeft && ballX <= BALL_MAX_X){
     canBounce = (ballY >= kidRacket_Y && ballY <
          (kidRacket_Y + RACKET_LENGTH)?true:false);

     ballX+=BALL_INCREMENT;
     table.setBallPosition(ballX,ballY);
     // Can bounce?
     if (ballX >= KID_RACKET_X && canBounce){
        movingLeft=true;
     }
  }

  // Step 3. Move computer's racket up or down
  //         to block the ball
```

Class `PingPongGameEngine` (part 3 of 5)

```
  if (computerRacket_Y < ballY
                && computerRacket_Y < TABLE_BOTTOM){
      computerRacket_Y +=RACKET_INCREMENT;
  }else if (computerRacket_Y > TABLE_TOP){
      computerRacket_Y -=RACKET_INCREMENT;
  }
  table.setComputerRacket_Y(computerRacket_Y);

  // Step 4. Sleep a little
  try {
      Thread.sleep(SLEEP_TIME);
  } catch (InterruptedException e) {
      e.printStackTrace();
  }

  // Step 5. Update the score if the ball is in the
  // green area but is not moving
   if (isBallOnTheTable()){
      if (ballX > BALL_MAX_X ){
        computerScore++;
        displayScore();
      }else if (ballX < BALL_MIN_X){
        kidScore++;
        displayScore();
      }
    }
   } // End if ballServed
  } // End while
 }// End run()

// Serve from the current position of the kid's racket
 private void kidServe(){

   ballServed = true;
   ballX = KID_RACKET_X-1;
   ballY=kidRacket_Y;

   if (ballY > TABLE_HEIGHT/2){
      verticalSlide=-1;
   }else{
      verticalSlide=1;
   }

   table.setBallPosition(ballX,ballY);
   table.setKidRacket_Y(kidRacket_Y);
 }
```

Class `PingPongGameEngine` (part 4 of 5)

```
  private void displayScore(){
    ballServed = false;

    if (computerScore ==WINNING_SCORE){
      table.setMessageText("Computer won! " +
                      computerScore +  ":" + kidScore);
    }else if (kidScore ==WINNING_SCORE){
      table.setMessageText("You won! "+ kidScore +
                                  ":" + computerScore);
    }else{
      table.setMessageText("Computer: "+ computerScore
                               +  " Kid: " + kidScore);
    }
  }

 // checks if ball did not cross the top or bottom
  // borders of the table
  private boolean isBallOnTheTable(){
    if (ballY >= BALL_MIN_Y && ballY <= BALL_MAX_Y){
       return true;
    }else {
       return false;
    }
  }
 }
```

Class `PingPongGameEngine` (part 5 of 5)

Congratulations! You've completed your second game. Compile the classes and play the game. After you feel more comfortable with the code, try to modify it – I'm sure you'll have some ideas of how to make this game better.

What to Read Next on Game Programming

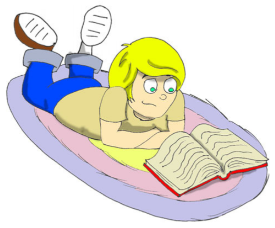
1. CodeRally is an IBM sponsored Java-based, real-time programming game based on the Eclipse platform. It allows users unfamiliar with Java to easily compete while they learn the Java language. Players develop a rally car and make decisions about when to speed up, turn, or slow down based on the location of other players or checkpoints, their current fuel level, and other factors.

http://www.alphaworks.ibm.com/tech/codeRally

2. Robocode is a fun programming game that teaches Java by letting you create Java Robots.

http://www.alphaworks.ibm.com/tech/robocode

Additional Reading

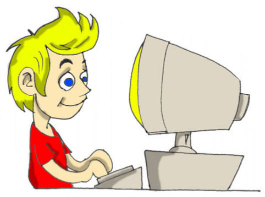


Java Threads Tutorial:
http://java.sun.com/docs/books/tutorial/essential/threads/

Introduction to Java Threads:
http://www-106.ibm.com/developerworks/edu/j-dw-javathread-i.html

Class java.awt.Graphics:
http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Graphics.html

Practice



1. The class `PingPongGameEngine` sets the coordinates of the white point using the code like this:
 `table.point.x = e.getX();.`

In the class `PingPongGreenTable` make the variable `point` private and add a public method
`setPointCoordinates(int x, int y).`

Change the code of the engine class to use this method.

2. Our ping pong game has a bug: after a winner is announced, you can still press the key *S* on the keyboard and the game will continue. Fix this bug.

Practice for Smarty Pants

1. Try to change the values of the RACKET_INCREMENT and BALL_INCREMENT. Higher values increase the speed of racket and ball movements. Change the code to allow selection of the player's level from 1 to 10. Use selected values as ball and racket

increments.

2. When the kid's racket bounces the ball in the top part of the table, the ball moves diagonally an upward and quickly falls off the table. Modify the program to move the ball diagonally down from the top part of the table, and diagonally up from the bottom part.

Appendix A. Java Archives - JARs

Computer users pretty often need to exchange files. They could either copy files on floppy disks, CD, use e-mail, or just send the data across the network. There are special programs that can *compress* multiple files into a single *archive* file .

The size of such archive is usually smaller than combined sizes of each file, and this makes copying faster and also saves space on your disks.

Java comes with a program called `jar` that is used to archive multiple Java classes and other files into a file having the name extension `.jar`.

Internal format of jar files is the same as in a popular program called WinZip (we used it in Chapter 2).

The following tree commands illustrate the use of the `jar` tool:

To create a jar that will contain all files with extension `.class`, open this black command window, get into the folder where your classes are, and type the following command:

```
jar cvf myClasses.jar *.class
```

After the word `jar` you have to specify the `options` for this command. In the last example *c* is for creating a new archive, *v* is for displaying what goes in there, and *f* means that the file name of the new archive is provided.

Now you can copy this file to another disk or email it to your friend. To *unjar* (extract) the files from the archive `myClasses.jar`, type the following command:

```
jar xvf myClasses.jar
```

All files will be extracted into the current directory. In this example the option *x* is for extracting files from the archive.

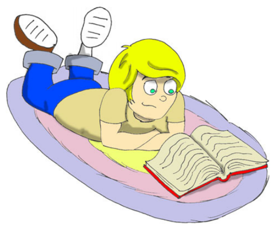
If you just want to see the content of the jar without extracting the files, use the next command where *t* is for tables of contents:

```
jar tvf myClasses.jar
```

Actually, I prefer using the program WinZip to see what's in the jar file.

In many cases real-world Java applications consist of multiple classes that live in jars. Even though there are many other options that could be used with the `jar` program, three examples from this chapter is all you need to know for most of your future projects.

Additional Reading



Java Archive Tool:
http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/jar.html

Appendix B. Eclipse Tips

Eclipse has many little convenient commands that make Java programming a little faster. I've listed some of the useful Eclipse tips here, but I'm sure you'll find more of them when you start using this tool.

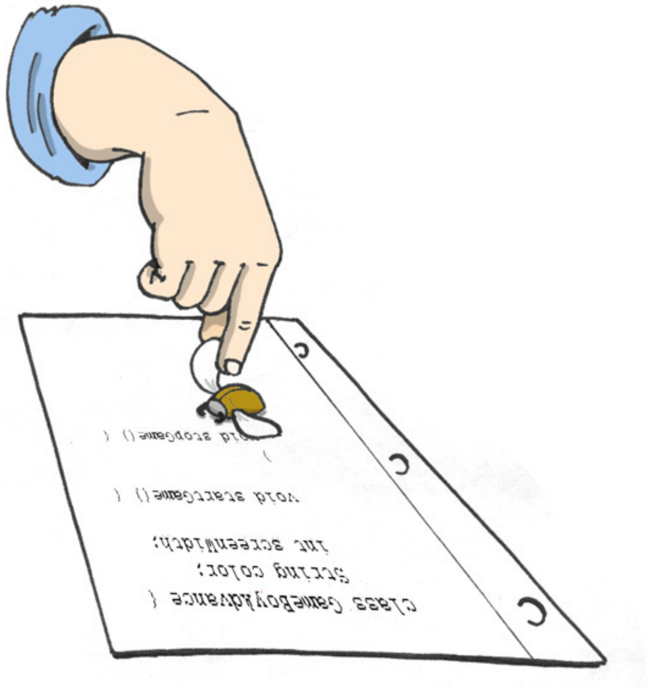
- ✓ If you see a little asterisk in the tab with the class, this means that the class has some unsaved code changes.
- ✓ Highlight the name of the class or a method that is used in your code and press the button *F3* on your keyboard. This will take you to the line where this class or method was declared.
- ✓ If some of the lines are marked with red error circles, move the mouse over the circle to see the error text.
- ✓ Press *Ctrl-F11* to run the last-launched program again.
- ✓ Place the cursor after a curly brace and Eclipse will mark the matching brace.
- ✓ To change the superclass when creating a new class, click on the button *Browse*, delete the class `java.lang.Object` and enter the first letter of the class you'd like to use. You'll see a list of available classes to choose from.
- ✓ To copy a class from one package to another, select the class and press *Ctrl-C*. Select the destination package and press *Ctrl-V*.
- ✓ To rename a class, a variable or a method, right-click on it and select *Refactor* and *Rename* from the popup menu. This will rename every occurrence of this name.

- ✓ If your project needs some external jars, right-click on the project name, select *Properties*, *Java Build Path* and press the button *Add External Jars*.

Eclipse Debugger

The rumor has it, that about 40 years ago, when computers were large and would not even fit in your room, all of a sudden one of the programs started giving wrong results. All these troubles were caused by a small bug that was sitting inside the computer somewhere in the wires. When people removed the bug, the program started working properly again. Since then, *to debug a program* means to find out why it does not give the expected results.

Do not confuse bugs with the compilation errors. Say for example, instead of multiplying the variable by 2, you'll multiply it by 22. This typo will not generate any compilation errors, but the result will be incorrect. Debuggers allow you to step through a running program one line at a time, and you can see and change values of all variables at each point of the program execution.

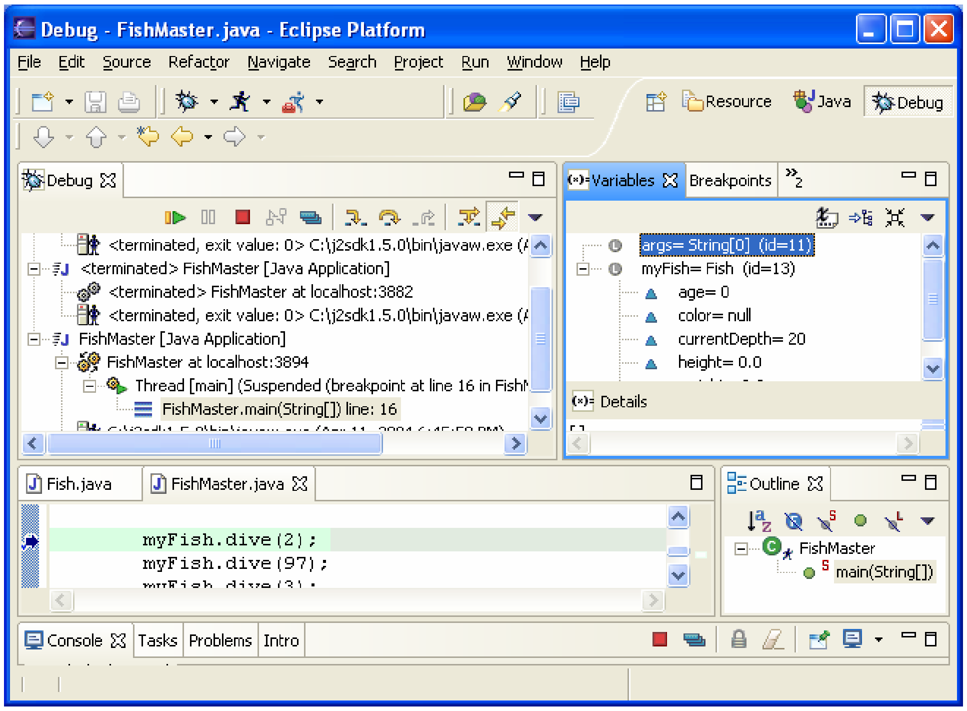


I'll show you how to use Eclipse debugger using the FishMaster program from Chapter 4.

A *breakpoint* is a line in the code where you'd like program to pause so you can see/change current values of the variables, and some other run-time information. To set a breakpoint just double click on the gray area to the left of the line where you want a program to stop. Let's do it in the FishMaster class on the line `myFish.dive(2)`. You'll see a round bullet on this line which is a breakpoint. Now, select the menus *Run*, *Debug…*. Select the application `FishMaster` and press the button *Debug*.

FishMaster will start running *in the debug mode*, and as soon as the program reaches the line `myFish.dive(2)`, it will stop and will wait for your further instructions.

You will see a window similar to the next one.


Debug - FishMaster.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Resource Java Debug
Debug
Variables Breakpoints
Details
Fish.java FishMaster.java
Outline
Console Tasks Problems Intro

<terminated, exit value: 0> C:\j2sdk1.5.0\bin\javaw.exe
<terminated> FishMaster [Java Application]
<terminated> FishMaster at localhost:3882
<terminated, exit value: 0> C:\j2sdk1.5.0\bin\javaw.exe
FishMaster [Java Application]
FishMaster at localhost:3894
Thread [main] (Suspended (breakpoint at line 16 in FishM
FishMaster.main(String[]) line: 16

args= String[0] (id=11)
myFish= Fish (id=13)
age= 0
color= null
currentDepth= 20
height= 0.0

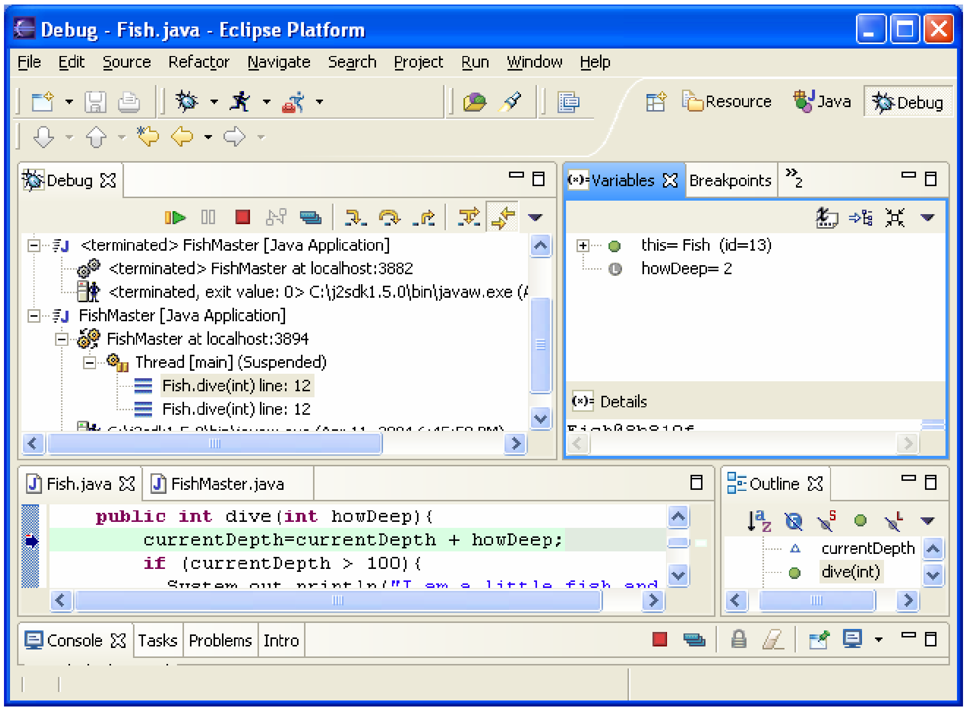
```
        myFish.dive(2);
        myFish.dive(97);
```

FishMaster
main(String[])

In the left bottom part of the debug perspective, you see that the line with the breakpoint is highlighted. The blue arrow points at the line that is about to be executed. On the right side (in the *Variables* view) click on the little plus sign by the variable `myFish`. Since this variable points at the object `Fish`, you will see all member variables of this class and their current values, for example `currentDepth=20`.

The arrows in the top left area allow you to continue execution of the program in different modes. The first yellow arrow means *step into* the method. If you press this arrow (or *F5*), you'll find yourself inside the method `dive()`. The window changes and you see the values of the argument `howDeep=2` as in the next screenshot. Click on the little plus by the word `this` to see what are the current values of member variables of this object.

To change the value of the variable, right-click on the variable and enter the new value. This can help when you are not sure why the program does not work correctly and would like to play *what if* game.



```
<terminated> FishMaster [Java Application]
    <terminated> FishMaster at localhost:3882
    <terminated, exit value: 0> C:\j2sdk1.5.0\bin\javaw.exe (
FishMaster [Java Application]
    FishMaster at localhost:3894
        Thread [main] (Suspended)
            Fish.dive(int) line: 12
            Fish.dive(int) line: 12
```

```
this= Fish (id=13)
howDeep= 2
```

```
public int dive(int howDeep){
    currentDepth=currentDepth + howDeep;
    if (currentDepth > 100){
```

```
currentDepth
dive(int)
```

To continue execution one line at a time, keep pressing the next arrow step over (or the button F6).

If you want to continue program in the fast mode, press a small green triangle or the button F8.

To remove the breakpoint just double-click on the little round bullet and it'll disappear. I like using debugger even if my program does not have a bug – it helps me better understand what exactly happens inside the running program.

Where to put a breakpoint? If you have an idea which method gives you problems, put it right before suspicious line. If you are not sure, just put in the first line of the method `main()` and slowly walk through the program.

Appendix C. How to Publish a Web Page

Internet pages consist of HTML files, images, sound files, etc.

HTML was briefly mentioned in Chapter 7, but if you are planning to become a Web designer, you should spend more time learning HTML, and one of the good places to start is a Web page www.w3chools.com. Actually, there are many Web sites and programs that allow you create a Web page in a several minutes without even knowing how it's being done. These programs will generate HTML anyway, but they just hide this from you. But if you've mastered this book, I declare you a **Junior Java Programmer** (I'm not kidding!), and learning HTML for you is a piece of cake.

To develop a Web page, you usually create one or more HTML files on your computer's disk, but the problem is that your computer *is not visible* to other Internet users. That's why, when the page is finished, you need to copy (*upload*) these files to a place that everybody can see. Such place is a disk located in the computer of the company that is your *Internet Service Provider (ISP).*

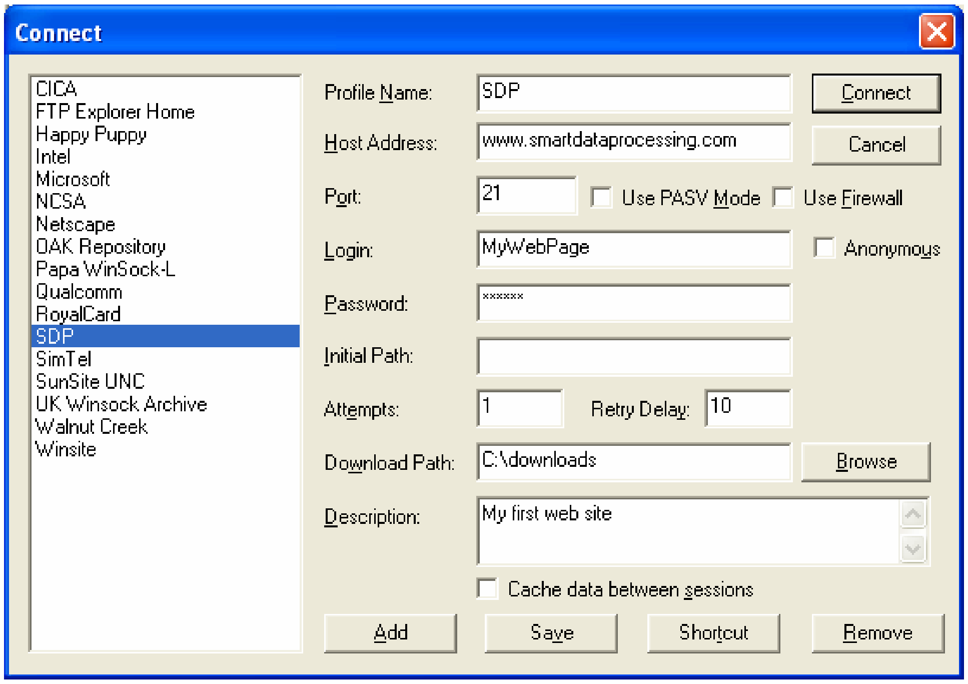
First of all, you need to have your own folder on your ISP's computer. Contact your ISP by phone or e-mail, saying that you created an HTML page and want to publish it. They will usually respond with the following information:

- The network name of their computer (*host machine*).
- Name of the folder on their computer where they allow you to keep your files.
- A Web address (*URL*) of your new page - you will be giving it to anyone who is interested in seeing your page.
- The user id and the password that you'll need to upload new or modify old files.

These days, most of the ISP's will give you at least 10MB of space on their disk for free, which is more than enough for most of people.

Now you will need a program that will allow you to copy files from your machine to your ISP's computer. Copying files from your machine onto the Internet’s computer is called *uploading*, and copying files from the Internet to your machine is called *downloading*. You can upload or download files using so-called *FTP client program*.

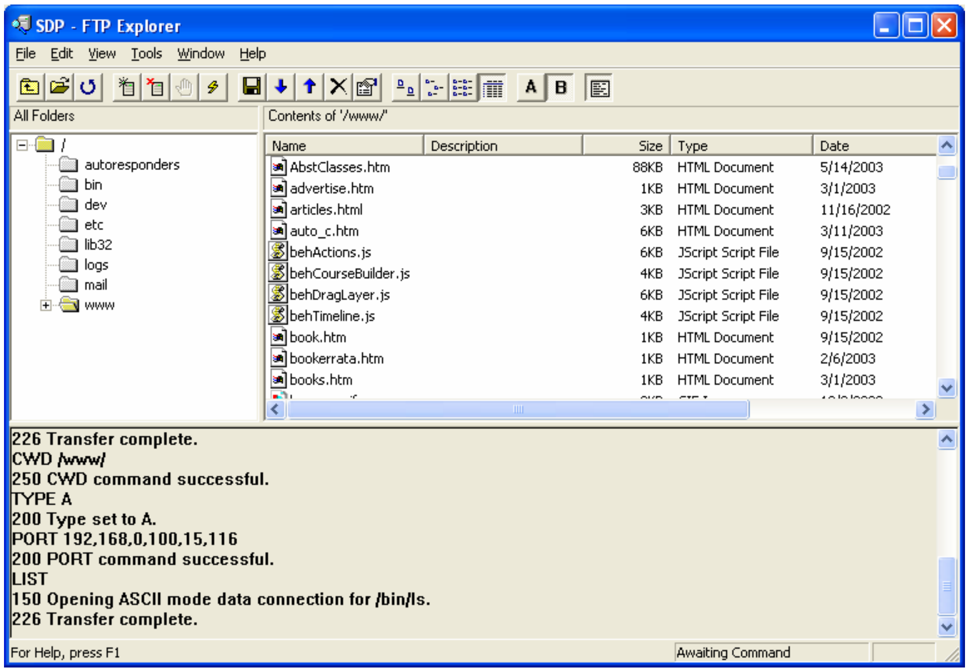
One of the simple and easy to use FTP clients is *FTP Explorer* and you can download it from www.ftpx.com. Install this program and add your ISP machine to connection list of your FTP client - start FTP Explorer and the first window you see is a connection screen. You can also click on the *Connection* item in the *Tools* menu.



Profile Name: SDP
Host Address: www.smartdataprocessing.com
Port: 21 Use PASV Mode Use Firewall
Login: MyWebPage Anonymous
Password: ******
Initial Path:
Attempts: 1 Retry Delay: 10
Download Path: C:\downloads
Description: My first web site
Cache data between sessions

Press the button *Add*, and enter the host, login id and the password that you've got from your ISP. Just type in the name of your ISP in the *Profile Name* field. If you did everything right, you will see your new connection profile in the list of available FTP servers. Press the button *Connect* and you’ll see the folders on your ISP's machine. Find your folder over there and start the uploading process that is described next.

The toolbar has two blue arrows. The arrow that points up is for uploading. Press this arrow, and you will see a standard window that will allow you to get into the folder with your HTML files. Select the files that you are planning to upload and press the button *Open*. In a couple of seconds you will see these files on your ISP's machine.



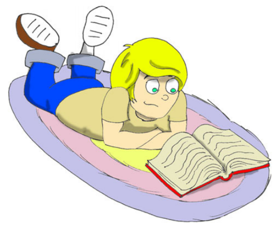
Name	Description	Size	Type	Date
AbstClasses.htm		88KB	HTML Document	5/14/2003
advertise.htm		1KB	HTML Document	3/1/2003
articles.html		3KB	HTML Document	11/16/2002
auto_c.htm		6KB	HTML Document	3/11/2003
behActions.js		6KB	JScript Script File	9/15/2002
behCourseBuilder.js		4KB	JScript Script File	9/15/2002
behDragLayer.js		6KB	JScript Script File	9/15/2002
behTimeline.js		4KB	JScript Script File	9/15/2002
book.htm		1KB	HTML Document	9/15/2002
bookerrata.htm		1KB	HTML Document	2/6/2003
books.htm		1KB	HTML Document	3/1/2003

```
226 Transfer complete.
CWD /www/
250 CWD command successful.
TYPE A
200 Type set to A.
PORT 192,168,0,100,15,116
200 PORT command successful.
LIST
150 Opening ASCII mode data connection for /bin/ls.
226 Transfer complete.
```

Pay attention to the bottom part of this window to make sure that there were no problems during uploading.
Name the main file of your page `index.html`. This way your URL will be shorter and people will not need to type the file name at the end of your URL. For example, if the name of the folder in the ISP disk is www.xyz.com/~David, and the main file of your Web page is `myMainPage.html`, the address of your Web page would be www.xyz.com/~David/myMainPage.html. But if the name of the main page is `index.html`, the URL of your page is shorter – www.xyz.com/~David. From now on, everyone who knows this URL, will be able to see your page online. If, later on, you decide to modify this Web page, you will repeat the same process again - make corrections on your disk, and after that just upload it, to replace the old files with the new ones.

If you decide to become a Web designer, the next language to learn is JavaScript. This language is a lot simpler than Java and will allow you to make your Web pages fancier.

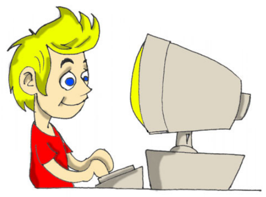
Additional Reading



1. Webmonkey for Kids:
http://hotwired.lycos.com/webmonkey/kids/

2.The World Wide Web
http://www.w3schools.com/html/html_www.asp

Practice



Create a Web page and publish the Tic-Tac-Toe game from Chapter 7. To start, just upload to your Web page files `TicTacToe.html` and `TicTacToe.class`.

The End

Index