# Java Programming Book 1

# Java Programming Book 1

## Student Workbook

**itcourseware**
TRAINING MATERIALS FOR IT PROFESSIONALS

*Java Programming - Book 1*

# Contents

# Chapter 1 - Course Introduction

# Course Objectives

❋     Write stand-alone applications using the Java language.

❋     Accurately implement Object-Oriented concepts using Java features such as classes, interfaces and references.

❋     Create well-scoped classes using packages.

❋     Write programs which both handle and create exceptions.

❋     Create and run unit tests with JUnit.

# Course Overview

❋    **Audience:**  This course is designed for programmers who want to move into the Java language, but have no C, C++, or C# background.

❋    **Prerequisites:**  This course assumes that the student is a programmer learning Java.

❋    **Classroom Environment**:

   ➢    One  Java development environment per student.

# Using the Workbook

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. **Hands On** sections provide opportunities for practical application of key concepts. **Try It** and **Investigate** sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Topics are organized into first (✳), second (➢) and third (▪) level points.

Callout boxes point out important parts of the example code.
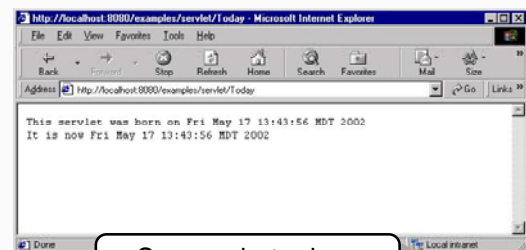
Screen shots show examples of what you should see in class.

Pages are numbered sequentially throughout the book, making lookup easy.

# Suggested References

Arnold, Ken, James Gosling, and David Holmes. 2013. *The Java Programming Language.* Addison-Wesley, Reading, MA. ISBN 978-0132761680.

Bloch, Joshua. 2008. *Effective Java (2nd Edition)*. Addison-Wesley, Reading, MA. ISBN 978-0321356680.

Cadenhead, Rogers. 2012. *Sams Teach Yourself Java in 21 Days (6th Edition)*. Sams, Indianapolis, IN. ISBN 978-0672335747.

Eckel, Bruce. 2006. *Thinking in Java (4th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 978-0131872486.

Horstmann, Cay and Gary Cornell. 2012. *Core Java 2, Volume I: Fundamentals (9th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 978-0137081899.

Horstmann, Cay and Gary Cornell. 2013. *Core Java 2, Volume II: Advanced Features (9th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 978-0137081608.

Schildt, Herbert. 2011. *Java, A Beginner's Guide (5th Edition)*. McGraw Hill, New York, NY. ISBN 978-0071606325.

Schildt, Herbert. 2011. *Java The Complete Reference (8th Edition)*. McGraw Hill, New York, NY. ISBN 978-0070435926.

Sierra, Kathy and Bert Bates. 2005. *Head First Java (2nd Edition)*. O'Reilly & Associates, Sebastopol, CA. ISBN 978-0596009205.

*http://stackoverflow.com/questions/tagged/java*
*http://www.javaworld.com*
*http://www.javaranch.com*
*http://www.oracle.com/technetwork/java*

# Chapter 2 - Getting Started with Java

## Objectives

❇    Define Java terms such as JRE,
     JSDK, and JVM

❇    Write, compile, and run a Java
     program.

# What is Java?

❋ Java is an Object-Oriented Programming language with extensive class libraries.

    ➢ You need a Java Runtime Environment (JRE), consisting of a Java *Virtual Machine* (VM) and a copy of the Java API libraries, to run a Java program.

    ➢ You need the Java Software Development Kit (JSDK), including a Java VM, a copy of the Java API libraries, and a Java compiler, to create a Java program.

❋ Write Once, Run Anywhere.

    ➢ Code written on any platform, with any Java compiler, can be run on any platform with any Java Virtual Machine (interpreter).

❋ Java borrowed the best of several programming languages, including C++, C#, Objective C, Cedar/Mesa, Smalltalk, Eiffel, and Perl.

The Java white paper lists several design goals:

**Simple** — The language syntax is familiar, very much like C and C++, but many of the really nasty things have been removed.

**Secure** — Compile time and runtime support for security.

**Distributed** — The Java API library includes the **java.net** package.

**Object-Oriented** — Designed from the beginning to be strictly object-oriented. The Java API libraries include a large number of classes arranged in packages (like class libraries). There are no functions or global data outside of classes, e.g., **main**() or C++ friend functions.

**Robust** — Strongly typed — stronger type checking than C++. Compile-time checking for type mismatches. Simplified memory management. No pointers. Exception handling.

**Portable** — Java code is compiled into architecture-neutral bytecode. No implementation-dependent aspects in the language (e.g., the **int** type is 32 bits regardless of the platform word size).

**Interpreted** — The  bytecode is interpreted on any platform that implements the Java Virtual Machine.

**Multithreaded** — Language and library support for multiple threads of control.

**Dynamic** — Classes are loaded as needed, locally or across the net. Runtime type information is built in.

**High performance** — Interpreted, so not as fast as C in execution speed. Just In Time (JIT) compilers make Java programs almost as fast as C. Automatic Garbage Collection helps ensure needed memory is available.

# How to Get Java

✳  You can download the Java Software Development Kit for free from Oracle.

> ➢  Oracle has ports for Solaris, Windows, OS X, and Linux platforms, and publishes links to other ports.

> ➢  The Oracle Java web site has many other resources, including the JSDK documentation, many demo programs, the FAQ, Java language spec, the white paper, etc.

✳  Open source development environments like NetBeans and Eclipse are freely available for download.

# A First Java Program

## Hands On:

❊ Type the following program into a text editor:

```
package examples;

public class Hello {
    public static void main(String[] args) {
        System.out.print("Hello, ");
        if (args.length == 0)
            System.out.println("World!");
        else
            System.out.println(args[0] + "!");
    }
}
```

➢ We are defining a top-level class named **Hello**.

❊ The name of the file must be *ClassName.java*; e.g., *Hello.java*.

➢ Case is important.

❊ The file should reside in a directory that corresponds to the package name.

The classic first program in Java is just "Hello, World!":

examples/HelloWorld.java

```java
package examples;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Ours is very similar, but a little more interesting.

# Compiling and Interpreting Applications

✷    The steps to compile and interpret a Java application:

1.    Create the source file with a text editor.

2.    Compile the source into bytecode:

```
javac examples/Hello.java
```

3.    Interpret the bytecode:

```
java examples/Hello
```

or

```
java examples/Hello  Bob
```

**Environment Variables**

The easiest way to run **java** and **javac** is to have their location in your **PATH** environment variable.

On Linux:
```
PATH=$PATH:/JAVA_HOME/bin
```

On Windows:
```
PATH=%PATH%;c:\JAVA_HOME\bin
```

Where *JAVA_HOME* is your Java installation directory.

**Stand-alone Applications**

Java programs can run as stand-alone applications. They can be started from a DOS or Linux prompt. On a Windows platform, or a graphical Linux platform, you can run a Java program that has a graphical user interface from an icon on your desktop.

*examples\Hello.java*

```
public class Hello
  public static void
    System.out.print("
    if (args.length ==
      System.out.prin
    else
```

**javac examples\Hello.java**

*examples\Hello.class*

```
feca beba 0300 2d00
0007 071a 1b00 0007 0a
0005 0a0a 0300 0b00 00
1700 000c 001f 010d 03
4c28 616a 6176 6c2f 6e
676e 293b 0156 1600 5b
```

java examples/Hello Bob

Java VM

1. Load bytecode from the file *Hello.class*

| Class Hello |
| --- |
| public void main() |

2. Interpret and run bytecode:

```
Hello.main("Bob")
```

# The JSDK Directory Structure

❋ Java should be installed in some globally-accessible directory, such as */Library/Java/JavaVirtualMachines* or *C:\Program Files\Java*.

➢ *src.zip* is an archive which contains all of the source code for the standard Java library classes; this can be expanded to view the source code.

❋ *bin/* contains the **javac** compiler and the **java** interpreter, among other tools; this directory needs to be in your **PATH** environment variable.

❋ *lib/* contains Java Archive (JAR) files used by the Java tools.

❋ *jre/* contains subdirectories with files for the Java Runtime Environment.

➢ *jre/bin/* contains copies of the tools used at runtime and libraries that they use.

➢ *jre/lib/* contains resource files for fonts, etc., and the standard Java library classfiles (in *rt.jar*).

❋ *include/* contains C header files for use with the Java Native Interface.

Oracle provides several Java tools with the JSDK. Here a few of the ones used most:

**javac** — the source code compiler.

> **javac** produces architecture-neutral "bytecode." The compiler produces a bytecode file *Classname.class*; e.g., *javac HelloWorld.java* produces *HelloWorld.class*.

**java** — the bytecode interpreter.

> The "virtual machine" (**java**) interprets bytecode for specific architectures. This is how stand-alone (non-applet) Java programs are executed.

```
java Classname
```

e.g.,

```
java HelloWorld
```

> Do not include the *.class* extension.

**appletviewer** — a mini-browser for testing applets.

> **appletviewer** reads an HTML file containing an <**applet** ...> tag, and loads and runs that applet class file.

**javap** — a "disassembler" for bytecode.

**jdb** — the bytecode debugger.

**javadoc** — a utility to generate *.html* files which document the methods and hierarchy of your classes. For **javadoc** to work, you must:

> 1. Add **javadoc** comments to your *.java* file:

```
/** This is a Javadoc comment */
```

> 2. Run **javadoc** on your *.java* file:

```
javadoc HelloWorld.java
```

**jar** — a utility to create JAR files (similar to ZIP files).

# LABS

❶     Write a Java application called **MyName** that prints out your name, and compile it using **javac**. List your directory. What was created? Run your program.
(Solutions: *MyName.java*)

❷     What happens if the name of the *.java* file is different from the class name contained in it? Copy *MyName.java* to *Name.java*. What messages do you get from the compiler?
(Solution: *Name.txt*)

❸     Try disassembling one of your *.class* files with **javap -c** (leave off the *.class* extension). Try **javap -help**.
(Solution: *MyName_javap_output*)

# Chapter 3 - Eclipse

## Objectives

❋     Install and configure Eclipse.

❋     Create and manage Java projects.

❋     Write, compile, run, and debug Java
      programs in Eclipse.

# Introduction to Eclipse

❋ Eclipse is the most popular Java Integrated Development Environment (IDE) available today.

➢ It is freely available for download from the Eclipse Foundation.

❋ IBM started the Eclipse project in April of 1999 with the goal of "eclipsing" the dominance of Microsoft Visual Studio within the integrated development environment (IDE) space.

➢ IBM donated the initial Eclipse code base to the open source community in November of 2001 — just one month after version 1.0 was released.

❋ Eclipse has become much more than a pure Java, open source IDE; it is a framework on which companies can develop their own tools.

➢ Users can combine tools from different vendors to accomplish their goals.

❋ The *Eclipse Project* is made up of multiple sub-projects, including:

➢ The *Eclipse Platform* is the core IDE that most people associate with the term "Eclipse."

▪ It is extensible: you can plug-in tools from various vendors to customize it to your needs.

➢ The *Java Development Tools (JDT) project* provides the necessary plug-ins to support development of Java applications.

▪ These tools include the incremental compiler, the debugger, and the editor.

➢ The *Plug-in Development Environment (PDE) project* provides tools that allow you to build your own plug-ins for Eclipse.

Eclipse has had numerous releases since its inception:

Eclipse 1.0 — October, 2001
Eclipse 2.0 — June, 2002
Eclipse 3.0 — June, 2004
Eclipse 3.1 — June, 2005
Eclipse 3.2 (Callisto) — June, 2006
Eclipse 3.3 (Europa) — June, 2007
Eclipse 3.4 (Ganymede) — June, 2008
Eclipse 3.5 (Galileo) — June, 2009
Eclipse 3.6 (Helios) — June, 2010
Eclipse 3.7 (Indigo) — June, 2011
Eclipse 4.2 (Juno) — June, 2012
Eclipse 4.3 (Kepler) — June, 2013
Eclipse 4.4 (Luna) — June, 2014
Eclipse 4.5 (Mars) — June, 2015

# Installing Eclipse

❋ Download the most recent Eclipse release build from *http://www.eclipse.org/ downloads/*.

    ➢ You will find versions supporting Windows, Linux, and Mac OS X.

    ➢ The file you download will be a *.zip* file (or *tar.gz*), rather than an install program.

    ➢ The download does not include a Java Runtime Environment.

        ▪ You must install a Java 6 or newer JRE for Eclipse to work.

        ▪ You do not need the full Java Software Development Kit unless you want to be able to step through the library source code during debug.

❋ Extract the downloaded file to your local drive.

    ➢ An *eclipse* directory will be created with several subdirectories.

❋ That is all there is to it — you have successfully downloaded and installed Eclipse!
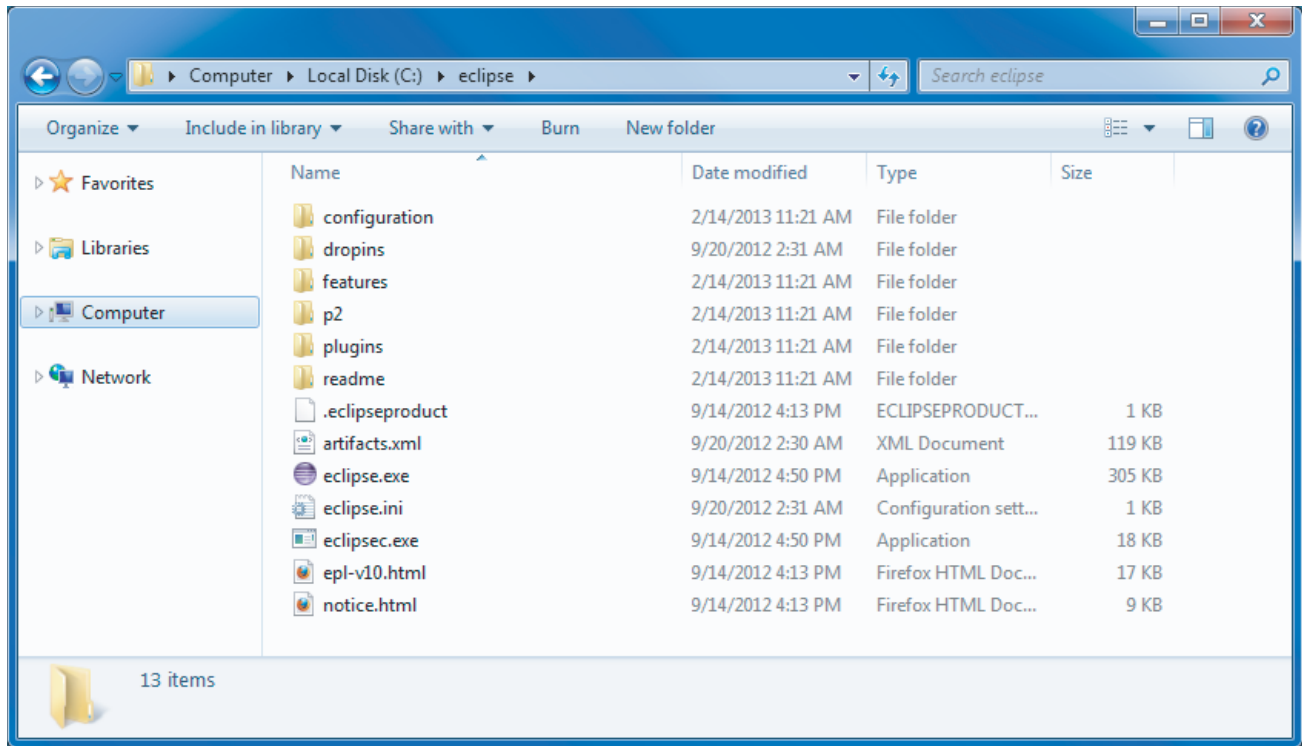
# Running Eclipse for the First Time

❋ Run the Eclipse executable to start the Eclipse IDE.

➢ The executable is found in the *eclipse* directory and is called *eclipse.exe* on Windows and *eclipse* on Linux.

➢ Eclipse will use the first Java VM it finds in your **PATH** when it runs.

▪ To explicitly set the VM, pass the **-vm** argument to the Eclipse executable:

```
eclipse –vm c:\jre\bin\javaw.exe
```

❋ The first time you run Eclipse, it will ask you to select a workspace location.

➢ A *workspace* is a directory where your projects will be stored.

➢ This value can be passed in on the command-line with the **-data** argument:

```
eclipse –data c:\code\myworkspace
```

❋ After you have selected a workspace, you will see a welcome screen that provides links to tutorials, sample applications, overview topics, and information on what's new in the current version of Eclipse.

➢ You can skip this introductory content by choosing the Workbench link on the right of the screen or by clicking the x next to the word Welcome in the upper left corner of the screen.

▪ To get back to the welcome screen in the future, you can choose the Help→Welcome menu item.
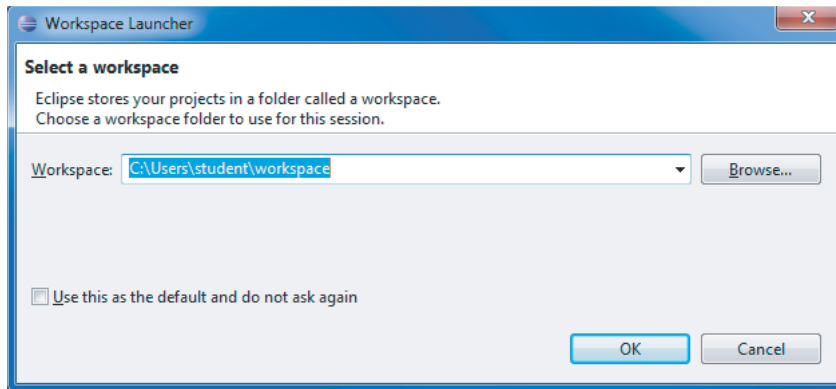
Click here to switch
to the workbench.
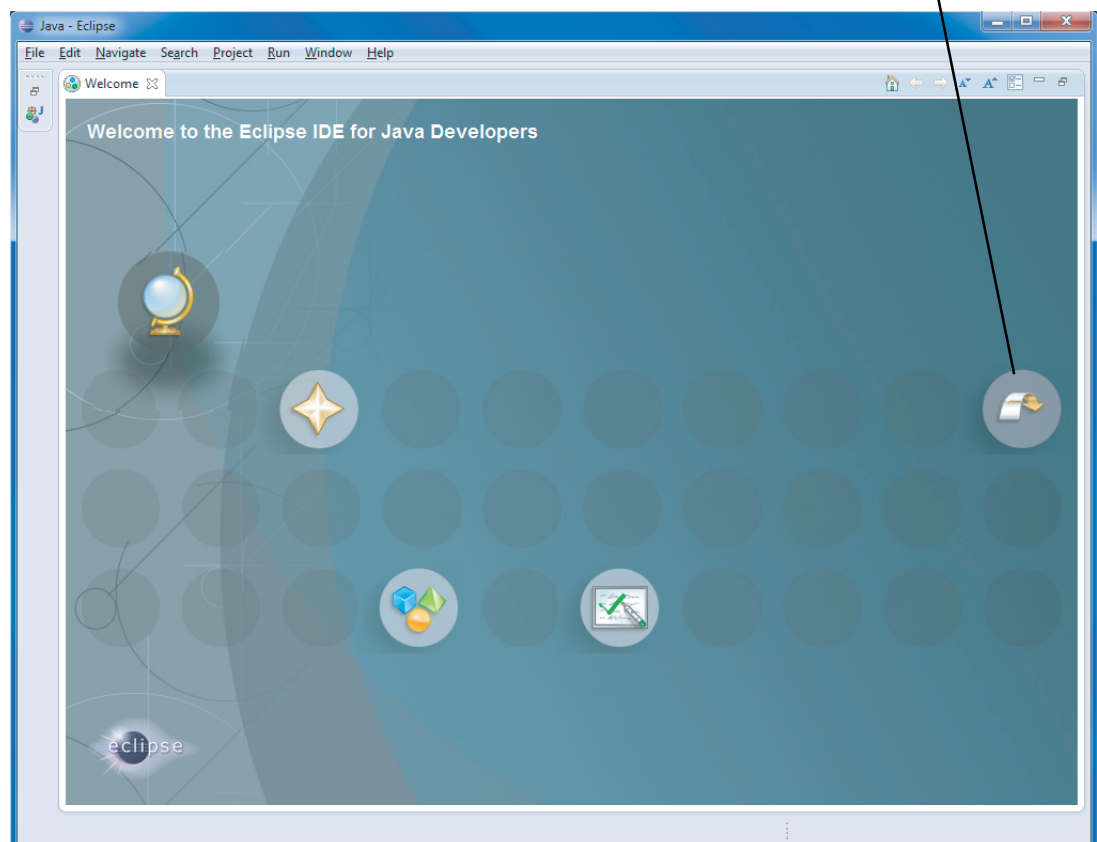
For more command-line options, see:

*http://help.eclipse.org/juno/*

*index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fmisc%2Fruntime-options.html*

# Editors, Views, and Perspectives

✻ You use an editor to enter text into Eclipse.

    ➢ The JDT Java Editor provides syntax coloring, code completion, and code formatting, among other things.

    ➢ A simple text editor is also included as part of the Eclipse Platform.

    ➢ Other editors, such as an HTML editor, can be added as plug-ins.

    ➢ Multiple editors can be open at once and they will appear as stacked instances with individual tabs for selection.

✻ *Views* display information about an object; they typically supplement the data that is visible in the current editor.

    ➢ The JDT provides a Package Explorer View and Outline View, among others.

    ➢ Display additonal views by choosing Window→Show View.

✻ *Perspectives* are combinations of editors and views arranged in the workbench.

    ➢ You can switch from one perspective to another to see the appropriate combination of views and editors for your current needs by choosing Window→Open Perspective.

    ➢ The Window menu also allows you to save, customize, reset, and close Perspectives.

✻ The Java Perspective is the default perspective that you will see after dismissing the welcome screen.
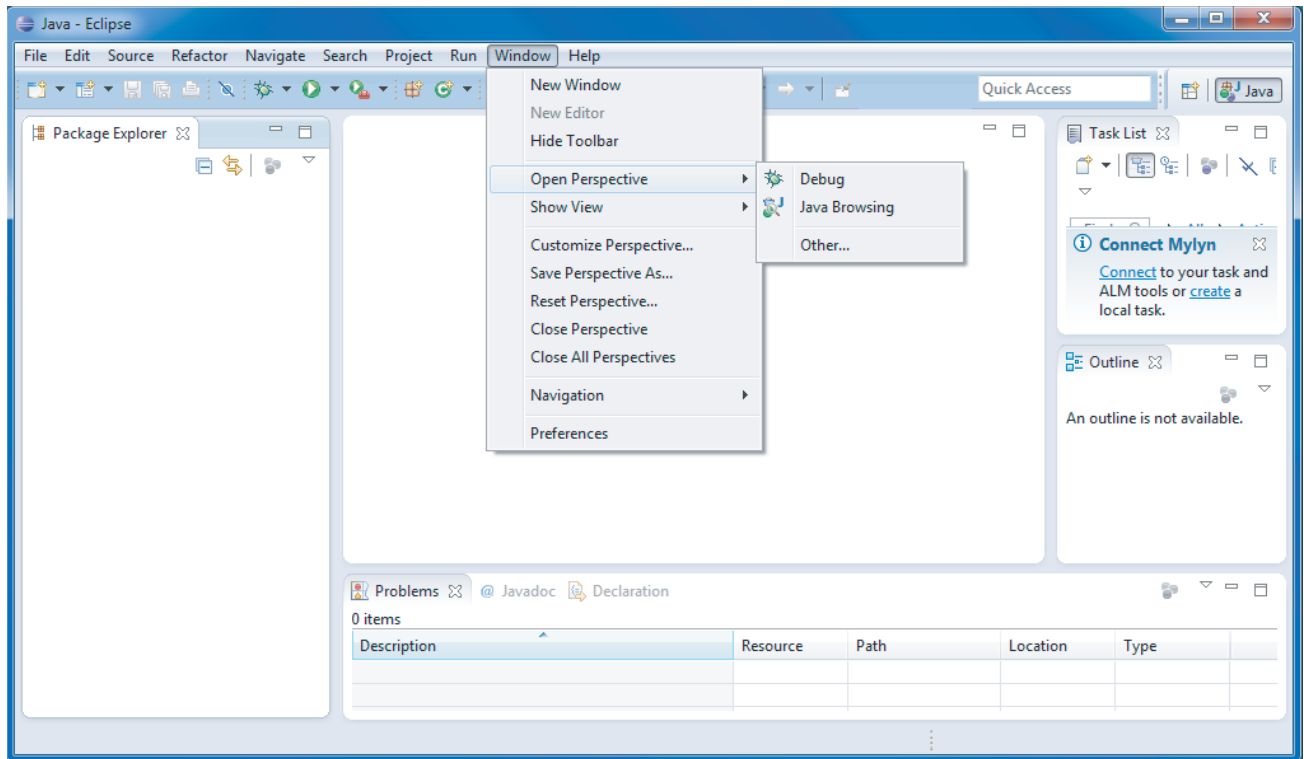
# Setting up a Project

✳ Before you can edit any Java code, you must first create a project.

➢ A *project* is a group of packages.

➢ You can have one default package per project, at most.

➢ You can find the *.java* files that correspond to the code you are editing within your workspace directory.

✳ To create a new project, choose File→New→Java Project.

➢ Enter a project name into the Project Name text box.

▪ The project name will become a subdirectory within your workspace directory.

➢ You can also specify which version of Java your project should be compliant with.

➢ Click Finish to create the project.

✳ Once your project has been created, you will see the project name listed in the Package Explorer view.
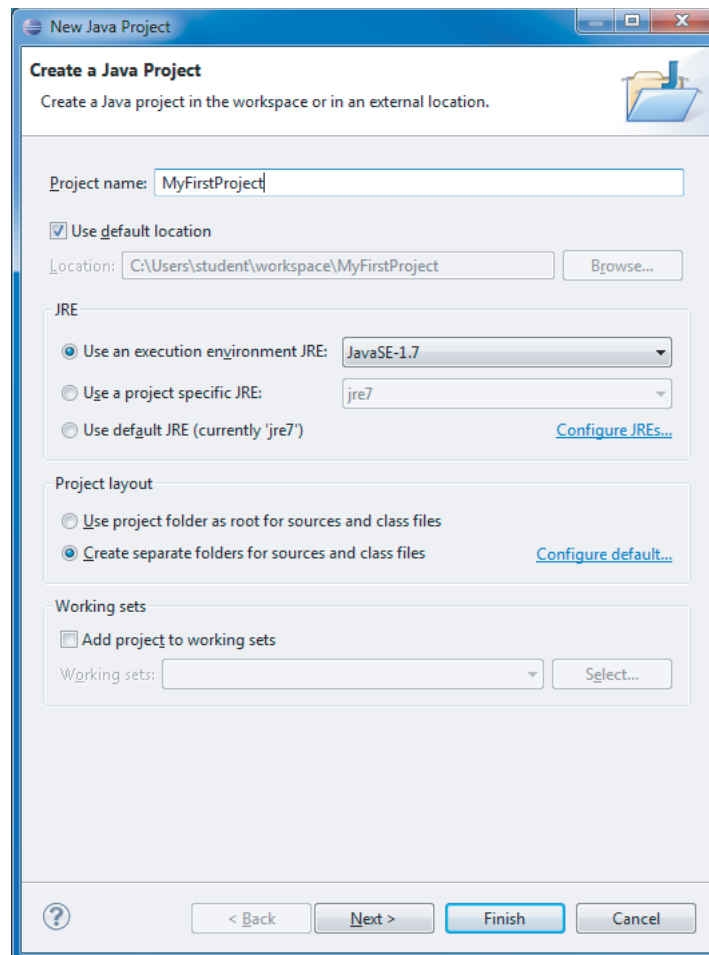
# Creating a New Java Application

❉ To create a Java class, click on File→New→Class.

    ➢ You can enter the name of the package to which your new class should belong.

    ➢ You must specify a name for the new class in the Name text field.

    ➢ Check the appropriate radio button and checkboxes to apply the appropriate modifiers to the new class.

    ➢ Eclipse can also automatically generate the **main()** method for you.

    ➢ Click Finish to create the new class.

❉ The new class will automatically open up in a Java editor where you can add your code to the class.

❉ Every time you save the file, Eclipse will compile your class for you.

    ➢ Any compiler errors are visible in the Problems view at the bottom of the screen.

Look for compiler errors here.
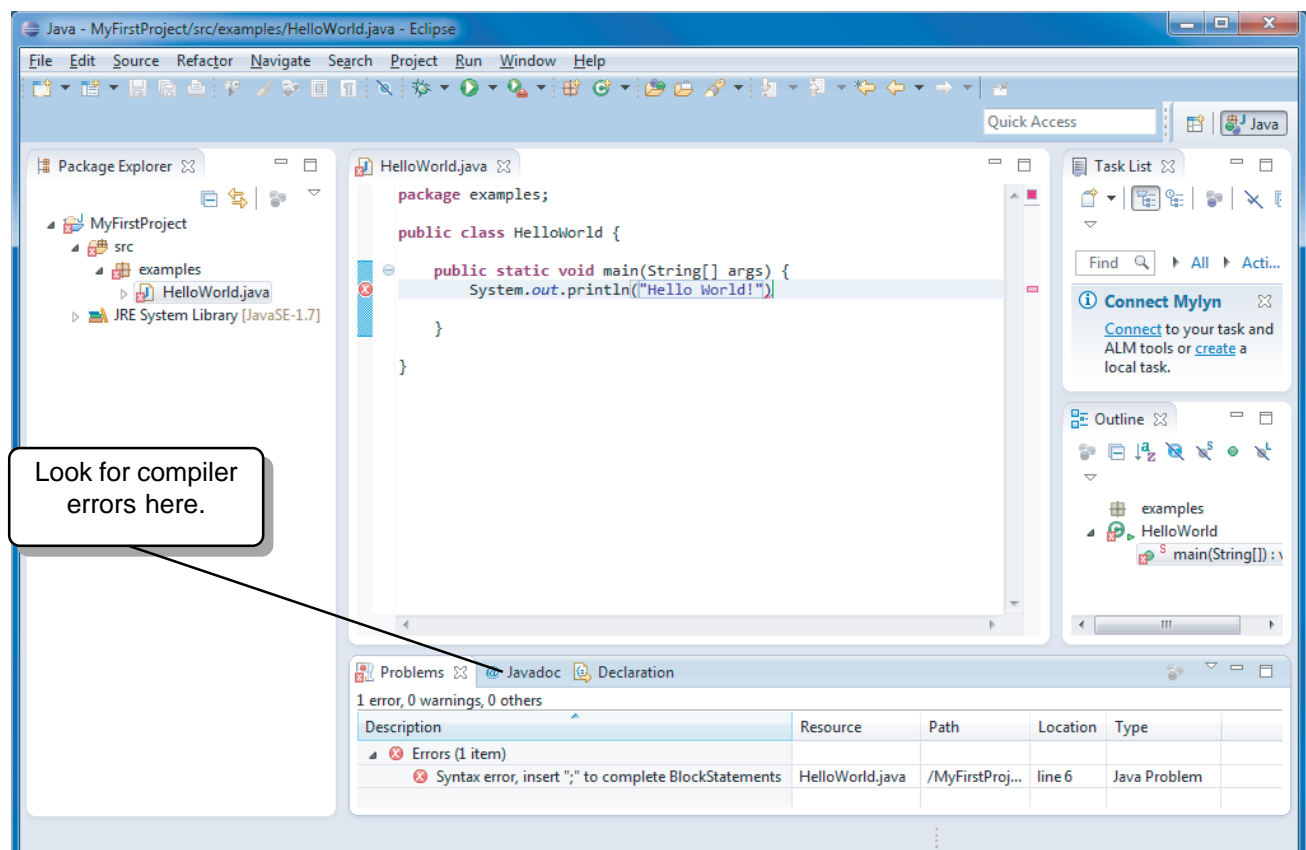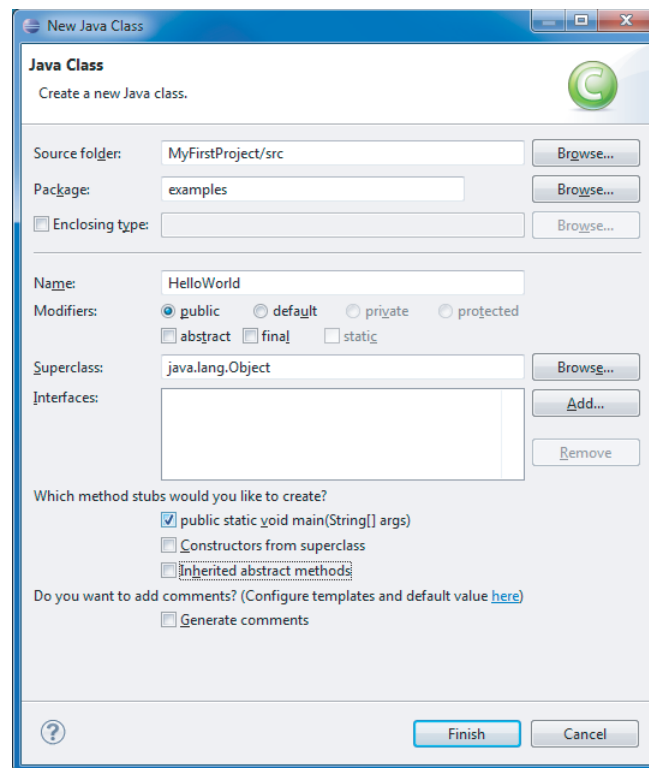
# Running a Java Application

❈ To run your program, click the Run button in the toolbar.

➢ The results will be displayed in the Console view at the bottom of the screen.

❈ If you would like to run your program with custom options, choose Run→Run Configurations.

➢ Choose Java Application and click the New button.

➢ Use the Arguments tab to pass in arguments and various other tabs to customize the environment before you invoke the program.

➢ Click the Run button to execute your program.

This is the Run button.

Standard out
and standard
err are shown in
the console.
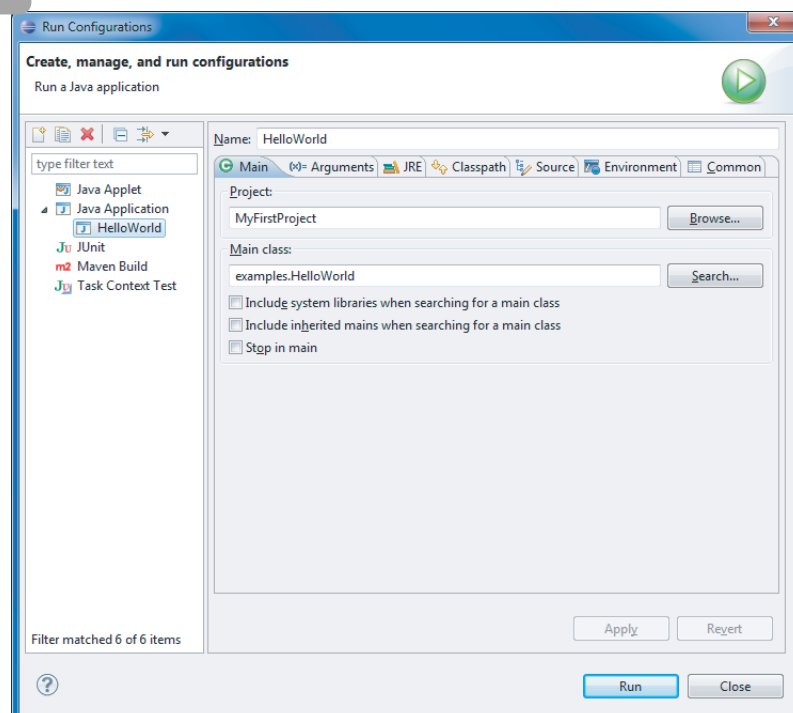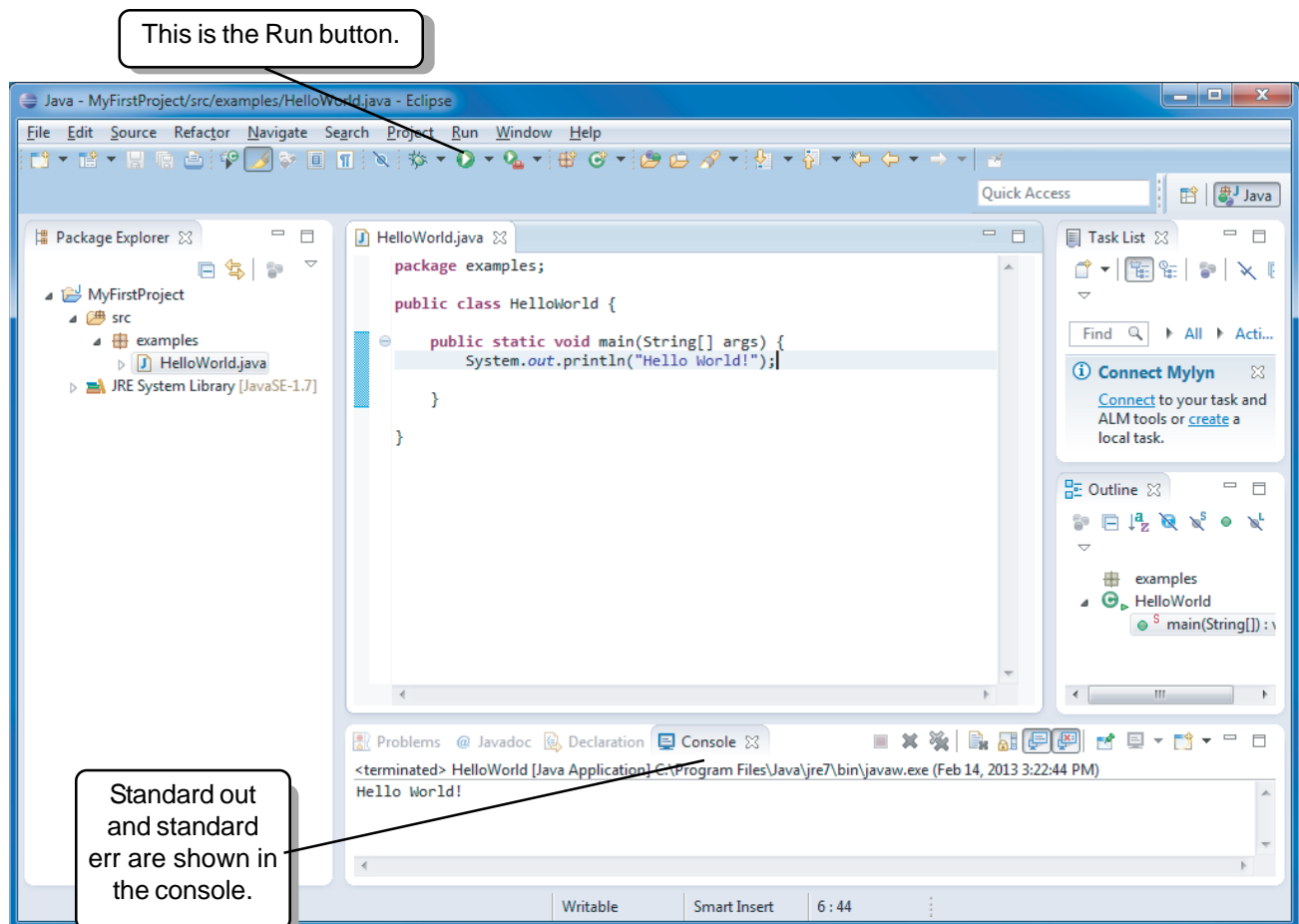
# Debugging a Java Application

❈ Before debugging your program, set at least one breakpoint by double-clicking on the blue margin of the Java editor to the left of the line of code where you wish to set the breakpoint.

❈ Click the Debug button in the toolbar.

   ➢ You will be notified that the perspective will change to the debug perspective before the debugger starts.

❈ Within the debug perspective, you are presented with multiple views and a different toolbar.

   ➢ Use the toolbar buttons to step through the program as it runs.

   ➢ The Debug view displays the current location within the method call stack.

   ➢ The Variables view shows you the current value of the variables in your program.

      ▪ You can also hover your mouse over any variable reference in the Editor to see its current value.

   ➢ The Breakpoints view allows you to manage the breakpoints in your debug session.

   ➢ The Console view displays the current results that have been written to standard out.

   ➢ The Outline view displays the structure of your class and highlights the current method you have stepped into.

❈ After you have completed debugging your program, switch back to the Java Perspective by clicking Window→Open Perspective→Java or you can click the Java button in the upper right corner of the screen.

Use this toolbar to step
through your application.
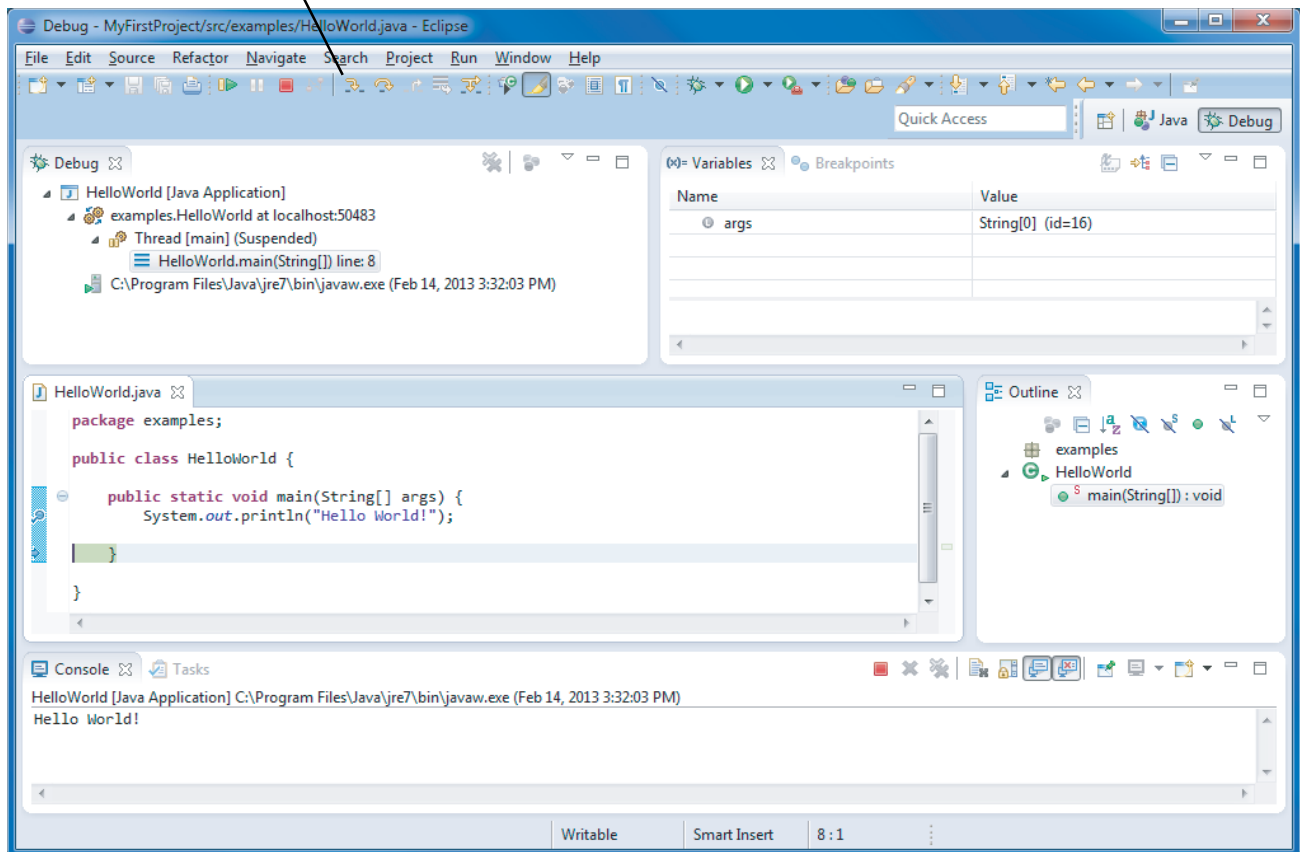


**Note:**
If you have installed a JSDK rather than a JRE, Eclipse can step into the source code for the Java API libraries.

# Importing Existing Java Code into Eclipse

❋    To load pre-existing Java files into an Eclipse project, use the Import feature by choosing File→Import...

    ➢    Choose General→File System to tell Eclipse to look on your local drives for the existing Java files.

    ➢    Browse to the directory above where your packages are located.

    ➢    Choose the checkboxes for the packages to import into your project.

    ➢    Make sure the Into folder text field specifies the name of the workspace folder within your project you wish to import into.

    ➢    Click Finish.

❋    You can also copy the files into the workspace folder using your file system's copy utilities.

    ➢    Choose File→Refresh and the files will be automatically imported into your project.

❋    On Windows systems, you can even drag the files from your file system into an Eclipse project.

❋    To load existing Eclipse projects into your workspace, choose File→Import...→General→Existing Projects into Workspace and select either the root directory or archive file to load the projects from.
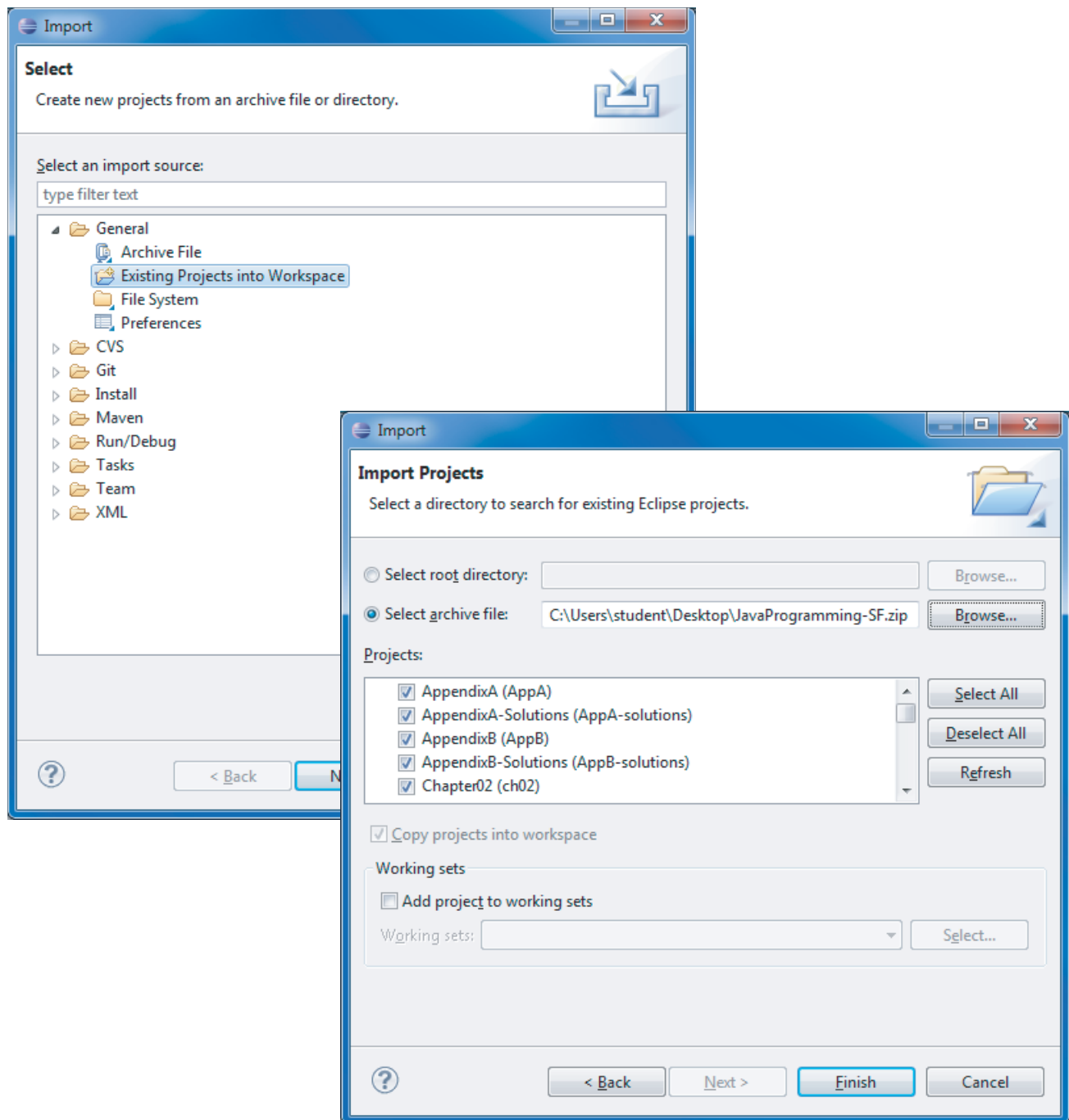
**Try It:**

Load the pre-created Eclipse projects for this course into your Eclipse workspace using the Import wizard. First, click on File→Import→General→Existing Projects into Workspace. Next, choose the Select archive file radio button and browse to the location of the student zip file (ask your instructor where this zip file is located). Click on Finish. You should see a project for each chapter loaded into your workspace.

# CHAPTER 4 - DATATYPES AND VARIABLES

## OBJECTIVES

❋ List the basic Java datatypes and describe their properties.

❋ Describe the difference between primitive and non-primitive datatypes.

❋ Write variable declarations for programs.

❋ Use numeric, character, and **String** literals in your programs.

❋ Use arrays in your programs.

# PRIMITIVE DATATYPES

�֍   Java uses eight basic (*primitive*) datatypes.

✖   Integer numeric datatypes store whole numbers.

**byte**         A one-byte, signed integer value.

**short**        A two-byte, signed integer value.

**int**          A four-byte, signed integer value.

**long**         An eight-byte, signed integer value.

✖   Floating-point numeric datatypes store numbers with a fractional component.

**float**        A four-byte floating point, IEEE 754-1985 value.

**double**       An eight-byte floating point, IEEE 754-1985 value.

✖   Character datatypes actually store numeric codes that represent characters of any alphabet via 16-bit Unicode.

**char**         A two-byte, unsigned integer character code.

✖   Boolean (true/false) values are not numbers.

**boolean**      A non-numeric value, either **true** or **false**.

| Type | Minimum | Maximum | Default | Size |
|------|---------|---------|---------|------|
| byte | -128 | 127 | 0 | 8-bit signed integer |
| short | -32768 | 32767 | 0 | 16-bit signed integer |
| int | -2147483648 | 2147483647 | 0 | 32-bit signed integer |
| long | -9223372036854775808 | 9223372036854775807 | 0 | 64-bit signed integer |
| float | -1.40239846e-45 | 3.40282347e+38 | +0.0F | 32-bit IEEE float |
| double | -4.94065645841246533e-324 | 1.79769313486231570e+308 | +0.0D | 64-bit IEEE double |
| boolean | false | true | false | N/A |
| char | \u0000 | \uffff | \u0000 | 16-bit Unicode |

Whenever you need to deal with a numeric value that has a fractional component, you may store such a value in either a **float** or a **double** variable. The only difference is how precise a value you store. With a **float**, you get about 6 decimal places of precision; with a **double**, about 15 decimal places.

## Investigate:

What datatype would you choose for the following data:

- A person's middle initial?
- The length and width of a room?
- The national debt?
- The current year?

# Declarations

❈ A *declaration* specifies the name and datatype, and allocates storage for a piece of data.

```
int age = 30;
```

- Each **int** variable is allocated four bytes for storage.
- This **int** is named **age**.
- **age** may only hold **int** values.
- Its initial value is **30**.

❈ A local variable declaration can occur anywhere in a method body.

➢ The declaration has to occur somewhere before the first statement that uses the local variable.

➢ You must *initialize* (assign an initial value) a local variable before using it.

```
int age;      // Next line won't compile.
System.out.println("Age is " + age);
```

❈ A variable that is declared in a class outside of any methods is called a *field*.

➢ A field does not have to be initialized in its declaration.

- The compiler gives it a default initial value (**0** for numeric variables, **false** for a **boolean**).

❈ Variables declared with the modifier **final** must be assigned before use and cannot be changed once assigned.

```
final short cards = 52;
cards = 53;   //Illegal!  Can't change a final.
```

A line that begins with // is a *comment*. It's not required, and is ignored by the compiler. Anything following a // is just a note to ourselves that Java ignores, up to the end of that line. Use the /* */ comment characters if you want to comment out multiple lines.

Initialize.java

```
package examples;

public class Initialize {

   // These fields are automatically initialized.
   static int fieldI;
   static double fieldD;
   static boolean fieldB;

   public static void main(String[] args) {
      /*
         If you don't initialize these local variables,
         you can't compile.
      */
      int localI = 17;
      double localD = 1.414213562373095;
      boolean localB = true;

      System.out.println("             Field     Local Variable");
      System.out.println("int:     " + fieldI + "              " + localI);
      System.out.println("double:" + fieldD + "         " + localD);
      System.out.println("boolean: " + fieldB + " " + localB);
   }
}
```

## Try It:
Run *Initialize.java* to see how fields are automatically initialized.

## Note:
A declaration can declare more than one variable.

```
int age = 30,
   siblings = 0;
```

# Variable Names

❋ A variable name (*identifier*) is a sequence of letters, digits, and underscores.

```
int port2;
```

❋ An identifier must start with a letter or underscore.

```
int 4thPort;        // invalid!
```

❋ Identifiers are case-sensitive.

```
char    MiddleInitial;    /* All different */
char    middleInitial;
char    middleinitial;
```

❋ Identifiers can be as long (or short) as you want.

```
int theTotalNumberOfCustomersWhoHaveRegisteredSoFar;
int n;
```

❋ An identifier must not be a keyword or a literal value.

Keywords are tokens reserved by the language. You cannot use these Java keywords as identifiers:

```
abstract     continue  for        new         switch
†assert      default   if         package     synchronized
boolean      do        *goto      private     this
break        double    implements protected   throw
byte         else      import     public      throws
case         ‡enum     instanceof return      transient
catch        extends   int        short       try
char         final     interface  static      void
class        finally   long       strictfp    volatile
*const       float     native     super       while
```

> *Reserved, but not currently used
> †Added at 1.4
> ‡Added at 1.5 (Java 5.0)

The words **true** and **false** are not formally *keywords*, but *boolean literals*; however, the distinction is slight and you should not try to use them as identifiers. The same is true of the *null value literal*, **null**.

Valid identifiers:
```
total_amount    //totalAmount is preferred
getRadius
MyClass
$revenue
This_is_a_mighty_long_identifier_and_could_be_even_longer  //  Yuck!
_$_$____$        //  YUCK!!!
```

Invalid identifiers:
```
#sterling    // Invalid character, '#'
get Radius   // Two words (invalid character, ' ')
class        // This is a keyword
```

## Note:
By convention, most Java programmers capitalize class names and lowercase variable and method names. The **$** character is intended to be used in variable names only in code that is generated programmatically.

# NUMERIC LITERALS

❋ *Literals* are used in expressions and to assign values to variables.

```
float radius = diameter / 2.0F;
int age = 19;
```

❋ A numeric literal with no decimal point will be considered an **int**.

➢ Specify a numeric literal to be a **long** by appending an **L** or **l**.

```
long worldPop = 6973738433L;
```

➢ As of Java 7, you can use an underscore within your numeric literals to break up hard to read numbers.

```
long worldPop = 6_973_738_433L;
```

❋ A numeric literal with a decimal point is considered a **double**.

```
double circum = 47.25;
```

➢ A trailing **F** or **f** on a floating point literal specifies a **float**.

```
float height = 43.8F;
```

➢ You can also use exponential notation to specify floating point literals.

```
double atomicWeight = 2.65e-8;
float momsHeight = 6.65E+1F;
```

Volumes.java

```
...
public class Volumes {
  public static void main(String[] args) {
    float eRadius = 6371000.0F; // earth radius
    float hRadius = 37.1E-12F;  // hydrogen atom radius

    float earthRadiusCubed = eRadius * eRadius *eRadius;
    float hydrogenRadiusCubed = hRadius * hRadius *hRadius;

    float eVolf = 0.0F;            // earth volume (float)
    float hVolf = 0.0F;            // hydrogen volume (float)
    float ehRatiof = 0.0F;         // earth to hydrogen ratio (float)
    float heRatiof = 0.0F;         // hydrogen to earth ratio (float)

    double eVold = 0.0;            // earth volume (double)
    double hVold = 0.0;            // hydrogen volume (double)
    double ehRatiod = 0.0;         // earth to hydrogen ratio (double)
    double heRatiod = 0.0;         // hydrogen to earth ratio (double)


    eVolf = (4.0F / 3.0F) * (float) Math.PI * earthRadiusCubed;
    eVold = (4.0 / 3.0) * Math.PI * earthRadiusCubed;

    hVolf = (4.0F / 3.0F) * (float) Math.PI * hydrogenRadiusCubed;
    hVold = (4.0 / 3.0) * Math.PI * hydrogenRadiusCubed;

    ehRatiof = eVolf / hVolf;
    ehRatiod = eVold / hVold;

    heRatiof = hVolf / eVolf;
    heRatiod = hVold / eVold;

    System.out.println("        Earth                    Hydrogen");
    System.out.println("float: " + eVolf + "            " + hVolf);
    System.out.println("double: " + eVold + "  " + hVold);

    System.out.println();
    System.out.println("        E/H Ratio          H/E Ratio");
    System.out.println("float: " + ehRatiof + "              " +
      heRatiof);
    System.out.println("double: " + ehRatiod + " " + heRatiod);
  }
}
```

Try It:
Run *Volumes.java* to calculate the volume of the earth and of a hydrogen atom.

# Character Literals

❊ A character enclosed in single quotes is a *character literal*.

```
char middleInitial = 'A';
```

❊ The value of a character literal is stored as a Unicode character code number.

**'A'** is 65          **'9'** is 57
**'Z'** is 90          **'Ù'** (Greek Small Letter Omega) is 937
**'0'** is 48          **'•'** (Euro Sign) is 8364

➢ Unicode values are usually given in hexadecimal notation.

**'A'** is 0041          **'9'** is 0039
**'Z'** is 005A          **'Ù'** is 03A9
**'0'** is 0030          **'•'** is 20AC

➢ Unicode values can represent characters from almost every written language.

❊ You can specify a character using Unicode by preceding its Unicode value with a **\u**:

```
middleInitial = '\u0041';
char theta = '\u03B8';
char euroSign = '\u20AC';
```

➢ You may not be able to accurately view or print a Unicode character unless you have the appropriate font installed on your system.

---

A character variable in the C programming language is only one byte: an ASCII character code. ASCII character codes use only 7 of a byte's 8 bits (the other bit was used in older data transmission protocols). So, such a variable can hold one of only 128 distinct values of ASCII character codes. These codes represent the letters of the American alphabet (both capital and small letters), the digit characters, punctuation, and about 30 *non-printable* control codes originally used to control teletype machines.

C programmers are accustomed to using ASCII character codes to represent all text. They even memorize a few of the ASCII codes; for example, most C programmers know that ASCII code 10 is the newline (line-feed) character, represented in a C program as **'\n'**.

The Unicode standard uses the same character codes as ASCII for the American (formally, the "Basic Latin") alphabet. That is, 65 is a capital A in both ASCII and Unicode; 10 is a newline character in both (and is also represented as **'\n'** in a Java program). So, C programmers are comfortable in the Unicode world of Java.

A character variable in Java is 2 bytes: a Unicode character code. Using all 16 bits, Unicode can represent 65,536 distinct character codes. These include the alphabets for languages from around the world, a wide range of symbols, and so on. The keywords of the Java language itself are always written in Basic Latin characters, but the names of variables, methods, and classes can be written in Chinese, Japanese, Korean, Arabic, Greek, Hebrew, any of several Indian languages, etc.

Unicode also defines some characters beyond the ones representable with 16 bits. In Java, these must be represented using more than one **char** value in sequence, via an encoding called **UTF-16**. There are additional facilities provided in Java's **Character** class, which can use a 32-bit integer to represent any Unicode character.

For more information on Unicode, visit ***http://www.unicode.org***.

## STRING

❋ Java's **String** class handles most string processing.

```
String name = "James";
String lastName = "Gosling";
```

❋ To see if two **String** objects have equivalent content, use the **equals()** method instead of ==.

```
boolean b = name.equals(lastname);
```

➢ The == operator checks whether two reference values refer to the exact same object.

➢ The **equals()** method compares the contents of the objects.

❋ **String** objects are *immutable*; they cannot be changed.

➢ The concatenation operator (+) creates a new **String** object.

```
name = name + " " + lastName;
```

Compare.java

```java
package examples;

public class Compare {
  public static void main(String[] args) {

    if (args.length == 2) {
      // Check ==
      if (args[0] == args[1]) {
        System.out.println("Both Strings are ==");
      }
      else {
        System.out.println("Both Strings are NOT ==");
      }

      // Check equals()
      if (args[0].equals(args[1])) {
        System.out.println("Both Strings are 'equal'");
      }
      else {
        System.out.println("Both Strings are NOT 'equal'");
      }

    }
    // wrong number of command line arguments
    else {
      System.out.println("Usage: java Compare arg1 arg2");
    }
  }
}
```

## Try It:

Run *Compare.java*. Try passing in your firstname for **args[0]** and **args[1]**.

```
java Compare James James
```

# STRING LITERALS

❋ A *string literal* is a sequence of characters within double quotes.

```
String greeting = "Hello";
```

➢ A string literal can contain "escape sequences" that represent certain special characters.

```
String message = "End of the line...\n";
greeting = "I said, \"Hello\"";
String name = args[0] + "\t" + args[1];
```

➢ String literals are converted into **String** objects.

➢ Methods of the **String** class also work with string literals.

```
boolean designer = "James Gosling".equals(name);
```

❋ String literals with identical content become references to the same, shared object.

```
String orig = "xyz";
String dup = "xyz";
String other = new String("xyz");
System.out.println(orig == dup); // prints true
System.out.println(orig == other); // prints false
```

➢ This space-saving feature is provided by the compiler.

Escape sequences for special characters:

| Escape | Unicode | Special Character |
| --- | --- | --- |
| \n | \u000A | newline |
| \t | \u0009 | tab |
| \b | \u0008 | backspace |
| \r | \u000D | return |
| \f | \u000C | form feed |
| \\ | \u005C | backslash |
| \' | \u0027 | single quote |
| \" | \u0022 | double quote |
| \ddd | \u0000-\u00FF | a character in octal notation (for old C programmers) |

# Arrays

❉ A variable declared with brackets, **[]**, is an *array reference*.

➢ Put the **[]** next to the datatype or the name; your choice.

```
float price[];
String[] title, author; // Both are array refs
```

❉ You must use **new** to actually create the array, which is an object.

```
price = new float[44]; // Create array of 44 floats
String[] names = new String[10]; //Declare & create
```

➢ The array holds a fixed number of elements of the indicated type.

➢ An array's **length** field gives the number of elements in the array.

➢ Once created, an array cannot be resized.

❉ Each element in an array is a variable with no name of its own.

➢ Primitive-type array elements are initialized to default values.

➢ If the array is an array of object references, then each object must also be created with **new**.

```
names[1] = new String("Bob Dobbs");
```

❉ Elements are indexed using **[]**, and numbered from **0**.

```
price[0] = 99.95F;
```

➢ Indexing past the end of an array throws an exception.

❉ Multi-dimensional arrays are implemented as arrays of arrays.

Consider the **args** array, which is the array of **String**s containing the command-line arguments passed to the **java** program.

Echo.java

```
...
public class Echo {
  public static void main(String args[]) {
     System.out.println(args[0]);
  }
}
```

```
java Echo Howdy, Folks!
Howdy,
```

To print the last command-line argument:
LastOne.java

```
...
public class LastOne {
  public static void main(String args[]) {
     System.out.println(args[args.length - 1]);
  }
}
```

```
java LastOne one two three
three
```

ArrayInit.java

```
...
public class ArrayInit {
  public static void main(String args[]) {
    int evens[] = new int[5];
    evens[0] = 0;
    evens[1] = 2;
    evens[2] = 4;
    evens[3] = 6;
    evens[4] = 8;

    int odds[] = {1,3,5,7,9};

    for (int i = 0; i < evens.length; i=i+1) {
       System.out.println(evens[i]);
       System.out.println(odds[i]);
    }
  }
}
```

> Instead of using **new**, you can initialize an array with a list of values inside braces.

# NON-PRIMITIVE DATATYPES

✻ A non-primitive datatype is defined as a *class*.

➢ Java comes with many useful classes already defined.

✻ A particular value of any class type is called an *object*, and it differs from a primitive value in several ways:

➢ All objects are created at runtime on the *heap* (dynamically-allocated memory), using the keyword **new**.

➢ Java code deals with an object via a *reference*, which identifies where the object is actually stored.

➢ A variable declared to have a class type will not hold any object of that class, but only a reference to an object.

▪ Here, the variable **s** holds a reference to a **String** object:

```
String  s = new String("James Gosling");
```

The Java language syntax provides special support for two kinds of objects:

- **String** objects — the + operator performs concatenation, and double-quoted literals create **String** objects.
- **array** objects — square brackets allow indexing, and a special syntax specifies initial values.

Concat.java
```
package examples;

public class Concat {
  public static void main(String[] args) {
    if (args.length > 1) {
      String s = new String("<" + args[0] + args[1] + ">");
      System.out.println(s);
    }
  }
}
```

All objects must be created with the **new** keyword, except for **String** objects with pre-determined contents (or array variables declared with an initializer), for which special support is provided in the Java syntax. For example, in *Concat.java*, the declaration of **s** has an initializer that uses the **new** keyword unnecessarily:

```
String s = new String("<" + args[0] + args[1] + ">");
```

It could have been written this way instead:

```
String s = "<" + args[0] + args[1] + ">";
```

# The Dot Operator

❋ A variable that is declared with a non-primitive datatype can hold a reference to an object.

```
String name;
```

➢ This type of variable is sometimes called a *reference* or a *reference variable*.

❋ The dot operator (**.**) is used to call methods on a reference.

```
name = "James Gosling";
int strLength = name.length();
```

➢ The **length()** method is called on the object referenced by **name**.

❋ You can also use the dot operator to access fields from a reference.

```
String[] months = ...
int arrayLength = months.length;
```

➢ The **length** field is retrieved from the array referenced by **months**.

❋ You can only use the dot operator on reference variables that have been assigned a non-**null** value.

➢ You will get a **NullPointerException** if you attempt to use the dot operator on a **null** reference.

➢ Reference variables that are declared as fields are automatically initialized to **null**.

NullError.java

```
package examples;

public class NullError {
  public static void main(String args[]) {
    String  s;                             // uninitialized
    // System.out.println(s.length());   // Compiler Error
    s = null;                              // s refers to null
    System.out.println(s.length());       // Runtime Error
  }
}
```

Try It:
Run *NullError.java* to see a runtime **NullPointerException**.

# Labs

❶     Write a program that creates variables to hold information about a dog. What datatypes would you
       use to store a name, weight, length of tail, and whether or not it's friendly?
       (Solution: *Dog1.java*)

❷     Modify your solution to ❶ to assign a value to each variable. Print each variable out.
       (Solution: *Dog2.java*)

❸     Write a program that declares and initializes two **int** variables: **big (2147483647)** and **bigger
       (big + 1)**. Print them both out and explain the results.
       (Solution: *Overflow.java*)

❹     Write a program that declares and initializes three **char** variables: one each for your first, middle,
       and last initials. Use a **System.out.println()** statement to print "**Initials:** " followed by the values of
       the variables.
       (Solution: *MI.java*)

❺     Modify your solution to ❹ to use Unicode escapes, rather than character literals, to initialize the
       variables. (Hint: Unicode for **'A'** is **'\u0041'**, **'Z'** is **'\u005A'**.)
       (Solution: *MIUnicode.java*)

❻     What happens if the **System.out.println()** statement contains <u>only</u> the three **char** variables, and not
       the string "**Initials:** " or anything else?
       (Solution: *MIChars.java*)

❼     Write a program that creates an array of 12 **String**s. Populate each member of the array with the
       name of a month. Print out the name of the month in which you were born.
       (Solution: *MonthsArray.java*)

To send output to your terminal screen or console, use **System.out**:

```
System.out.println("Hola, Mundo!");
```

You'll find out all about Java input and output streams later. For now, it's enough to know that there's an object called **System.out** that has methods to print information to your screen. The **print()** method prints its argument and nothing else. **println()** prints its argument, followed by a newline character. Both methods can take either **String** or numeric arguments:

```
double p = 3.14159265358979932384626433;
System.out.print("The number is: ");
System.out.println(p);
```

These methods convert numeric arguments into **String**s before printing. Another way to accomplish this is to combine a number with a **String** using the concatenation operator, +, which also converts the number to a **String**.

```
System.out.println("The number is: " + p);
```

The **System** class has another object, **err**, which is commonly associated with the "standard error" destination (the terminal again, by default):

```
System.err.println("Warning: this number was rounded.");
```

# CHAPTER 5 - OPERATORS AND EXPRESSIONS

## OBJECTIVES

✽ Use expressions to perform calculations and to test results.

✽ Use the various operators provided by Java to manipulate variables.

✽ Learn and apply the unique notations that Java uses for assignments, incrementing, and decrementing.

✽ Apply operator precedence and associativity in expressions.

✽ Convert operands to different datatypes.

✽ Design program logic using conditional expressions.

# Expressions

❋ *Expressions* are combinations of variables, literals, and operators.

❋ You write expressions to:

➢ Determine values.

```
5 – radius
inValue >= 0
width * height
total / 2
length + 6.13
```

➢ Alter values stored in variables.

```
numClasses = 6
gpa = gradeTotal / numClasses
```

❋ Method calls are also expressions.

```
System.out.println("This is an expression")
root = Math.sqrt(2.0)
```

❋ A *statement* can be an expression followed by a semicolon.

```
numClasses = 6;
gpa = gradeTotal / numClasses;
root = Math.sqrt(2.0);
```

❋ Such a statement can only appear in a block of code, such as a method.

➢ Expressions cannot appear outside of a class definition.

# ASSIGNMENT OPERATOR

✽    Use = to assign a value to a variable.

```
min = 5;
```

   ➢    The left operand must be a variable.

```
5 = x;  // illegal!
```

✽    The assignment itself is an expression that has a value — the value that was assigned.

```
current = (min = 5);  // Sets current to 5
```

✽    Assignments are evaluated right to left, so you can actually stack them in a single statement:

```
max = current = min = 5;
```

   is equivalent to

```
max = (current = (min = 5));
```

Consider the following expression:

```
count1 = (count2 = (count3 = 0));
```

1.  = is an operator.

2.  **(count3 = 0)** is an expression.

3.  The expression results in a value, which is the value of the assignment (**0** in this case).

4.  The result of the expression is used in the next expression:
    **(count2 = 0)**

5.  This expression also results in **0**.

6.  **count1 = 0;** is finally evaluated.

So all variables get assigned the value **0**.

# Arithmetic Operators

❋ *Binary operators* take two operands.

 + Addition

   `age + 1`

 **-** Subtraction

   `daysLeft – 3`

 **\*** Multiplication

   `width * height`

 **/** Division

   `totalLines / numPages`

 **%** Modulus

   `curLine % 60`

   ▪ Modulus is the remainder of ***operand1*/*operand2***.

     ```
     10 % 3 is 1
     10 % 2 is 0
     ```

❋ A *unary operator* takes one operand.

 **-** Unary negative

   `–balance`

 + Unary positive — actually does nothing!

The division operator actually works differently for floating point division and integer division. Floating point division is performed if either operand has a floating point datatype.

```
float diameter = 13F;
float radius = diameter / 2;          // radius is 6.5
```

Integer division is performed if both operands have an integral (non-floating point) datatype.

```
int diameter = 13;
int radius = diameter / 2;            // radius is 6
```

With integer division, any decimal place is lost. This is even true if the result is stored in a floating point variable.

```
int diameter = 13;
float radius = diameter / 2;          // radius is 6.0
```

Floating point division can be forced, by making one of the operands a floating point number.

```
int diameter = 13;
float radius = diameter / 2.0F;       // radius is 6.5
```

The arithmetic operators will always promote any integral value smaller than an **int** to an **int** value. Types **byte** and **short** are converted by sign extension, while type **char** (being unsigned) is converted by zero-extension. The resulting value will therefore always be at least as large as an **int**. To get back to a smaller type, an explicit cast will be required:

```
   short a = 12;
   short b = 2;
   short c = 3;
   a = a * b + c; // illegal, potential loss of precision
```

# Relational Operators

❋ A *relational operator* compares two expressions and results in a **true** or **false** value (a **boolean**).

|    |                          |
|----|--------------------------|
| >  | Greater than             |
| >= | Greater than or equal to |
| <  | Less than                |
| <= | Less than or equal to    |
| == | Equal to                 |
| != | Not equal to             |

❋ A relational expression can be used to control the flow of your program.

```
if (x >= hiValue)
   System.out.println("x is too big");

if (args.length != 2) {
   System.out.println("Usage: java MyProg a1 a2");
}
```

❋ The result of a relational expression can be stored in a **boolean** variable.

```
boolean tooBig = (x >= hiValue);
if (tooBig)
   System.out.println("x is too big");
```

Relate.java

```
package examples;

public class Relate {
  public static void main(String[] args) {
    int speed = 35;
    int speedLimit = 30;

    System.out.println("speed: " + speed);
    System.out.println("speedLimit: " + speedLimit);

    boolean result = speed < speedLimit;
    System.out.print("speed < speedLimit results in ");
    System.out.println(result);

    result = speed > speedLimit;
    System.out.print("speed > speedLimit results in ");
    System.out.println(result);
  }
}
```

Is **speed** less than **speedLimit**?

Or is **speed** greater than **speedLimit**?

## Try It:

*Relate.java* uses < and > operators to find out if **speed** is too fast.

# LOGICAL OPERATORS

✸ To combine the results of two relational expressions, use a *logical operator*.

|| A logical OR will result in **true** if either operand, or both, is true.

```
length < 12 || width > 18
```

&& A logical AND will result in **true** if <u>both</u> operands are true.

```
height >= 10 && height <= 20
```

! Use a logical NOT to negate a relational expression.

```
!(height >= 10 && height <= 20)
```

✸ The **&&** and || operators use *short-circuit evalution*.

➢ If the left operand determines the outcome, then the right operand expression is not evaluated.

▪ For **&&**, the expression to the right is performed only when the expression to the left is **true**.

▪ For ||, the expression to the right is performed only when the expression to the left is **false**.

➢ This acts somewhat like an **if** statement.

ShortCircuit.java

```
package examples;

public class ShortCircuit {
   public static void main(String[] args) {
      int i = 0;
      int max = 8;

      boolean result = (i >= max) && ((i = max) > 5);
      System.out.println(result + ", with i == " + i);
   }
}
```

## Try It:
Run *ShortCircuit.java* to see how an expression with a side effect used as the right operand of **&&** will not be performed if the left operand is **false**. Now, change the inital value of **i** to **10** to see the side effect occur.

## Investigate:
Examine the file *LogicTable.java*. Predict what the results will be before you compile and run it.

# Increment and Decrement Operators

❋ Increment and decrement are unary operators: they only require one operand.

➢ The operand must be a variable.

❋ These operators modify the value stored in their operand.

++ The increment operator adds one to its operand.

```
n++;    // same as n = n + 1;
```

-- The decrement operator subtracts one from its operand.

```
n--;    // same as n = n – 1;
```

❋ Increment and decrement come in two forms, prefix and postfix.

➢ The *prefix* version modifies the operand before its value is used.

```
int a = 0;
int b = 0;
a = ++b;    // a is 1, b is 1
```

➢ The *postfix* version modifies the operand after its value is used.

```
int a = 0;
int b = 0;
a = b++;  // a is 0, b is 1
```

PrePost.java

```
package examples;

public class PrePost {
  public static void main(String[] args) {
    int a = 0;
    int b = 0;

    a = ++b;
    System.out.println("Prefix:  a is " + a + ", b is " + b);

    // Reinitialize the variables:
    a = 0;
    b = 0;

    a = b++;
    System.out.println("Postfix: a is " + a + ", b is " + b);
  }
}
```

What are the values of **a** and **b**?

## Try It:
*PrePost.java* illustrates the difference between **prefix** and **postfix** increment operators.

# OPERATE-ASSIGN OPERATORS (+=, ETC.)

❋ Increment and decrement operators are very handy for adding or subtracting **1** from an operand.

```
int n = 0;
n++;          // increment n by 1 (n = n + 1)
```

❋ The "operate-assign" operators allow you to modify an operand by <u>any</u> value.

```
int n = 0;
n += 2;       // increment n by 2 (n = n + 2)
n += 50;      // increment n by 50  (n = n + 50)
```

➢ The value can even be a variable.

```
int age = 0;
int numYears = 5;
age += numYears;    // increment age by 5
```

❋ These operators correspond to all the basic arithmetic operators.

```
n += 50;     // increment n by 50 (n = n + 50)
n -= 50;     // decrement n by 50 (n = n – 50)
n *= 50;     // multiply n by 50  (n = n * 50)
n /= 50;     // divide n by 50    (n = n / 50)
n %= 50;     // modulate n by 50  (n = n % 50)
```

OpAssign.java

```
package examples;

public class OpAssign {
    public static void main(String[] args) {
        int numPeople = 11;
        float flight = 575.0F;
        float hotel = 876.35F;
        float carRental = 135.50F;
        float tripPrice = 0F;

        tripPrice += flight;
        tripPrice += hotel;
        tripPrice += carRental;

        System.out.println("Cost per person: $" + tripPrice);

        tripPrice *= numPeople;
        System.out.println("Total cost: $" + tripPrice);
    }
}
```

## Try It:

This example uses the += and *= operators to calculate the **tripPrice**.

# The Conditional Operator

✸ The *conditional operator* (**?:**) takes <u>three</u> operands, but doesn't modify any of them.

```
op1 ? op2 : op3
```

✸ **op1** is a **boolean** expression that evaluates to **true** or **false**.

➢ If **op1** is **true**, **op2** is the result of the conditional operator.

➢ Otherwise, **op3** is the result of the conditional operator.

```
char status;  //'a' for adult, 'm' for minor
int age = 16;
status = age >= 18 ? 'a' : 'm'; // minor!
```

✸ The result of a conditional operator is usually used in another expression or assignment.

➢ The conditional operator is sometimes called the *ternary operator* (because it's the only operator that takes three operands).

➢ The conditional operator can replace certain simple conditional (**if**) statements.

Often, an **if** statement can be clearer than a conditional operator. However, there are situations where the opposite is true.

NameAndPhone.java
```
package examples;

public class NameAndPhone {
   public static void main(String[] args) {

      String name = args.length > 0 ? args[0] : "anonymous";
      String phone = args.length > 1 ? args[1] : "not listed";
      System.out.println("Name: " + name + " Phone: " + phone);

      /*
         if (args.length > 0)
            name = args[0];
         else
            name = "anonymous";

         if (args.length > 1)
            phone = args[1];
         else
            phone = "not listed";
         System.out.println("Name: " + name + " Phone: " + phone);
      */
   }
}
```

## Try It:
Run *NameAndPhone.java*. Pass in a name and phone number as arguments. Run it again without the phone number. Run it once more with no arguments.

# Operator Precedence

❋ Expressions are evaluated according to precedence rules.

```
2 + 3 * 7 // * is done first (result is 23)
```

➢ Operators with higher precedence are evaluated before operators with lower precedence.

❋ You can change the order of evaluation using parentheses.

```
(2 + 3) * 7 // + is done first (result is 35)
```

➢ You can even nest several sets of parentheses to dictate the exact order that you need.

❋ If operators in an expression have the same precedence, they are evaluated either from left-to-right, or right-to-left, according to their associativity.

| Expression | Answer | Order |
|---|---|---|
| 8 + 3 * 7 - 9 | 20 | *, +, - |
| 8 + 3 * (7 - 9) | 2 | -, *, + |
| (8 + 3) * (7 - 9) | -22 | +, -, * |

Precedence and Associativity of Operators

| Operator | Class | Precedence | Associativity |
|---|---|---|---|
| ++ -- (prefix or postfix)<br>! - ~ +<br>(*cast*) | unary | 1 | right-to-left |
| * / % | binary | 2 | left-to-right |
| + - | binary | 3 | left-to-right |
| << >> >>> | binary | 4 | left-to-right |
| instanceof < <= > >= | binary | 5 | left-to-right |
| == != | binary | 6 | left-to-right |
| & | binary | 7 | left-to-right |
| ^ | binary | 8 | left-to-right |
| \| | binary | 9 | left-to-right |
| && | binary | 10 | left-to-right |
| \|\| | binary | 11 | left-to-right |
| ?: | ternary | 12 | right-to-left |
| = *= /= %= += -= <<=<br>>>= &= \|= ^= >>>= | binary | 13 | right-to-left |

# IMPLICIT TYPE CONVERSIONS

❋ Operands of different datatypes in an expression are converted to a common type.

```
float total;
float f = 6.5F;
int i = 5;
total = f + i;    //float = float + int
```

❋ There are rules that specify how the conversions occur.

➢ In general, smaller datatypes are converted to larger datatypes:

▪ **short** is converted to **int**, **int** is converted to **long**.

▪ **short**, **int**, or **long** are converted to **float**.

▪ **float** is converted to **double**.

❋ These conversions are implicit, because they happen automatically.

```
total = f + i;    // i is converted to float
```

❋ Conversions that would truncate data will not happen implicitly.

```
float f = 6.5F;
float g = 7.7F;
int total;
total = f + g;    //error – truncation
```

➢ The **float** result of the addition will not be truncated to an **int** to fit in **total**.

Conversion from smaller to larger numeric types is implicit (it occurs automatically when needed). Converting from an integral type to a floating point type (**int** or **long** to **float**, or **long** to **double**) can result in a loss of precision, but it is still done implicitly when necessary. Notice that converting from **short** to **char** or vice versa is not implicit since there is no appropriate way to deal with values that have the high-bit set (signed in one case, unsigned in the other).

Sphere.java

```
package examples;

public class Sphere {
  public static void main(String[] args) {
    float PI = 3.14F;
    int radius = 6;

    float circumference = radius * 2 * PI;
    float volume = 4.0F / 3.0F * PI * radius * radius * radius;

    System.out.print("A sphere of radius ");
    System.out.print(radius);
    System.out.print(" has a circumference of ");
    System.out.print(circumference);
    System.out.print(" and a volume of ");
    System.out.print(volume);
    System.out.println(".");
  }
}
```

Try It:
Run *Sphere.java* to test implicit type conversions.

# THE CAST OPERATOR

❋ Conversions can be forced by explicitly casting a value or expression to a new data type.

```
float f = 6.5F;
float g = 7.7F;
int total = (int)(f + g);
```

➢ The unary cast operator consists of a datatype in parentheses, followed by a value or expression.

❋ If the value being cast is too large to fit in the smaller location, the bits of the value will be truncated.

```
f = 6.6F;
g = 7.7F;
float fTotal = f + g;     // 14.299999
int iTotal = (int)(f + g);  // 14
```

➢ Losing too many bits with truncation will give you a false result!

```
f = 6666666666.6F;
g = 7777777777.7F;
float fTotal = f + g;     // 1.44444436E10
int iTotal = (int)(f + g);  // 2147483647
```

❋ Only cast when you are <u>sure</u> the result will fit in the new datatype.

❋ To convert to a **String**, do not use the cast operator.

➢ Any type can be converted to a **String** implicity using the + operator where the other operand is a **String**.

```
double d = 12.5;
String s = "" + d;
```

When casting from **double** to **float**, or from **long** to **int**, or from either floating point type to either **long** or **int**, some care is taken to retain both the sign and the magnitude of values. When the original value is too big to fit into the target type, then the appropriate maximum or minimum value of the target type will be used instead.

But this care is not taken when casting to a type smaller than **int**. In such cases, high-order bits are simply chopped off to leave the number of bytes required for the target type. Notice that this operation does not necessarily even preserve the sign of the value:

```
int i = 65157;
short s = (short) i; // s has the value –379  !
```

When casting from floating point to one of the types that are smaller than **int**, the conversion is first to **int**, and then the extraneous bits are chopped off.

# Labs

❶    Write a fuel mileage calculator. Your program should declare variables for miles driven and gallons of fuel consumed. The formula for fuel mileage is:

   fuel mileage = distance driven / fuel consumed

(Solution: *Mileage.java*)

❷    Write a body mass index calculator. Your program should declare variables for height in inches and weight in pounds. The formula for body mass index is.

   body mass index = (weight * 703) / (height * height)

(Solution: *BMI.java*)

❸    Modify your body mass index calculator to store height in two separate variables. One for the person's height in feet and the other for the remaining height in inches.
(Solution: *BMI2.java*)

Other operators not covered in this chapter and what they do:

Bitwise operators deal with each bit independently:

    **~**        unary negation inverts bit values
    **&**        binary AND for corresponding bits of operands
    |        binary OR for corresponding bits of operands
    **^**        binary XOR (exclusive OR, not exponentation) for corresponding bits of operands

Shift operators move the bits of the left operand to the left or right by the number of positions indicated by the right operand:

    <<      binary left shift (toward most significant)
    >>      binary right shift (toward least signifiicant)
    >>>    binary right shift without retaining the sign bit (result is always positive)

# Chapter 6 - Control Flow

<div style="border: 2px solid; border-radius: 20px; padding: 20px;">

## Objectives

❋     Use both simple and compound statements in control flow statements.

❋     Control the flow of program execution using conditional statements and loops.

❋     Use relational and logical operators to test program values.

❋     Test program values in **if** and **switch** conditional statements.

❋     Write programs that use **while**, **do**, and **for** loops to iteratively execute statements and blocks.

</div>

# Statements

❋   A single statement is an expression followed by a semicolon.

```
area = width * length;
System.out.println("area is " + area);
```

➢   The semi colon is a statement terminator.

❋   Curly braces group statements into compound statements, called *blocks*.

```
{
   area = width * length;
   System.out.println("area is " + area);
}
```

➢   Anywhere a single statement can be used, so can a block.

❋   Blocks are usually used in conjunction with conditional and loop statements.

```
if (lineNumber == 60) {
   area = width * length;
   System.out.println("area is " + area);
}
```

# Conditional (if) Statements

❋ An **if** statement allows you to run another statement or block of statements if a condition is **true**.

➢ The condition can use any of the relational operators, and can combine them using logical operators.

```
if (age < 16)
    System.out.println("Cannot drive.");

if (age < 12 && age > 3) {
    System.out.println("Order kid's meals.");
    System.out.println("Buy child movie ticket.");
}
```

❋ Place an **else** after the **if** to run statements when the condition is **false**.

```
if (age < 16)
    System.out.println("Cannot drive.");
else
    System.out.println("Can drive and have a job.");

if (age < 12 && age > 3) {
    System.out.println("Order kid's meals.");
    System.out.println("Buy child movie ticket.");
}
else {
    System.out.println("Pay full price for meals.");
    System.out.println("Buy adult movie ticket.");
}
```

Kids.java
```
...
public class Kids {
   public static void main(String[] args) {
      int age = 9;
      //int age = 14;
      //int age = 23;
      System.out.println("If your age is " + age + " then you:");

      if (age < 16)
         System.out.println("Cannot  drive.");
      else
         System.out.println("Can drive and have a job.");

      if (age < 12 && age > 3) {
         System.out.println("Order  kid's  meals.");
         System.out.println("Buy child movie ticket.");
      }
      else {
         System.out.println("Pay full price for meals.");
         System.out.println("Buy  adult  movie  ticket.");
      }
   }
}
```

### Try It:
Run *Kids.java*. View the output. Change the commented lines to a different age, recompile and run it again.

### Note:
The **else** keyword always corresponds to a preceding **if** in the same block, regardless of how the code is indented. When in doubt, use curly braces.

```
if (age < 12)
  if (age > 3) {
     System.out.println("Order kid's meals.");
     System.out.println("Buy child movie ticket.");
  }
else {
```
Indentation does not match actual function.

Here, the **else** statement block will be executed when **age<=3** and **age<12**, but not executed when **age>=12**.

# Adding an else if

✳ Adding an **else if** to an **if** statement allows you to check additional conditions.

✳ The **if** is evaluated top-to-bottom until one condition evaluates to **true**.

```
if (age < 12 && age > 3) {
    System.out.println("Order kid's meals.");
    System.out.println("Buy child movie ticket.");
}
else if (age >= 12 && age < 16) {
    System.out.println("Cannot drive.");
}
else if (age >= 16) {
    System.out.println("Can drive and have a job.");
    System.out.println("Pay full price for meals.");
    System.out.println("Buy adult movie ticket.");
}
else // babies and toddlers
    System.out.println("Get into movies free.");
```

➢ Only one statement (or block) is executed.

➢ If **age** is **9**, the first block is executed, and the others are all skipped!

✳ An **else** is still optional, and will execute only if none of the conditions are true.

Kids2.java

```java
package examples;

public class Kids2 {
  public static void main(String[] args) {
    int age = 2;
    //int age = 9;
    //int age = 14;
    //int age = 23;
    System.out.println("If your age is " + age + " then you:");

    if (age < 12 && age > 3) {
      System.out.println("Order kid's meals.");
      System.out.println("Buy child movie ticket.");
    }
    else if (age >= 12 && age < 16) {
      System.out.println("Cannot drive.");
    }
    else if (age >= 16) {
      System.out.println("Can drive and have a job.");
      System.out.println("Pay full price for meals.");
      System.out.println("Buy adult movie ticket.");
    }
    else // babies and toddlers
      System.out.println("Get into movies free.");
  }
}
```

## Try It:
This version uses **else if** conditions.

# Conditional (switch) Statements

❋ Use a **switch** statement when you are testing the same expression for several possible literal values.

➢ Each possible value is placed in a **case** label.

❋ Control is transferred to the statements in the **case** that is matched.

```
switch (choice) {
  case 'a':
        System.out.println("Adding");
        break;
  case 'd':
        System.out.println("Deleting");
        break;
  case 'q':
        System.out.println("Quitting");
        break;
  default:
        System.out.println("Invalid selection");
        break;
}
```

❋ The **break** statement keeps execution from continuing into the next **case**.

➢ A **case** without a **break** is said to "fall through" to the next **case** — when it's matched, its statements execute, as well as the next **case**'s and the next, until a **break** is encountered.

❋ Place an optional **default** label after the last **case** to catch any other values (works like the **else** in an **if** statement).

❋ If there is no **default** and none of the **case** labels match, then the program will continue at the statement after the entire **switch.**

Decision.java
```java
package examples;

public class Decision {
  public static void main(String[] args) {

    char choice = 'a';

    switch (choice) {
      case 'a':            // fall through to next case
      case 'A':
         System.out.println("Adding");
        break;
      case 'd':            // fall through to next case
      case 'D':
         System.out.println("Deleting");
        break;
      case 'q':            // fall through to next case
      case 'Q':
         System.out.println("Quitting");
        break;
      default:
         System.out.println("Invalid selection");
        break;
    }
  }
}
```

## Try It:
Run *Decision.java*. Change the **choice** and test it again.

## Note:
**switch** statements can only compare expressions of type **byte**, **short**, **int**, **char**, and **String**. Java 5.0 also added **enum**s to this list.

## WHILE AND DO-WHILE LOOPS

✳    The most basic looping structure is a **while** loop.

```
while (age < retirementAge) {
     System.out.println("Work another year");
     age++;
}
```

➢    The expression in parentheses following **while** (the *condition*) must yield a **boolean** result.

➢    The statements in the loop (the *loop body*) are run over and over as long as the condition evaluates to **true**.

➢    A single statement or a block of statements may be used with a **while** loop.

```
while (age < retirementAge)
     System.out.println("You are " + age++);
```

➢    One of the expressions in the loop body will usually modify a value being tested in the condition; otherwise the loop may never end!

✳    A **do-while** loop is similar to a **while** loop, but the condition is tested after the statements are executed.

➢    This means that the loop will always execute at least one time.

```
int curRow = 1;
do
     System.out.println(curRow++);
while (curRow <= 100);
```

▪    Note the required semicolon following the **while()** condition.

FahrToCel.java

```java
package examples;

public class FahrToCel {
  public static void main(String[] args) {
    int fahr = 0;
    int cel = 0;
    double round = 0.5;          // set to 0 to avoid rounding

     System.out.println("Fahrenheit\tCelsius");

    while (fahr <= 300) {
      cel = (int) ((5.0 / 9.0) * (fahr - 32.0) + round);
      System.out.println(fahr + "\t\t" + cel);
      fahr += 20;
    }
  }
}
```

## Try It:

*FahrToCel.java* uses a **while** loop to display a table of Fahrenheit to Celsius conversions.

# for Loops

❋ Use a **for** loop to iterate through statements a specific number of times.

```
for (i = 0; i < 5; i++) {
    System.out.println(i);
    System.out.println(" squared is ");
    System.out.println(i*i);
}
```

➢ The first expression is evaluated only once; this is for initialization.

▪ A variable can be declared in the initialization portion and is usable only inside the loop.

```
for (int i = 0; i < 5; i++) {
```

➢ The second expression is a condition that is evaluated before each iteration of the loop (including the first); this is a *test condition*.

▪ While the condition is **true**, the statements in the loop will be executed.

➢ The third expression is executed at the end of each loop; this is used to increment a value.

❋ It's very easy to read a **for** loop to see how many times it will be run.

➢ Our example loop runs 5 times; **i** has the values **0**, **1**, **2**, **3**, **4**.

FahrToCel2.java

```java
package examples;

public class FahrToCel2 {
   public static void main(String[] args) {
      int fahr = 0;
      int cel = 0;
      double round = 0.5;                  // set to 0 to avoid rounding

      System.out.println("Fahrenheit\tCelsius");

      for (fahr = 0; fahr <= 300; fahr += 20) {
         cel = (int) ((5.0 / 9.0) * (fahr – 32.0) + round);
         System.out.println(fahr + "\t\t" + cel);
      }
   }
}
```

## Try It:
This version of *FahrToCel* uses a **for** loop to display the conversion table.

# A FOR LOOP DIAGRAM

❋    Consider the following loop:

```
for (i = 0; i < 20; i++) {
     total += i;
}
```



|  |  |
|---|---|
| i = 0 | *initialization* |
| i < 20   *false* | *test condition* |
| *true* |  |
| total += i; | *statement(s)* |
| i++ | *increment* |
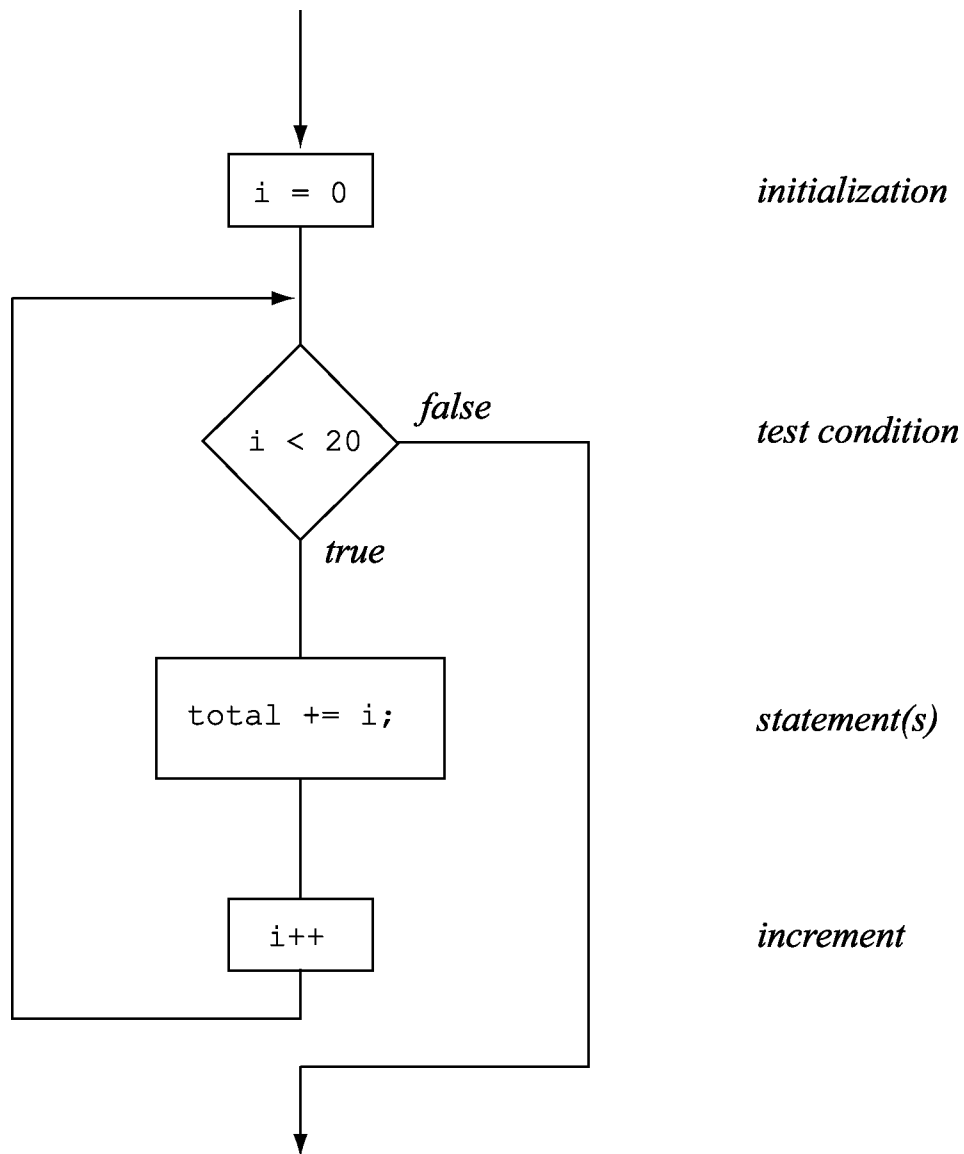
The general form of the **for** loop:

```
for (expr₁; expr₂; expr₃)
    loopbody;
```

is almost the same as a corresponding **while** loop of this form:

```
expr₁;
while (expr₂) {
  loopbody;
  expr₃;
}
```

The **for** loop differs in two ways:

1. **$expr_1$** and **$expr_3$** can each be a sequence of expressions separated by commas.

2. Any **continue** statement (discussed next) within the **loopbody** jumps to the **$expr_3$** before returning to the **$expr_2$**.

# Enhanced for Loop

❋ To loop through an array, you can use the array's built-in **length** field to determine the size of the array.

```
int odds[] = {1,3,5,7,9};
for(int n = 0; n < odds.length; n++){
   System.out.println(odds[n]);
}
```

❋ Java 5.0 added the *enhanced for* loop to make it easier to step through a collection, such as an array.

```
for (declaration : expression)
   statement;
```

➢ The variable defined in the declaration should be the same type as your array's elements.

➢ The expression is simply the name of the array itself.

```
for (int n : odds){
   System.out.println(n);
}
```

➢ The enhanced for loop is sometimes called the *for each* loop.

ArrayTest.java

```
...
public class ArrayTest {
   public static void main(String args[]) {
      int odds[] = {1,3,5,7,9};
      String bookInfo[][] = {
         {"Java In a Nutshell", "Flanagan"},
         {"Core Java", "Horstmann"},
         {"Thinking in Java", "Eckel"}
      };

      for (int i = 0; i < odds.length; i++) {
         System.out.println(odds[i]);
      }

      for (int i = 0; i < bookInfo.length; i++) {
         for (int j = 0; j < bookInfo[i].length; j++) {
            System.out.print(bookInfo[i][j] + " ");
         }
         System.out.println();
      }
   }
}
```

ArrayTest2.java

```
...
public class ArrayTest2 {
   public static void main(String args[]) {
      int odds[] = {1,3,5,7,9};
      String bookInfo[][] = {
         {"Java In a Nutshell", "Flanagan"},
         {"Core Java", "Horstmann"},
         {"Thinking in Java", "Eckel"}
      };

      for (int num : odds) {
         System.out.println(num);
      }

      for (String[] book : bookInfo) {
         for (String s : book) {
            System.out.print(s + " ");
         }
         System.out.println();
      }
   }
}
```

# The continue Statement

✷ Use a **continue** statement when you would like to jump to the next iteration of your loop.

    ➢ For **while** and **do** loops, the **continue** causes the loop test condition to be executed.

    ➢ In a **for** loop, a **continue** causes the increment and then the loop test condition to be executed.

✷ **continue** can usually be replaced with a large, but perhaps cumbersome, **if** statement.

With a **continue** statement:

Work.java
```java
package examples;

public class Work {
  public static void main(String[] args) {
    int day;
    for (day = 1; day <= 7; day++) {
      if (day == 6 || day == 7) {
        System.out.println("It's a weekend!");
        continue;
      }
      System.out.println("Get up early.");
      System.out.println("Go to work.");
      System.out.println("Sit in meetings.");
    }
  }
}
```

Rewritten without a **continue** statement:

Work2.java
```java
package examples;

public class Work2 {
  public static void main(String[] args) {
    int day;
    for (day = 1; day <= 7; day++) {
      if (day == 6 || day == 7) {
        System.out.println("It's a weekend!");
      }
      else {
        System.out.println("Get up early.");
        System.out.println("Go to work.");
        System.out.println("Sit in meetings.");
      }
    }
  }
}
```

The **else** replaces the **continue**.

## Try It:
Is there a difference in the output of these two programs?

# The break Statement

❋ The **break** immediately exits any loop**.**

❋ **break** is almost always placed inside of an **if** statement.

```
while (true) {
   choice = myMenu.displayIt();
   if (choice == 'q' || choice == 'Q')
      break;
   myDatabase.processIt(choice);
}
```

❋ Most loops can be written without the use of a **break** statement.

```
choice = myMenu.displayIt();
while (choice != 'q' && choice != 'Q') {
   myDatabase.processIt(choice);
   choice = myMenu.displayIt();
}
```

You can precede any statement with a label:
```
Label: statement;
```

You can use a label as the target for a **break** or **continue**:

```
break [Label];
```
Transfers control to the end of the innermost enclosing loop, enclosing statement block, or switch, or to the end of the loop preceded by the specified label.

```
continue [Label];
```
Transfers to the next iteration of the innermost loop, or of the loop preceded by the specified label. Does not break out of iteration.

Example of **break** and **continue**:
```
outer:
for (int i = 0; i < 5; i++) {
  inner:
  for (int j = 0; j<10; j++) {
    if (i == 2 && j == 5)
      continue outer;
    else if (i == 3 && j == 5)
      break;
    System.out.println("i = " + i + "j = " + j);
  }
}
```

The **continue** will resume the next iteration of the outer loop, including the increment of **i**.

The **break** will immediately exit the inner loop and resume the next iteration of the outer loop (giving very similar results, in this case!).

Investigate:

Examine the file *LookupAge1.java* to see a double-nested **for** loop that makes good use of a **continue** statement with a label. How might you eliminate its use? Notice the statement after the inner **for** loop; when is it intended to execute?

Now examine the file *LookupAge2.java* to see the outer **for** loop replaced with a **foreach** loop. Could you also replace the inner **for** loop with a **foreach** loop? If not, why?

# Labs

❶ Modify your body mass index (BMI) calculator from the previous lab and print out "Underweight" if the BMI is below 18.5, "Normal" for values between 18.5 and 24.9, "Overweight" for values between 25 and 29.9, and "Obese" for values above 30.
(Solution: *BMI3.java*)

❷ Write a program that declares an array of five integers. Use a loop to determine the smallest and largest number in the array and print them.
(Solution: *MinMax.java*)

❸ Modify your solution to ❷ to work with an array of any size.
(Solution: *MinMax2.java*)

❹ (Optional) Write a program to print the reverse of any integer value. For example, for the integer 123, the program should print 321.
(Solution: *Reverse.java*)

# CHAPTER 7 - METHODS

**OBJECTIVES**

❋ Call methods.

❋ Pass parameters to methods and use method return values.

❋ Define methods.

# Methods

❋ A *method* is a set of statements, with a name, that is defined in a class.

➢ There are no functions, subroutines, or procedures in Java except for methods.

➢ Methods are where all the work gets done in Java.

❋ You "call" a method, passing it data if necessary.

```
System.out.println("println is a method.");
```

➢ A method can be called from different places with different data.

❋ A method can "return" data to its caller.

```
root = Math.sqrt(2.0);
```

❋ The **main()** method of your class is where Java starts your program's execution.

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

➢ If your class doesn't have a properly declared **main()** method, Java can't start your program.

➢ If your **main()** method doesn't do anything, your program won't do anything.

A Java program contains a **public** class with a **public** method, return type **void**, named **main()**. The Virtual Machine loads the compiled class (and any other classes that this class uses). The Virtual Machine begins execution by invoking the **main()** method of the class. The Virtual Machine copies runtime arguments from the Java program's command line into an array of **String**s, which is **main**'s only parameter. Most programmers name the array **arg** or **args**.

If **main()** is declared  incorrectly, the Virtual Machine won't recognize it, and therefore won't be able to start your application:

- **public**         So that the VM has access to the **main()** method.
- **static**         **main()** resides in its class, not in objects of its class.  The VM doesn't
                     need to create an object before invoking **main()**.
- **void**           **main()** doesn't return a value.
- **main**           The name must be exact, not **Main**, **start**, or anything else.
- **String args[]**  **main()** must be declared with exactly one parameter, an array of
                     **Strings**.  This parameter must exist, even if you don't use the arguments
                     in your code.  It can be declared as **String[] args**.

Consider, for example, the following Java program:
Echo.java
```
...
public class Echo {
  public static void main(String args[]) {
     System.out.println(args[0]);
  }
}
```

You can compile it with the command:

```
javac Echo.java
```

and run it with a command such as this:

```
java Echo   "first argument"
```

### Try It:
Edit *Echo.java* to attempt some minor change to the code, such as removing the keyword **static**, or changing the type of the array **args[]** from **String** to **int**.  Now try running it. The Virtual Machine should not be able locate the **main()** method, resulting in an error.

The reason that the **main()** method must strictly have the correct form is due to the fact that Java allows for overloading of methods, as described on the next page.

# CALLING METHODS

❋   A method call might make up a complete statement by itself.

```
System.out.println("Pick a number from 1 to 10.");
```

❋   The return value of a method can be assigned to a variable.

```
root = Math.sqrt(2.0);
```

❋   A method that returns a value can be used in an expression.

```
num = (int)(Math.random() * 10.0) + 1;
```

❋   *Arguments* you pass to methods can be literals, variables, or even expressions (including other method calls!).

```
hyp = Math.sqrt((a * a) + (b * b)); //One argument!
System.out.println(Math.sqrt(2.0));
```

❋   If there is more than one method with the same name, then the types of the arguments provided are used to determine which method to call.

➢   The ability to have more than one method with the same name, but that take different types of arguments, is called *overloading*.

▪   We can pass different types of arguments to **System.out.println()**, because the **println()** method is overloaded.

Here, the **System.out.println**() method is called repeatedly, each time with a different type of argument. Actually, there are four separate methods being called that share the same name. The **println**() method is overloaded.

PrintMax.java
```
package examples;

public class PrintMax {
  public static void main(String[] args) {
    int iMax = 2147483647;
    long lMax = 9223372036854775807L;
    float fMax = 3.40282347e38F;
    double dMax = 1.79769313486231570e308;

    System.out.println(iMax);
    System.out.println(lMax);
    System.out.println(fMax);
    System.out.println(dMax);
  }
}
```

## Try It:
Run *PrintMax.java* to see overloaded methods in action.

# Defining Methods

❋ To define a method, you must specify:

➢ The datatype that the method returns.

➢ The method's name.

➢ The types of all the arguments that must be passed to the method when it is called.

```
float getRectangleArea(float width, float height) {
   float area = width * height;
   return area;
}
```

❋ Your method can contain a **return** statement that returns the correct type of data.

```
return area;
```

➢ The **return** statement can return the result of an expression.

```
return width * height;
```

❋ If your method doesn't return a value, declare its return type as **void**.

```
void printOptions(short appMode) {
   ...
   return;
}
```

➢ You usually omit the **return** statement if it would be the last statement in the method body.

# METHOD PARAMETERS

❋ Each *parameter* in your method definition has a name and a datatype.

```
float getRectangleArea(float width, float height) {
   float area = width * height;
   return area;
}
```

➢ Within the method, the parameters act as local variables.

❋ The arguments passed to your method when it's called are copied into the parameters.

```
float a1 = getRectangleArea(24.0F, 37.5F);
float a2 = getRectangleArea(72.0F, 80.0F);
```

➢ The method can then operate on the data in the parameters.

❋ A method can take no parameters at all.

```
int getMenuChoice() { ... }
```

SphereMeth.java

```
package examples;

public class SphereMeth {
  static float PI = 3.14159F;

  public static void main(String[] args) {
    int radius = 6;
    float circum = getCircum(radius);
    float volume = getVolume(radius);

    System.out.print("A sphere of radius ");
    System.out.print(radius);
    System.out.print(" has a circumference of ");
    System.out.print(circum);
    System.out.print(" and a volume of ");
    System.out.print(volume);
    System.out.println(".");
  }

  static float getCircum(float rad) {
    return rad * 2 * PI;
  }

  static float getVolume(float rad) {
    float   vol = (4.0F / 3.0F) * PI * (float) Math.pow(rad, 3.0);
    return vol;
  }
}
```

## Try It:

This example uses methods to calculate the circumference and volume of a sphere.

# Scope

✳  A method's parameters, as well as any variables declared within the method body, are *local* to the method.

    ➢  No other method can use those parameter or variable values.

✳  In fact, any variable declared in a block is local to that block.

    ➢  The variable is created for each execution of its containing block, then destroyed at the end of that block.

        ▪  Java allows *recursion*: a method can call itself directly or indirectly; the method's local variables will exist separately for each call.

✳  Variables declared outside of any block of code are called *fields*.

    ➢  All methods can use the values of fields in their class.

Local.java

```java
package examples;

public class Local {
  // field
  static int total;

  public static void main(String[] args) {
    int count; // count is local to main

    for (count = 0; count <= 10; count++) {
      total += count;
    }

    if (total > 20) {
      int half;  // half is local to the if block
      half = total / 2;
      System.out.println("total=" + total + ", half=" + half);
    }
    // The following line will not compile
    // System.out.println("total=" + total + ", half=" + half);

    System.out.println("total=" + total);

    printTotal();
  }

  static void printTotal() {
    System.out.println("printTotal: total=" + total);
  }
}
```

Uncomment this line to see a compiler error.

## Try It:

*Local.java* illustrates differences in scope. Uncomment the line above and try to compile it. What does the error message say?


## Investigate:

Examine the file *FibRecurse.java* to see an example of a recursive method. Compile and run the program. Every recursive method should have a well-defined *terminating condition* to ensure that recursion is not infinite. What is the terminating condition of the **fib()** method?

# So, Why All the static?

❋ A method declared **static** can use only those fields that are also declared **static**.

❋ A method declared **static** can directly call other methods only if they are also declared **static**.

❋ If a method is declared without **static**, then you can not call that method until you create an object.

> ➢ You make the object invoke the method.

> ➢ The non-**static** method can then use any non-**static** fields, because non-**static** fields belong to the object itself.

> ➢ If multiple objects are created, each has its own copies of all the non-**static** fields.

❋ **Static** fields and methods reside in the class itself, and are usable as soon as your program starts and the class is loaded.

❋ Non-**static** fields and methods reside in the individual objects, which must be created before their methods can be called and their fields used.

> ➢ Since an object is an instance of its class, the term *instance* is sometimes used as a synonym for non-**static**.

SphereObj.java

```
...
public class SphereObj {
  int  radius;  // instance field
  static float PI = (float) Math.PI;  // static field

  public static void main(String[] args) {
    SphereObj  s = new SphereObj();
    s.radius = 6;

    float circum = s.getCircum();
    float volume = s.getVolume();

    System.out.print("A sphere of radius ");
    System.out.print(s.radius);
    System.out.print(" has a circumference of ");
    System.out.print(circum);
    System.out.print(" and a volume of ");
    System.out.print(volume);
    System.out.println(".");
  }

  float getCircum() {
    return radius * 2 * PI;
  }
  ...
}
```

This is where the **SphereObj** object is created.

## Try It:

This example creates a **SphereObj** object and uses it to set the **radius**, and call **getCircum()** and **getVolume()**.

When calling a **static** method in a different class, the method name must be qualified with the class name first (and the dot operator), as illustrated here in calling **random()**:

MethCall.java

```
...
public class MethCall {
  public static void main(String[] args) {
    System.out.println("Pick a number from one to ten.");
    int num = (int) (Math.random() * 10.0) + 1;
    System.out.println("Is it " + num + "?");
  }
}
```

# Labs

❶ Convert your Body Mass Index program from a previous lab to use a method to calculate the BMI. The method should take two parameters for the height and weight and return back the calculated BMI.

Give your method and its parameters appropriate names. Declare the method as **static** so that you can test it directly from **main()**. In **main()**, use the method to perform several BMI calculations, printing out the results.
(Solution: *BMIMeth.java*)

❷ Modify your solution to ❶ by creating another **static** method to determine whether the passed in BMI is "Underweight", "Normal", etc. Your new method should return a **String**. Call your method from within **System.out.println()**.
(Solution: *BMIMeth2.java*)

# Chapter 8 - Objects and Classes

## Objectives

❋        Create Java classes.

❋        Use your classes to instantiate objects.

❋        Define instance data and class data.

❋        Write and use methods.

❋        Use constructors to initialize your objects.

❋        Control the visibility of data members and
         methods.

# Defining a Class

❋   <u>All</u> fields and methods in Java are defined within *classes*.

❋   A class describes something you want to write programs about.

➢   Real objects or entities.

➢   Business processes or concepts.

❋   The fields and methods of a class should model the real-world object to the extent necessary for use in applications.

➢   *Object-Oriented Analysis and Design* (OOAD) is a formal discipline for the correct design of classes.

❋   Each **public** class must be defined in its own *.java* file.

➢   There can be additional classes in the same file, but none of them can be declared **public**.

➢   Normally, you'll create a separate *.java* file for every class you define.

Account1.java

```
package examples;

public class Account1 {
    double balance;

    void deposit(double amount) {
        balance = balance + amount;
    }

    void withdraw(double amount) {
        balance = balance - amount;
    }
}
```

Customer1.java

```
package examples;

public class Customer1 {
    String firstName;
    String lastName;
    Account1 acc;
}
```

# Creating an Object

❋ Using a class, you can create objects.

➢ An *object* is the actual representation of a distinct, individual entity of the type described by the class.

❋ Creating an object is also called *instantiation*, and an object is often referred to as an *instance* of the class.

❋ The standard way to instantiate an object is to use the **new** operator:

```
Dog spot = new Dog();
```

➢ This allocates storage space for all non-static fields of the object, initializes them, and executes one or more constructors; more on all these shortly.

❋ The **new** operator returns a reference to the newly-created object.

➢ You must assign this reference to a variable of the appropriate type: a variable declared using the name of the class.

➢ Declaring a variable of a class type (that is, a *reference variable*) does not create an object.

```
Dog fido; // No Dog exists yet.
fido = new Dog();  // A Dog, referred to by fido.
```

➢ You can have more than one reference to an object.

```
Dog spot = fido;
```

➢ If you don't assign the reference to a variable, the object still exists, at least temporarily.

```
new Dog();  // A stray Dog...
```

Account2.java

```
...
public class Account2 {
    double balance;

    void deposit(double amount) {
        balance = balance + amount;
    }

    void withdraw(double amount) {
        balance = balance – amount;
    }
}
```

Customer2.java

```
...
public class Customer2 {
    String firstName;
    String lastName;
    Account2 acc;

    void addAccount(double initialBalance) {
        acc = new Account2();
        acc.balance = initialBalance;
        System.out.println("Account added for " + firstName + " "
                + lastName);
    }
}
```

Bank2.java

```
...
public class Bank2 {

    public static void main(String[] args) {
        Customer2 cust = new Customer2();
        cust.firstName = "Jim";
        cust.lastName = "Stewart";

        cust.addAccount(250.0);
    }
}
```

## Investigate:
If we load and run the *Bank2* class, how many objects will exist?

# Instance Data and Class Data

�֍ Fields declared as **static** are *class fields*.

➢ Class fields belong to the class as a whole and not to any particular instance.

➢ You can reference class fields even if no object of the class has been instantiated.

▪ You can also refer to a class field through any instance of the class.

✖ Fields <u>not</u> declared as **static** are *instance fields*.

➢ Each object has its own copies of all the instance fields defined in its class.

➢ You can optionally use the **this** keyword to reference a field.

➢ You cannot refer to an instance field without an object.

✖ Fields modified with **final** cannot be changed once they have been set.

➢ **final** fields must be initialized when declared, or in <u>every</u> constructor of the class.

➢ Most **final** fields are defined as **static** as well (since instances do not need their own copies).

▪ **java.lang.Math** defines *pi* and *e* in this way (**Math.PI** and **Math.E**).

▪ By convention, **static final** fields appear in all capital lettering.

Account3.java

static final field is declared with all capital letters.

```
...
public class Account3 {
    double balance;
    String accountId;
    static int nextId = 0;
    static final int ROUTING_NUMBER = 123456789;

    void deposit(double amount) {
        balance = balance + amount;
    }

    void withdraw(double amount) {
        balance = balance - amount;
    }
}
```

Customer3.java

Use the object reference to access the instance field.

Use the class to access the static field.

```
...
public class Customer3 {
    String firstName;
    String lastName;
    Account3 acc;

    void addAccount(double initialBalance) {
        acc = new Account3();
        acc.accountId = "ACCT-" + Account3.nextId++;
        acc.balance = initialBalance;
        System.out.println(acc.accountId + " added");
    }
}
```

Bank3.java

```
...
public class Bank3 {

    public static void main(String[] args) {
        Customer3 cust = new Customer3();
        cust.firstName = "Jim";
        cust.lastName = "Stewart";
        cust.addAccount(250.0);

        Customer3 cust2 = new Customer3();
        cust2.firstName = "Joan";
        cust2.lastName = "Stewart";
        cust2.addAccount(500.0);
    }
}
```

# Methods

✳  All activity in a Java program occurs in methods of classes or objects.

✳  A method can return a value of any type, including a primitive datatype, an object reference, or an array of either.

➢  If a method doesn't need to return a value, declare its return type as **void**.

✳  The name of the method, along with its list of parameter datatypes, is termed the method's *signature*.

➢  A class can have many methods of the same name, as long as they all have different signatures (*overloading*).

➢  The arguments you pass determine which method is called.

➢  The method's return type is not part of the signature.

**Chapter 8**                                          **Objects and Classes**

Account4.java

```
...
    void display() {
         System.out.println("**Account Information**");
         System.out.println("  ID: " + accountId);
         System.out.println("  Balance: " + balance);
    }

    static String getNextId() {
        return "ACCT-" + Account4.nextId++;
    }
}
```

> **static** method

> Use the **return** keyword in non-**void** methods.

Customer4.java

```
...
    void addAccount() {
        acc = new Account4();
        acc.accountId = Account4.getNextId();
        acc.balance = 0;
        acc.display();
    }

    void addAccount(double initialBalance) {
        acc = new Account4();
        acc.accountId = Account4.getNextId();
        acc.balance = initialBalance;
        acc.display();
    }
}
```

> Overloaded method

Bank4.java

```
...
public class Bank4 {

    public static void main(String[] args) {
        Customer4 cust = new Customer4();
        cust.firstName = "Jim";
        cust.lastName = "Stewart";
        cust.addAccount();

        Customer4 cust2 = new Customer4();
        cust2.firstName = "Joan";
        cust2.lastName = "Stewart";
        cust2.addAccount(500.0);
    }
}
```

> Which version of **addAccount()** is called here?

# Constructors

❋ When you create an object, a special method called a *constructor* automatically initializes the object's data members.

➢ A constructor looks like a method with the same name as the class of which it is a member.

➢ A constructor can take parameters and be overloaded, but cannot have a return type.

➢ A constructor with no parameters defined is sometimes called a *no-arg* constructor for the class.

❋ The **new** operator, followed by a constructor of the class, is what instantiates an object of that class.

❋ A constructor can call another constructor by using the keyword **this** as a method name.

➢ If you use it, **this()** must be the first statement in the constructor.

➢ Use **this()** to reduce code duplication.

❋ If you do not write any constructors for your class, the compiler will provide a *default no-arg constructor*.

➢ As soon as you provide <u>any</u> constructor, the compiler will no longer provide the default one for you.

Account5.java

```
...
public class Account5 {
    double balance;
    String accountId;
    ...
    Account5() {
        this(0.0);
    }

    Account5(double initialBalance) {
        balance = initialBalance;
        accountId = Account5.getNextId();
    }
  ...
}
```

Contructor is now responsible for setting the **accountId**.

Customer5.java

```
...
public class Customer5 {
    String firstName;
    String lastName;
    Account5 acc;

    Customer5(String fName, String lName) {
        this(fName, lName, 0.0);
    }

    Customer5(String fName, String lName, double initialBalance) {
        firstName = fName;
        lastName = lName;
        acc = new Account5(initialBalance);
        acc.display();
    }
}
```

Constructors do not have a return type; not even **void**.

Bank5.java

```
...
public class Bank5 {

    public static void main(String[] args) {
        new Customer5("Jim", "Stewart");
        new Customer5("Joan", "Stewart", 500.0);
    }
}
```

# Access Modifiers

❋ The *scope* of an object, method, or class refers to its accessibility and visibility.

➢ Scope can also refer to the lifetime of an object; we are concerned with visibility here.

❋ The access control of a method or a field can be specified as **public**, **protected**, **private**, or left unspecified.

**public** Methods of any class anywhere have access.

**private** Can only be accessed in methods defined in the same class.

**protected** Methods of subclasses and of any class in the same package (directory) have access.

➢ A member with unspecified access control is visible to methods of any class in the same package (directory): it has *default* or *package* access.

❋ The access of a class itself can be specified as **public**, or left unspecified (in which case it is visible throughout its package).

❋ In most cases, your constructors will have the same access as the class they are defined in.

Customer6.java
```java
package examples;

public class Customer6 {
    private String firstName;
    private String lastName;
    private Account6 acc;

    public Customer6(String fName, String lName) {
        this(fName, lName, 0.0);
    }

    public Customer6(String fName, String lName, double initialBalance){
        firstName = fName;
        lastName = lName;
        acc = new Account6(initialBalance);
        acc.display();
    }

    public String getName() {
        return firstName + " " + lastName;
    }

    public Account6 getAccount() {
        return acc;
    }
}
```

# Encapsulation

❋ One of the most important features of Object-Oriented Programming is encapsulation.

➤ *Encapsulation* groups fields and methods into a class so that the fields are only accessible through a **public** interface.

❋ Encapsulation insulates changes from propagating throughout your program.

➤ If a **private** part is changed, that change is constrained to the class in which it was defined.

❋ Because of encapsulation, you should:

➤ Give fields and methods the most restricted access possible.

```
private String accountId;
```

➤ Make all fields **private** and provide non-**private get** and **set** methods.

▪ A **get** method returns the value of the field.

```
public String getAccountId() {
   return accountId;
}
```

▪ A **set** method modifies the value of the field.

```
public void setAccountId(String id) {
   accountId = id;
}
```

Account6.java
```java
package examples;

public class Account6 {
    private double balance;
    private String accountId;
    private static int nextId = 0;
    public static final int ROUTING_NUMBER = 123456789;

    public Account6() {
        this(0.0);
    }

    public Account6(double initialBalance) {
        balance = initialBalance;
         this.setAccountId(Account6.getNextId());
    }

    public String getAccountId() {
        return accountId;
    }

    private void setAccountId(String id) {
        accountId = id;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        balance = balance + amount;
    }

    public void withdraw(double amount) {
        balance = balance - amount;
    }

    public void display() {
         System.out.println("**Account Information**");
         System.out.println("  ID: " + accountId);
         System.out.println("  Balance: " + balance);
    }
    ...
}
```

> The return type of the get method must match the parameter type of the set method.

# Labs

❶    Write an **Employee** class with appropriate **public** fields (first name, last name, salary, employee id). Make sure it compiles. Write an **EmpTest** class whose **main**() method instantiates an **Employee** object, and assigns values for all of the object's fields.
(Solutions: *Employee1.java*, *EmpTest1.java*)

❷    Add a **showEmp()** method to the **Employee** class. It should print out the values of the **Employee** object's fields. Call this method at the end of **EmpTest**'s **main()** to display your employee.
(Solutions: *Employee2.java*, *EmpTest2.java*)

❸    (Optional) Add two **static** fields to your **Employee** class: minimum wage and retirement age. Add them to the output of your **Employee** class's **showEmp()** method.
(Solutions: *Employee3.java*, *EmpTest3.java*)

❹    Make the four instance fields in your **Employee** class **private**. Add public **get** and **set** methods for each of them. Modify **EmpTest** to use the **set** methods to assign values for the object's fields.
(Solutions: *Employee4.java, EmpTest4.java*)

❺    Add a constructor to your **Employee** class, which takes two **String** arguments: first name and last name. In **EmpTest**'s **main()**, use this constructor to instantiate your employee, rather than setting the name fields after instantiation. Test to make sure this works. Add a no-arg constructor to the **Employee** class. It should invoke the other constructor, passing the default values **J.** and **Doe.** Test this too.
(Solutions: *Employee5.java*, *EmpTest5.java*)

# Chapter 9 - Using Java Objects

## Objectives

❋ Format your output with **System.out.printf**().

❋ Know when to use **StringBuffer** and **StringBuilder** instead of **String**.

❋ Use **toString()** to return your object's state.

❋ Perform comparisons of objects and of object references.

❋ Use the wrapper classes Java provides for the primitive datatypes.

❋ Declare a typesafe enumeration of values with the **enum** keyword.

# Printing to the Console

❋    **System.out.println()** prints to standard out, appending the platform-specific newline character(s).

    ➢    **System.out.print()** omits the newline.

❋    **System.out.printf()** allows you to print with formatting (similar to C).

    ➢    The first argument to **printf()** is the format string.

    ➢    Additional arguments specify what to print.

```
String formatString = "The result of averaging %1$d" +
    " %2$d and %3$d is %4$.2f";
System.out.printf(formatString, 12, 4, 6,
   ((12 + 4 + 6)/3.0));
```

Like other languages, in Java you can declare methods which take *variable arguments* (***varargs***). **printf**() uses this feature.

```
public PrintStream printf(String format, Object... args)
```

The **...** indicates that zero or more **Object**s can be passed in after the format string. Behind the scenes, **printf**()'s code treats **args** as an array of **Object**s.

# printf Format Strings

❋ The format string is made up of fixed text, as well as one or more format specifiers.

➢ For character and numeric types, the format specifier must follow a pre-defined syntax.

```
%[argument_index$][flags][width][.precision]conversion
```

▪ Each format specifier always starts with a leading percent sign.

➢ *argument_index* indicates the position of the argument in the argument list.

▪ **1$** is the first argument, **2$** is the second, etc.

➢ *flags* is used to modify the format of the output.

▪ For example, the **-** flag specifies that the output should be left justified.

➢ *width* indicates the minimum number of characters to output.

➢ *precision* is used to restrict the number of characters to output.

▪ For floating point numbers, this is the number of digits after the decimal.

▪ *precision* is not applicable to integers and characters.

➢ *conversion* is a required character that specifies how the argument should be formatted.

▪ For example, the **f** conversion indicates that a floating point decimal value is being formatted.

Some common conversion characters:
**b**   boolean
**s**   String
**d**   decimal integer
**x**   hex integer
**f**   decimal floating point
**a**   hex floating point
**n**   line separator

Some common flags:
**-**   left justify
**+**   signed
**0**   zero padded
**(**   negative numbers in parentheses

For more conversion characters and flags see the Java API documentation for **java.util.Formatter**.

PrintToConsole.java

```
package examples;

public class PrintToConsole {

    public static void main(String[] args) {
        System.out
                .printf("PI to 10 decimal places: %1$.10f%n", Math.PI);

        int a = 17;
        int b = -445;

        System.out.printf("Width flag set to 15: %1$15d%n", a);
        System.out.printf("Decimal value: %1$d, hex value: %1$x%n", a);
        System.out.printf("Negative in parens: %1$(d%n", b);

        String fs = "The average of %1$d, %2$d, and %3$d is %4$.2f%n";
        System.out.printf(fs, 12, 4, 6, ((12 + 4 + 6) / 3.0));

    }
}
```

**Try It:**
Run *PrintToConsole.java* to see the formatted output.

# StringBuilder and StringBuffer

❊   **String**s are immutable.

   ➤   Concatenation of **String**s results in new object creation.

```
String s = "Java";
s = s + "Programming";
```

❊   To avoid extra memory allocation, use a mutable version of **String**:
**StringBuilder** or **StringBuffer**.

   ➤   Both **StringBuilder** and **StringBuffer** have handy **append()** and **insert()**
methods for **String** modification.

```
StringBuilder b = new StringBuilder("Java");
b.append(" Programming");
```

   ➤   Use the **toString()** method to convert back to a **String**.

```
String s = b.toString();
```

❊   Use **StringBuffer** instead of **StringBuilder** in a multi-threaded application.

   ➤   **StringBuffer**'s methods are **synchronized** for thread safety.

      ▪   This also makes **StringBuffer** slower than **StringBuilder** in non-
threaded applications.

HTMLGenerator.java
```java
package examples;

public class HTMLGenerator {

    public static String generateHTML(String title) {
        StringBuilder builder = new StringBuilder(100);

        builder.append("<html>");
        builder.append("<head><title>");
        builder.append(title);
        builder.append("</title></head>");
        builder.append("<body>");
        builder.append("<h1>");
        builder.append(title);
        builder.append("</h1>");
        builder.append("</body>");
        builder.append("</html>");

        return builder.toString();
    }

    public static void main(String[] args) {
        System.out.println(generateHTML("The Web Page"));
    }
}
```

> You can pass an initial size to the constructor.

# Methods and Messages

❋ The terms "calling a method" and "sending a message" are equivalent (but reflect different programming perspectives).

➢ When you call a method, the object to which the method belongs is referred to as the *invoking object* (if you think in terms of calling methods), or the *target object* (if you think in terms of sending messages).

```
//rectangle is getArea's invoking object
int a = rectangle.getArea();

//rectangle is the target of getHeight
int h = rectangle.getHeight();
```

➢ The invoking object's method has **private** access to all of the invoking object's fields and methods.

❋ Java syntax allows cascaded messages: methods returning object references may be cascaded in one statement.

```
String s = emp.getFirstName();
System.out.println (s.toUpperCase());
```

Can be shortened to:

```
System.out.println (emp.getFirstName().toUpperCase());
```

When you run **java HelloWorld**, the VM loads the file *HelloWorld.class* and automatically sends the message **main**, passing a **String** array, to the **HelloWorld** class. The **HelloWorld** class' **main()** method sends the message **println** to the object **System.out**.

HelloWorld.java
```
package examples;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

In addition to **static**, other method declaration modifiers include **abstract**, **final**, **synchronized**, and **native**. **abstract** and **final** have to do with inheritance. The **synchronized** modifier has to do with threads. **native** methods are methods declared in a Java class, but written in another programming language.

# toString

❋ Add the **toString()** method to your class to make it easier to print your object's state.

```
public String toString() {
   return "Firstname: " + fname + "Lastname: " + lname;
}
```

➢ **toString()** is called automatically in situations where an object must be treated as, or converted to, a **String**.

▪ When an object is passed directly to **print()** or **println()**.

▪ When an object is concatenated directly to a **String** with the + operator.

➢ Your **toString()** must be **public**, return a **String**, and take no parameters.

❋ Printing an object without a **toString()** results in default behavior.

➢ By default, Java's **toString()** implementation returns the name of the class, an @ symbol, and the hashcode for this object.

❋ Integrated development environments, like Eclipse, can generate the **toString()** implementation for you.

Rectangle.java

```
...
public class Rectangle {
    private int width;
    private int height;

    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    public String toString() {
        return "Rectangle [width=" + width + ", height=" + height + "]";
    }
    ...
}
```

Square.java

```
...
public class Square {
    private int length;

    Square(int len) {
        length = len;
    }
    ...
}
```

**Square** does not provide a
**toString()** implementation.

TestToString.java

```
...
public class TestToString {

    public static void main(String[] args) {
        Rectangle r = new Rectangle(12, 5);
        System.out.println("Object with toString defined:");
        System.out.println(r);

        Square s = new Square(5);
        System.out.println("Object without toString defined:");
        System.out.println(s);
    }
}
```

Automatically calls **Rectangle**'s **toString()**.

## Try It:

Run *TestToString.java* to see the **toString()** method automatically called on **Rectangle** and the default
output for **Square**.

# Comparing and Identifying Objects

❋ With primitive datatypes, the == operator compares values.

❋ With reference variables, the == operator compares <u>references</u>; it evaluates to **true** if both references refer to the same object.

➢ Even if two objects have exactly the same state, == never returns **true** for variables referring to the two different objects.

❋ To perform value comparisons of objects, provide an **equals()** method for your class.

➢ This should be a **public boolean** method which, given a reference to another object of the same class, will compare its fields to the appropriate fields of the invoking object.

➢ Use the **this** keyword to refer to the current (invoking) object.

❋ You should also write a **hashCode()** method that returns the same **int** whenever two objects are **equal()** to each other.

➢ This is especially important if you plan to use your objects within data structures whose algorithms rely on an object's hash code in order to function properly.

❋ IDEs, like Eclipse, can generate both the **equals()** and the **hashCode()** implementations for you.

Rectangle.java

```
...
public class Rectangle {
    ...
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + height;
        result = prime * result + width;
        return result;
    }

    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Rectangle other = (Rectangle) obj;
        if (height != other.height)
            return false;
        if (width != other.width)
            return false;
        return true;
    }
}
```

> The **getClass()** method returns the class associated with this object.

Comparing.java

```
...
public class Comparing {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 5);
        Rectangle r2 = new Rectangle(10, 10);

        if (r1 == r2)
            System.out.println("Rectangle are ==");

        if (r1.equals(r2))
            System.out.println("Rectangle are 'equal'");
    }
}
```

## Try It:
Run *Comparing.java*.  Does **r1==r2**? Are they **equal**()?

# Parameter Passing

�  Recall that Java has primitive datatypes as well as reference datatypes.

�  When you pass a <u>primitive</u> to a method, it is passed <u>by value</u>.

  ➢  A copy of the primitive is sent into the method.

�  When you pass an <u>object</u> to a method, its <u>reference is passed by value</u>.

  ➢  A copy of the reference is sent into the method.

  ➢  If the method modifies its parameter, it will modify the original object.

Clock.java

```
...
public class Clock {
    private int hour;
    private int minute;
    private int second;
    private StringBuilder manufacturer;
    ...
    public void setManufacturer(StringBuilder m) {
        m.append(" Corporation");
        manufacturer = m;
    }
    ...
    public void setHour(int h) {
        if (h > 12)
            h = 24 - h;
        hour = h;
    }
    ...
}
```

setManufacturer() attempts to modify its **reference** parameter.

setHour() attempts to modify its **value** parameter, (**h**).

ParamPassing.java

```
public class ParamPassing {
  public static void main(String[] args) {
    Clock clock = new Clock(10, 18, 0);
    int theHour = 19;
    StringBuilder theManufacturer = new StringBuilder("Timex");
    clock.setHour(theHour);
    clock.setManufacturer(theManufacturer);

    System.out.println(clock);
    System.out.println(theHour);
    System.out.println(theManufacturer);
  }
}
```

## Try It:
Run *ParamPassing.java* to see the results of modifying a parameter.

# Destroying Objects

✱ You never explicitly destroy objects in Java.

➤ After the last reference to an object is lost, the Java Virtual Machine reallocates the object's memory at its convenience.

➤ This automatic Garbage Collection may happen immediately, when the program terminates, at some time in between, or never.

➤ You, the programmer, have no control over when, and in what order, unreferenced objects are garbage collected.

➤ Garbage Collections cannot be forced, but may be suggested by setting the last reference to an object equal to **null**.

```
Square s1 = new Square();
...
s1 = null;
```

✱ If an object has a **finalize()** method, the VM will invoke it just before the object is garbage collected.

```
protected void finalize() {
   System.out.println ("finalize called");
}
```

➤ Don't count on **finalize()** to free resources you need back soon, like open files or database connections.

✱ Many programmers follow the convention of creating a method called **dispose()**, which you then explicitly call when you are finished with an object.

Clock.java
```
...
protected void finalize() {
   System.out.println("finalize called");
}
...
```

You cannot force a particular object to be garbage collected, but you can explicitly invoke the garbage collector:

```
System.gc();
```

This will garbage collect all objects which are no longer referenced.

On most systems, you can log Garbage Collection events to a file, including the time in seconds since the first Garbage Collection event.

```
java -Xloggc:file JavaClass
```

TestGC.java
```
...
public class TestGC {
  public static void main(String[] args) {
    Clock[] clocks = new Clock[101];
    for (int i = 0; i < clocks.length; i++) {
      clocks[i] = new Clock();
      System.out.println("Clock created");
      clocks[i] = null;
      //System.gc();
    }

  }
}
```

## Try It:
Run *TestGC.java* and log the Garbage Collection events: **java -Xloggc:gc.out examples\TestGC**. Were any of the **Clock** objects garbage collected?


## Investigate:
Uncomment the **System.gc()** call after the **clocks[i] = null;**.

Run **TestGC** again, logging the Garbage Collection events. Were any of the **Clock** objects garbage collected this time?

# The Primitive-Type Wrapper Classes

❋ For each of Java's primitive types there is a *wrapper class*, which:

  ➢ Provides useful constants (e.g., **MAX_VALUE**).
  ➢ Allows generic container classes to store primitives.

❋ Each wrapper has a number of methods, including:

  ➢ A constructor with the primitive as an argument.
  ➢ A constructor with a **String** object as an argument.
  ➢ *type***Value()** methods producing the primitive type.
  ➢ An **equals()** method to test for equality.
  ➢ A **toString()** method.

❋ The **static parse***Type***()** method converts a **String** to a primitive type:

```
int x = Integer.parseInt("42");
```

❋ Convert a primitive into a **String** by either calling the **toString()** method of the primitive's wrapper class or by using the concatenation operator (+):

```
String s1 = Integer.toString(42);
String s2 = "" + 42;
```

❋ Use a feature called *autoboxing* to easily convert between a wrapper object and its primitive counterpart:

```
Double wrapper = 34.5;
double primitive = wrapper;
```

  ➢ Be aware that behind the scenes, the lines of code above are converted to something similar to the following:

```
Double wrapper = new Double(34.5);
double primitive = wrapper.doubleValue();
```

Abstract class **Number** (in *java.lang*) is the superclass of the numerical wrappers.

```
java.lang.Byte
java.lang.Short
java.lang.Double
java.lang.Float
java.lang.Integer
java.lang.Long
```

**Number** declares these abstract methods for converting values of wrapper objects to different primitive types:

```
byte byteValue()
short shortValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
```

Each numeric wrapper class includes the constants **MIN_VALUE** and **MAX_VALUE**. **Double** and **Float** also have **NaN**, **NEGATIVE_INFINITY**, and **POSITIVE_INFINITY**.

For **boolean** primitives, there is:

```
java.lang.Boolean
```

For the **char** primitives, there is:

```
java.lang.Character
```

In addition to the standard methods listed on the opposite page, **Character** includes lots of useful methods for testing and converting characters, and many character type constants.

```
char c = 'j';
if ( Character.isUpperCase(c) ) { ...
```

Once constructed, the value of a wrapper object cannot be changed.

If your program attempts to unbox a **null**, then you will get a **NullPointerException** at runtime.

```
Integer intWrapper = null;
int intPrimitve = intWrapper; // NullPointerException
```

# Enumerated Types

❋ Use the **enum** keyword to declare a typesafe enumeration of values.

```
public enum Color { RED, GREEN, BLUE }
```

➢ Declare the **enum** as a member of a class, or in its own source file.

❋ Access the **enum** constant using dot notation.

```
Color color = Color.RED;
```

❋ **enum**s can be used in **switch** statements.

```
switch (color) {
  case RED:
   ...
}
```

➢ Do not include the name of the enumeration within each **case**.

❋ The **enum** declaration generates a new class behind the scenes.

➢ Each **enum** constant is declared as **public**, **static**, and **final** field within the generated class.

▪ You cannot create new **enum** constants at runtime.

➢ There is no **public** constructor written for the class.

▪ You cannot create new **enum** constants at runtime.

➢ The generated class provides both an **equals()** and a **toString()** method.

▪ **Color.RED.toString()** yields **"RED"**.

OS.java

```
package examples;

public enum OS {
    ANDROID, IOS, WINDOWS, BLACKBERRY
}
```

HW.java

```
package examples;

public enum HW {
    APPLE, SAMSUNG, HTC, MOTOROLA, NOKIA, RIM,
}
```

TestEnum.java

```
package examples;

public class TestEnum {
    public static void main(String[] args) {
        CellPhone android = new CellPhone("Galaxy 4S", "303-555-5250",
                HW.SAMSUNG, OS.ANDROID);
        CellPhone iPhone = new CellPhone("iPhone 5", "303-555-5280",
                HW.APPLE, OS.IOS);

        System.out.println(android);
        System.out.println(iPhone);
    }
}
```

Note:

Since an **enum** is really a class, you are allowed to provide your own methods and fields in the **enum** declaration.

# Labs

For these labs, use either your solutions to a previous lab, or the *Employee5.java* and *EmpTest5.java* from the chapter directory.

❶ In **EmpTest**'s **main()**, pass the reference to your employee directly to **System.out.println()**. What does the object look like when printed as a **String**? Write a **toString()** method for your class and try it again.
(Solutions: *Employee6.java*, *EmpTest6.java*)

❷ Write an **equals()** method for your **Employee** class and test it. If you are generating code with an IDE, then feel free to generate the **hashCode()** method as well.
(Solutions: *Employee7.java*, *EmpTest7.java*)

❸ Print the salary of your employee using **System.out.printf()** so that there are two decimal places in the output.
(Solution: *EmpTest8.java*)

❹ (Optional) Use the Java API documentation to determine how to use **String.format()** to return the salary with two decimal places in the **toString()** you wrote previously.
(Solutions: *Employee8.java*, *EmpTest9.java*)

❺ Declare an enum called **Title** with values for the various job titles in your organization.
(Solution: *Title.java*)

❻ Modify your **Employee** class to include a field whose type is the enum you declared in ❺. Retrofit the existing code, including the constructors, to accomodate the new field. Modify your **EmpTest**'s **main()** method to include a title when creating a new instance of an **Employee** object.
(Solutions: *Employee9.java, EmpTest10.java*)
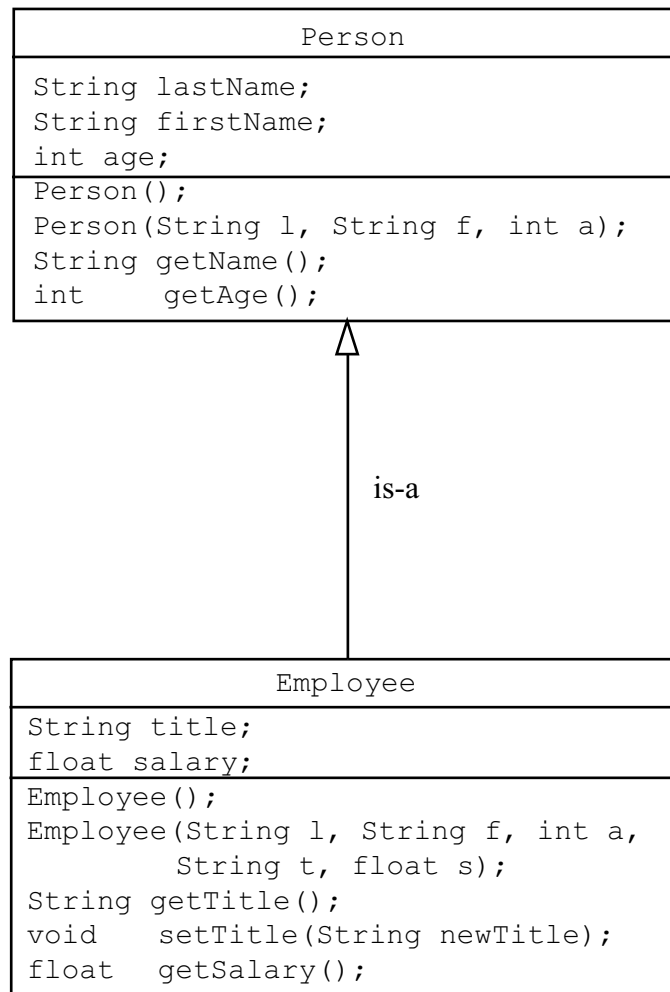
# CHAPTER 10 - INHERITANCE IN JAVA

# Inheritance

❋ *Inheritance* allows you to create a new class that contains all of the fields and methods of an existing class, and add to them.

➢ The class you inherit from is often called the *superclass*, and the inherited class is the *subclass*.

➢ The subclass may add new fields, but it always has all of the fields of the superclass.

➢ The subclass may add new methods, but it always has all of the methods of the superclass.

➢ Constructors are not inherited.

❋ Use inheritance when you want to avoid representing the same thing twice.

➢ Many subclasses can all inherit from a single superclass.

❋ When instantiating a subclass, only one object is built in memory.

➢ The new object contains memory allocations for all of the fields in the subclass, as well as those in the superclass.

➢ You will not have a separate object created for the subclass and the superclass.

Inheritance is a fundamental principle of Object-Oriented Programming. It allows developers to quickly expand the functionality of their programs, building on what was done before.

```
┌────────────────────────────────────────────────────┐
│                       Person                        │
├────────────────────────────────────────────────────┤
│ String lastName;                                    │
│ String firstName;                                   │
│ int age;                                            │
├────────────────────────────────────────────────────┤
│ Person();                                           │
│ Person(String l, String f, int a);                 │
│ String getName();                                   │
│ int     getAge();                                   │
└────────────────────────────────────────────────────┘
                          △
                          │
                          │
                        is-a
                          │
                          │
┌────────────────────────────────────────────────────┐
│                      Employee                       │
├────────────────────────────────────────────────────┤
│ String title;                                       │
│ float salary;                                       │
├────────────────────────────────────────────────────┤
│ Employee();                                         │
│ Employee(String l, String f, int a,                 │
│         String t, float s);                         │
│ String getTitle();                                  │
│ void    setTitle(String newTitle);                  │
│ float   getSalary();                                │
└────────────────────────────────────────────────────┘
```

A subclass is everything that the superclass is, plus some.

If the following statements are NOT true, then **Employee** should not **extend Person**.

An **Employee** is-a **Person**.
A **Person** has a **lastname**, **firstname**, and **age**.
An **Employee** has a **lastname**, **firstname**, **age**, **title**, and **salary**.

# Inheritance in Java

❋ The **extends** keyword indicates that a class inherits the fields and methods of another class.

```
public class Employee extends Person { ... }
```

➤ Equivalent terminology:

**Employee** inherits from **Person**.
**Employee** is a subclass of **Person**.
**Person** is the superclass of **Employee**.
**Employee** extends the **Person** class.
An **Employee** is a **Person**.

❋ A Java class can inherit from only one superclass.

❋ If you don't explicitly extend some superclass when you create a class, the new class will inherit from **java.lang.Object**.

```
public class Person { ... }
```

Is the same as:

```
public class Person extends Object { ... }
```

❋ A class declared as **final** can't be extended.

```
public final class String { ... }
```

➤ For example, you can't create subclasses of **java.lang.String**.

➤ Declaring a class as **final** also allows the compiler to perform some optimizations.

Person.java

```
...
public class Person {
    protected String lastName;
    protected String firstName;
    protected int age;

    public Person() {
        this("", "", 0);
    }
    public Person(String l, String f, int a) {
        lastName = l;
        firstName = f;
        age = a;
    }

    public String getName() {
        return firstName + " " + lastName;
    }
    public int getAge() {
        return age;
    }
}
```

**Person**
implicitly extends
**Object**.

Protected fields
are available to
subclasses.

Employee.java

```
...
public class Employee extends Person {
    private String title;
    private float salary;

    public Employee() {
        this("", "", 0, "Clerk", 20000);
    }
    public Employee(String l, String f, int a, String t, float s) {
        lastName = l;
        firstName = f;
        age = a;
        title = t;
        salary = s;
    }

    public String getTitle() {
        return title;
    }
    public void setTitle(String newTitle) {
        title = newTitle;
    }
    public float getSalary() {
        return salary;
    }
}
```

An **Employee**
is-a **Person**.

# Casting

❋ A variable can refer to an object of its own type, or of a subclass type.

```
Person p = new Employee();
```

➢ The **Employee** object returned by the **new** operator is implicitly cast to a **Person** before the assignment takes place.

➢ This is also called *upcasting*, because we are casting a subclass object *up* to its superclass datatype.

➢ This cast works because any field or method that you try to access through the superclass variable is guaranteed to exist in the subclass.

```
System.out.println(p.getName());
```

❋ *Downcasting*, explicitly casting a superclass to be a subclass, will only work if the superclass variable references a subclass object.

```
Person p = new Employee();
Employee e = (Employee) p;   // this works!
```

➢ If **p** does not refer to an **Employee** (or one of its subclasses) a **ClassCastException** will be thrown.

```
Person p = new Person();
Employee e = (Employee) p; //ClassCastException!
```

➢ Use the **instanceof** operator to avoid this exception.

```
if (p instanceof Employee) {
   Employee e = (Employee) p;
}
```

HumanResources.java

```
package examples;

public class HumanResources {

    public HumanResources() {
        Employee boss = new Employee("Lee", "Jan", 71, "Manager",
                900000);
        Employee lead = new Employee("Dobbs", "Bob", 54, "Lead", 75000);

        if (isPastRetirement(boss)) {
            System.out.println(boss.getName() + " can retire.");
        }

        if (isPastRetirement(lead)) {
            System.out.println(lead.getName() + " can retire.");
        }
    }

    public boolean isPastRetirement(Person p) {
        // will the following line compile?
        // float salary = p.getSalary();

        if (p.getAge() > 65) {
            return true;
        }
        else {
            return false;
        }
    }

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

The most common reason to cast a subclass to its superclass is for parameter passing. The
**isPastRetirement()** method can be called with either a **Person** object or an **Employee** object.

This works because **Employee** inherits the **getAge()** method from the **Person** class.


Try It:
Run *HumanResources.java*.

---

# Method Overriding

❈ A subclass can define a method that overrides a method in the superclass.

> ➤ The subclass can define new behavior for this method.

❈ The subclass method must have exactly the same signature as the superclass method.

> ➤ To instruct the compiler to validate that the signatures match, use the **@Override** annotation.

```
@Override
public String getName() {
```

> ➤ The method cannot be less accessible than the method it overrides.

>> ▪ A **public** method cannot be overridden with a **protected** or **private** method.

❈ A **final** method cannot be overridden by a subclass.

Person.java

```
...
public class Person {
    protected String lastName;
    protected String firstName;
    protected int age;
    ...
    public String getName() {
        return firstName + " " + lastName;
    }

    public int getAge() {
        return age;
    }
}
```

Employee1.java

```
...
public class Employee1 extends Person {
    private String title;
    private float salary;

    ...
    public String getTitle() {
        return title;
    }

    public void setTitle(String newTitle) {
        title = newTitle;
    }

    public float getSalary() {
        return salary;
    }

    @Override
    public String getName() {
        if (title.equals("Doctor"))
            return "Dr. " + firstName + " " + lastName;
        return firstName + " " + lastName;
    }
}
```

# Polymorphism

❋ By default, the Virtual Machine uses dynamic lookup to determine which method to invoke.

➢ The method that is called depends on the datatype of the target object.

➢ A method called on an **Employee** object will call the **Employee** method, even if it's called through a **Person** reference.

```
Person p = new Employee();
p.getName();   // calls Employee's method
```

➢ This is referred to as *dynamic binding*, or *polymorphism*.

❋ Polymorphism allows you to create objects of different (but related) types and manage them all in the same way, using the same code.

➢ You might store the objects in a single collection, such as an array.

➢ When you send a message to an object, it will respond in a way that is appropriate for its class.

➢ The code that manages the objects and sends messages to them does not need to know the datatypes of the individual objects.

❋ **final** methods, and methods in **final** classes, are statically bound, and the compiler may inline the code if it is short enough.

▪ Since subclasses cannot override these methods, the compiler, instead of the Virtual Machine, can bind the method to its invocation.

Tester.java
```
package examples;

public class Tester {
   public static void main(String args[]) {
      Person mom = new Person("Doe", "Jane", 71);
      Employee1 doc = new Employee1("Smith", "John", 45, "Doctor",
            275000);
      Person p;

      p = mom;
      System.out.println(p.getName()); // which method?

      p = doc;
      System.out.println(p.getName()); // which method?
   }
}
```

This is *polymorphism*; the ability to call the same method but get different behavior based on the object's class type.


**Try It:**
Run *Tester.java* to see polymorphism in action.

## SUPER

✱ In a subclass method, you can refer to the superclass using the keyword **super**.

✱ Use **super.*var*** to access a *shadowed* field in a parent class.

➢ If the **Person** class contained a field named **height** and **Employee** contained a field named **height**, **Employee**'s **height** would shadow **Person**'s  **height**.

✱ Use **super.*method*()** to call overridden methods in the superclass.

➢ This is typically done in methods that the subclass overrides; the subclass can add code before or after the call to the superclass' method.

➢ Following good OO practices like encapsulation, the subclass should only access the superclass' data through the methods the superclass provides.

✱ Use **super(*arg1, ...*)** to invoke a superclass constructor.

➢ This call must be the first line of code in the subclass' constructor.

➢ If you don't call a superclass constructor explicitly, then the compiler will insert a call to the superclass' no-arg constructor in every constructor in the subclass.

Person2.java
```
...
public class Person2 {
    private String lastName;
    private String firstName;
    private int age;

    public Person2() {
        this("", "", 0);
    }

    public Person2(String l, String f, int a) {
        lastName = l;
        firstName = f;
        age = a;
    }

    public String getName() {
        return firstName + " " + lastName;
    }
    ...
}
```

> **private** fields are not accessible by subclasses.

Employee2.java
```
...
public class Employee2 extends Person2 {
    private String title;
    private float salary;

    public Employee2() {
        this("", "", 0, "Clerk", 20000);
    }

    public Employee2(String l, String f, int a, String t, float s) {
        super(l, f, a);
        title = t;
        salary = s;
    }
    ...
    public String getName() {
        if (title.equals("Doctor"))
            return "Dr. " + super.getName();
        return super.getName();
    }
}
```

# The Object Class

✳ Ultimately, every class in Java descends from **java.lang.Object**.

> ➢ A variable of type **Object** can, therefore, refer to anything!

✳ The **Object** class contains several methods that are inherited by, and can be overridden by, its children, including:

**public boolean equals(Object obj)** — value comparison.
**public String toString()** — convert to a String.
**public final Class getClass()** — retrieve Class object.
**public int hashCode()** — retrieve hashcode for a Hashtable.

> ➢ Overriding **equals()** and **toString()** is very common and allows consistent processing of all object types.

✳ An **Object** variable can be used when the actual datatype of the object is not known (similar to C **void** pointers).

```
public void someMethod(Object o) { ... }
```

> ➢ In this example *someMethod*() could be passed an object from any class.

> ➢ If you need to use methods other than the ones found in the **Object** class, you can downcast the reference to the necessary type.

✳ This allows for generic containers (similar to C++ templates).

> ➢ **java.util.ArrayList** is a growable array of **java.lang.Object** references, which can refer to any object.

All Java classes are eventually extended from **java.lang.Object**. This includes the classes that you create, such as **Person** and **Employee**.

```
                          ┌─────────────┐
                          │   Object    │
                          └─────────────┘
                   is-a        is-a        is-a

     ┌─────────────┐    ┌─────────────┐    ┌─────────────┐
     │   Person    │    │   String    │    │   Number    │
     └─────────────┘    └─────────────┘    └─────────────┘
            is-a                          is-a        is-a

     ┌─────────────┐              ┌─────────────┐  ┌─────────────┐
     │  Employee   │              │   Integer   │  │    Float    │
     └─────────────┘              └─────────────┘  └─────────────┘
```
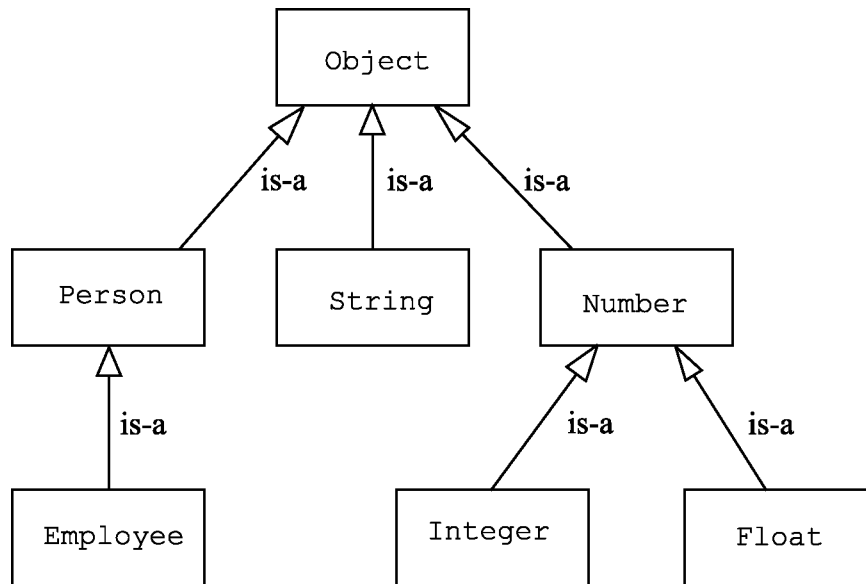
# Labs

❶ Create a class called **Vehicle** that contains fields for the vehicle's maximum speed and number of wheels. Provide both a no-argument constructor and a two-argument constructor. Provide accessor (**get**) and mutator (**set**) methods for the fields.
(Solution: *Vehicle.java*)

❷ Create a subclass of **Vehicle** called **Bicycle** that contains a field for the number of gears on the bike. All bikes should have two wheels.
(Solution: *Bicycle.java*)

❸ Add a **toString()** method to both the **Vehicle** class and the **Bicycle** class. Have **Bicycle**'s **toString()** use **Vehicle**'s **toString()**. Write an application that creates a **Vehicle** with three wheels and a top speed of 45 mph (like an ATV) and a **Bicycle** with ten gears and a top speed of 30 mph. Display both objects.
(Solutions: *Vehicle1.java*, *Bicycle1.java*, *VehicleTest.java*)

❹ Declare an array of three **Vehicles**. Make the first two elements **Bicycle** objects: a 10-speed and a 3-speed. Make the third element just a **Vehicle** object. Write a loop that runs through the array, printing each object. Which **toString()** is called each time?
(Solution: *VehicleArray.java*)

❺ What happens if you change the array's type to **Object**?
(Solution: *ObjectArray.java*)

❻ What happens if you change the array's type to **Bicycle**?
(Solution: *BicycleArray.java*)

# Abstract Classes

✱ An **abstract** method is a method without an implementation.

➢ A class with an **abstract** method must also be declared **abstract**.

✱ An **abstract** class cannot be instantiated directly.

✱ The purpose of an **abstract** class is to be extended.

➢ The subclass implements the **abstract** methods declared in its superclass.

➢ If a subclass does <u>not</u> provide an implementation of an **abstract** method, it will not compile unless it is also declared **abstract**.

▪ You can still use the **@Override** annotation with **abstract** methods, but it isn't necessary.

✱ Abstract classes can contain normal (concrete) methods, as well as the **abstract** methods.

➢ These concrete methods are inherited by the subclasses and can be overridden if necessary.

✱ A class can be declared **abstract** even if it does not contain any **abstract** methods.

➢ This will force the class to be subclassed by its user.

Light.java

```
...
public abstract class Light ... {
   private Status status;

   public void turnOn() {                    Concrete method.
      status = Status.ON;
   }
   ...                                        Abstract method.
   public abstract void changeBulb();
   ...
}
```

FluorescentLamp.java

```
...
public class FluorescentLamp extends Light {
    @Override
    public void changeBulb() {
        System.out.println("Change tube in fluorescent lamp.");
        System.out.println("Dispose of old tube properly.");
    }
}
```

HalogenLight.java

```
...
public class HalogenLight extends Light {
    @Override
    public void changeBulb() {
        System.out.println("Change bulb in halogen light");
        System.out.println("Don't touch the bulb with your bare hands");
    }
}
```
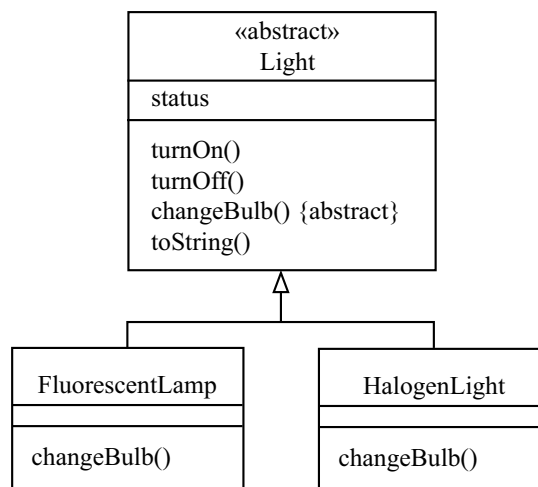
# Interfaces

✴ An **interface** specifies behavior by declaring methods.

 ➤ Interfaces are not allowed to contain implementation code.

 ➤ All methods in an interface are implicitly **public**.

 ➤ All methods in an interface are implicitly **abstract**.

 ➤ An **interface** may also contain fields (which are implicitly **static final**).

✴ Any class that implements an **interface** must support that behavior by writing the methods.

✴ Class inheritance fulfills an *is-a* relationship; interface implementation can be defined as an *is* relationship.

 ➤ A **HalogenLight** *is-a* **Light**; a **HalogenLight** *is* **Switchable**.

 ➤ A **Fan** *is* **Switchable**.

✴ A class may extend only one class, but may implement many interfaces.

✴ An interface can extend (inherit from) another interface.

 ➤ A class that implements the extended interface must implement all inherited methods.

Switchable.java

```
public interface Switchable {
    public void turnOn();
    public void turnOff();
}
```

These **abstract** methods must be provided by the implementing classes.

GasFireplace.java

```
...
public class GasFireplace implements Switchable {
    ...
    public void turnOn() {
        status = Status.ON;
        // turn on the gas, ignite the flame
    }
    public void turnOff() {
        status = Status.OFF;
        // turn off the gas
    }
...
```

Fan.java

```
...
public class Fan implements Switchable {
    ...
    public void turnOn() {
        status = Status.ON;
        // begin fan rotation
    }
    public void turnOff() {
        status = Status.OFF;
        // stop fan rotation
    }
...
```

Light.java

```
...
public abstract class Light implements Switchable {
    ...
    public void turnOn() {
        status = Status.ON;
    }
    public void turnOff() {
        status = Status.OFF;
    }
...
```

# Using Interfaces

❋ Interfaces are another way to support polymorphism.

➢ A variable declared of an interface type can refer to an object of any class that implements that interface.

```
Switchable s = new Fan();
```
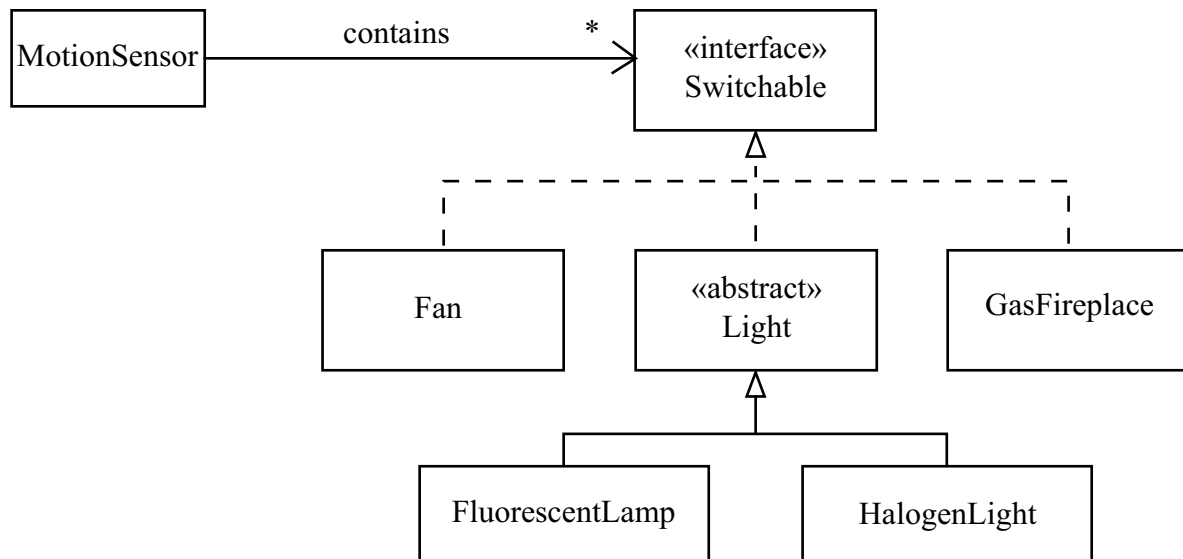
MotionSensor.java

```
...
public class MotionSensor {
    private Switchable[] items;
    private int count = 0;

    public MotionSensor(int numItems) {
        items = new Switchable[numItems];
    }
    public void add(Switchable s) {
        if (count < items.length)
            items[count++] = s;
        // else throw exception
    }
    public void motionDetected() {
        System.out.println("Motion Detected");
        for (Switchable item : items) {
            if (item != null)
                item.turnOn();
        }
    }
    public void timeout() {
        // Need to add code to determine if enough time has elapsed
        System.out.println("Timeout occurred");
        for (Switchable item : items) {
            if (item != null)
                item.turnOff();
        }
    }
    ...
    public static void main(String[] args) {
        MotionSensor sensor = new MotionSensor(4);
        sensor.add(new FluorescentLamp());
        sensor.add(new HalogenLight());
        sensor.add(new GasFireplace());
        sensor.add(new Fan());
        // sensor.add(new Light());

        sensor.motionDetected();
        System.out.println(sensor);
        sensor.timeout();
        System.out.println(sensor);
    }
}
```

Polymorphism ensures that the **turnOn()** method is called on the appropriate object.

Implicit upcast.

Why won't this compile?

## Try It:

Run *MotionSensor.java* to see polymorphic behavior applied to interfaces.

# Labs

❶      Create an **enum** named **Color** with values for your five favorite colors.
(Solution: *Color.java*)

❷      Create an abstract class named **Shape**. Provide a field for color. Add **get**s and **set**s as well as appropriate constructors. Add an abstract method called **getArea()**.
(Solution: *Shape1.java*)

❸      Develop two child classes of **Shape**: **Rectangle** and **Circle**. A **Rectangle** has a width and height, a **Circle** has a radius. Provide **get**s and **set**s for each field and appropriate constructors. Implement the **getArea()** method in each subclass. (Hint: Find **java.lang.Math** in the Java API documentation for usage information on **Math.PI** and the **Math.pow()** method for calculating $\pi r^2$.)
(Solutions: *Rectangle1.java, Circle1.java*)

❹      Write a tester program that contains a **main()** method that creates an array of three **Shape**s. Store two **Rectangle**s and a **Circle** in the array. Loop through the array, printing out the area of each shape.
(Solution: *ShapeTester.java*)

❺      Create an interface named **Drawable** with the **void** method **draw()** declared inside of it.
(Solution: *Drawable.java*)

❻      Retrofit **Shape** to implement **Drawable**, putting the actual implementation code in **Rectangle** and **Circle**. Don't worry about doing any graphics for drawing, just print out a simple message indicating the type and color of each shape you are "drawing."
(Solutions: *Shape2.java, Circle2.java, Rectangle2.java*)

❼      Modify your tester program to store an array of **Drawable**, instead of **Shape**. Call the **draw()** method on each, instead of the **getArea()** method.
(Solution: *DrawableTester.java*)

❽      (Optional) Create a class named **Text** that implements **Drawable** and has a **String** field called **value**. Add **get**s and **set**s, as well as an appropriate constructor. Have **Text**'s **draw()** method print the **value** field. Modify your tester program so that a **Text** object is added into the array instead of one of the **Rectangle**s.
(Solutions: *Text.java, DrawableTester2.java*)

# Chapter 12 - Packages

### Objectives

❋ Write programs that use classes from other packages.

❋ Write programs that import packages.

❋ Create and use your own packages.

# Packages

❋ A *package* is a convenient way of grouping classes that have related functionality, but may not be in the same file.

❋ Each *.java* file may specify that it is part of a package.

```
package examples;
```

➢ Package names will correspond to directory names.

❋ The Java API groups classes into packages with related functionality:

➢ **java.lang** — basic language classes.

➢ **java.sql** — database access classes.

➢ **java.util**— general utility classes.

➢ **java.util.regex**— regular expression utility classes.

➢ . . . and many others.

❋ Since classes may be loaded across the Internet, packages reduce possibilities of namespace collisions.

➢ Two packages can both contain a **List** class.

When you use a class that belongs to another package, precede the class name with the package name:

```
java.awt.Frame
```

**java.awt** is a package name.

**Frame** is a class name.

To make the code easier to read, Java developers have used the convention of capitalizing the first letter in class names. So the '**F**' in Frame tells us that Frame is the name of a class. Any name before a class name is a package or subpackage name.

# The import Statement

❋ To use a class from another package, just qualify the class name with the package name:

```
java.util.ArrayList a = new java.util.ArrayList();
```

❋ The **import** statement allows you to use a package's classes without fully specifying the package name each time.

```
import packagename.*;
```

➢ The package's classes become part of the current program's namespace.

❋ You may import a single class or an entire package.

```
import java.util.ArrayList;
```

➢ Use the wild card character **\*** to import an entire package.

```
import java.util.*;
```

▪ There is no memory or code size penalty for using the wild card.

❋ References to classes from an imported package do not need to be preceded by the package name.

➢ For example, **java.util.ArrayList** can be specified as simply **ArrayList**.

❋ If two imported packages define a class with the same name, you must fully specify the name of the class that you are trying to use.

Months1.java

```
package examples;

public class Months1 {
  public static void main(String[] args) {
    java.util.ArrayList<String> months = new java.util.ArrayList<>();
    months.add("January");
    months.add("February");
    months.add("March");
    ...
    for (String month : months) {
      System.out.println(month);
    }
  }
}
```

The same code using **import**:

Months2.java

```
package examples;

import java.util.ArrayList;

public class Months2 {
  public static void main(String[] args) {
    ArrayList<String> months = new ArrayList<>();
    months.add("January");
    months.add("February");
    months.add("March");
    ...
    for (String month : months) {
      System.out.println(month);
    }
  }
}
```

We can now use the **ArrayList** class without specifying the package.

One particular package that contains classes used in almost every program is **java.lang**. **java.lang** is automatically imported. Explicitly importing it won't hurt (or help).

The **System** class that we use for printing belongs to this package. We did not have to say: **java.lang.System.out.println("Hello, World");**.

## Static Imports

❉ For convenience, you can use static imports to save you typing.

➢ Any reference to a **static** field or method can be left unqualified.

```
import static java.lang.System.out;
...
   out.println("Hello World");
```

➢ You can also use the wildcard with **static** imports.

```
import static java.lang.Math.*;
...
   double area = PI * pow(radius, 2);
```

StaticImports.java
```
package examples;

import static java.lang.System.out;
import static java.lang.Math.*;

public class StaticImports {
    public static void main(String[] args) {
        int radius = 5;
        double area = PI * pow(radius, 2);

        out.printf(
                "The area of a circle with radius %1$d is %2$.2f %n",
                radius, area);
    }
}
```

# CLASSPATH and Import

❋ At runtime, the *Application Class Loader* searches for *.class* bytecode files in the *application classpath*.

➢ By default, the application classpath is the current directory.

❋ You can override the default by setting the environment variable **CLASSPATH**.

➢ Any of the directories in the **CLASSPATH** may have subdirectories; these are the package directories.

➢ **java** (as well as other JDK utilities) has a flag, **-classpath**, that overrides the **CLASSPATH** variable.

❋ **CLASSPATH**, or **-classpath**, if set, must include the directories containing any classes you create.

➢ The current directory, represented by "**.**" on most systems, is usually placed first.

➢ Any project directories that contain package subdirectories must be specified.

❋ For Java to load classes from these packages/directories, the file permissions must grant access.

Any time you use classes from your own packages, set your **CLASSPATH** properly.

Suppose that you have created a **university** package and are working on it under the directory */home/ projects/*:

```
/home/projects/university/Student.class
/home/projects/university/Professor.class
/home/projects/university/...
```

When compiling or running applications that use classes from your **university** package, make sure your **CLASSPATH** includes the directory that contains your package directory:

```
CLASSPATH=.:/home/projects
```

Do not include the *university* directory itself in your **CLASSPATH**!

The **java** interpreter actually searches for classes (bytecode files) in three directory paths:

The *bootstrap classpath* contains the directory for the standard Java bootstrap classes. This is typically */java-dir/lib*. These are the classes contained in *rt.jar*. This path can be changed by using the -**Xbootstrap** option with the various tools, but this is not recommended.

The *extension directories* contain packages that can be used in conjunction with the standard Java bootstrap classes, as though they were built-in. This is primarily used for third-party packages, or applets using additional packages. The extension directory is */java-dir/lib/ext*.

The *application classpath* contains directories for additional packages that you have created. By default, the current directory (".") is searched. This path can be changed by setting the **CLASSPATH** environment variable, or by using the **-classpath** option with the various tools. When you provide an application classpath, either with the environment variable or the command line option, "." is not automatically added.

# Defining Packages

❋ The **package** keyword allows you to define a package.

➢ The **package** statement must be the first non-comment line.

❋ The package can be a single name:

```
package university;
```

❋ The package can be a specialized dot-separated name:

```
package project.src;
```

❋ Package names correspond to directories.

➢ If your package name is **project.src**, then its class files must be in a directory named *project/src/*.

❋ It's a good idea to pick package names that won't conflict with other groups in your company; this will allow you to share classes with one another.

```
network.services
```

➢ You can further divide the package name:

```
network.services.accounting
```

❋ If you use your Internet domain name as part of your package name, you're pretty much guaranteed a unique namespace:

```
package com.example.guistuff;
```

➢ In fact, Oracle recommends, and uses, this practice:

```
package com.oracle.jdbc.driver;
```

Car.java
```
package examples.rentalcar;

public class Car {
    private String vIN;
    private String tag;

    public String getVIN() {
        return vIN;
    }

    public void setVIN(String vin) {
        vIN = vin;
    }

    public String getLicenseTag() {
        return tag;
    }

    public void setLicenseTag(String t) {
        tag = t;
    }
}
```

The compiled class for *Car.java* should be located under the *examples/rentalcar* directory.

By convention, Java programmers use lowercase letters for their package names.

If you omit the package declaration, then your code is said to be a member of the *default package*.  Its use is discouraged since you open yourself up to namespace collisions when you don't use packages.

# Package Scope

❋ When you define member data or methods within a class, you may choose to leave off the access specifiers **private**, **protected**, and **public**.

➢ The field or method is then said to have default access.

▪ Default access is also known as *package access*.

❋ *Default access control* means that any other class within the same package will have full access to that field or method.

The **public** modifier on a class makes it accessible outside its package.

Classes with default, or package, access are helper classes that can only be used from classes within the same package.

# LABS

❶ Create a package called **animal** with two **public** classes, **Dog** and **Cat**. Create a Java application, within another package, that creates both a **Dog** and a **Cat** object. Test without using **import**, then again with the **import** statement.
(Solutions: *animal/Dog.java*, *animal/Cat.java*, *AnimalTest.java*)

❷ Add a non-**public** class called **Jackal** to your **animal** package. Try creating a **Jackal** object in your application.
(Solutions: *animal/Jackal.java*, *AnimalTest2.java*)

❸ Create a second package named **zoo** that contains a **Cat** class (big zoo-type cats!). How can you use objects of both **Cat** classes in your application?
(Solutions: *zoo/Cat.java*, *ZooTest.java)*

# Chapter 13 - Exception Handling



## Objectives

❋    Use methods that throw exceptions.

❋    Handle exceptions thrown by methods.

❋    Defer exception handling.

❋    Raise exceptions in your own methods.

❋    Define your own exception classes.

# Exceptions Overview

❊    Upon encountering a condition in which it cannot continue, a method will throw an exception.

    ➢    An *exception* is an object whose type, data, and methods describe the problem that caused it to be thrown.

❊    Java supports two types of exceptions: checked and unchecked.

    ➢    The compiler guarantees that all checked exceptions are handled in code.

        ▪    This is not the case for unchecked exceptions.

❊    When a method calls another method that can throw a checked exception, the calling method must either:

    1.    Catch the exception the called method throws.

    2.    Declare that it, the caller, might itself throw the same type of exception.

❊    An exception propagates up the call chain until it is caught.

    ➢    If a method does not catch an exception, the method's caller will be thrown the same exception, and so on.

❊    If an unchecked exception is not caught, the default handler in the VM prints a stack trace and then exits.

# CATCHING EXCEPTIONS

❋ To handle a potential exception, enclose the code which may throw the exception in a **try** block.

❋ A **catch** block contains code which handles the specific exception type thrown.

```
catch(ExceptionClass e) { ... }
```

➢ The **catch** block must immediately follow the **try** block.

❋ There may be multiple **catch** blocks following a **try** block, to handle different types of exceptions.

➢ A **catch** catches all exceptions of the given class, or any of its subclasses.

▪ Subclass exceptions need to be caught before their superclass.

➢ As of Java 7, you can combine multiple catch blocks into one using a new syntax called multi-catch.

▪ Simply use the pipe symbol (|) to separate the exceptions you intend to catch.

```
catch(ParseException | ArithmeticException e) {
   System.err.println(e.getMessage());
}
```

❋ Once an exception is thrown, execution flow immediately transfers out of the **try** block to the first **catch** block matching the class, or a superclass, of the exception.

➢ Any statements remaining in the **try** block are not executed.

❋ If the **catch** block does not **return**, **exit**, or **throw** its own exception, execution will continue below the last **catch** block.

ParseTest.java

```
package examples;

import java.text.NumberFormat;
import java.text.ParseException;

public class ParseTest {
    public static void main(String[] args) {
         NumberFormat format = NumberFormat.getCurrencyInstance();
        String s;
        Number num;
        s = "$45.67";
        // s = "hi mom";

        try {
            num = format.parse(s); // may generate exception
            System.out.println("Float value = " + num.floatValue());
        }
        catch (ParseException e) {
            System.err.println("Invalid string \"" + s + "\"");
        }
    }
}
```

## Try It:

Run *ParseTest.java*. The float value should be displayed. Now comment out the line **s = "$45.67";** and uncomment the line **//s= "hi mom";**. Recompile and run the program. What happens?

# The finally Block

❋ A **finally** block contains code that will always execute after the preceding **try** block, <u>whether or not any exceptions are thrown</u>.

  ➢ Place the **finally** block below all **catch** blocks, if there are any.

❋ Any **return**, **continue**, **break**, or **throw** inside the **try** block will still cause the **finally** block to execute.

ParseFinally.java

```java
package examples;

import java.text.NumberFormat;
import java.text.ParseException;

public class ParseFinally {
    public static void main(String[] args) {
         NumberFormat format = NumberFormat.getCurrencyInstance();
        String s;
        Number num;
        s = "$45.67";
        // s = "hi mom";

        try {
            num = format.parse(s); // may generate exception
            System.out.println("Float value = " + num.floatValue());
        }
        catch (ParseException e) {
            System.err.println("Invalid string \"" + s + "\"");
        }
        finally {
            System.out.println("Original string was \"" + s + "\"");
        }
    }
}
```

### Try It:
Run *ParseFinally.java* as you did with *ParseTest.java*. Is the float value always displayed? How about the original string?

### Note:
A **finally** block, with a **try** block, is sometimes used even when no exception handling is needed; it is then simply code that will always be run.

# Exception Methods

❋ All exception classes extend **java.lang.Exception**, which extends **java.lang.Throwable**.

❋ **Throwable** methods provide diagnostic information.

➢ **getMessage()** returns a descriptive message.

➢ **printStackTrace()** shows where the exception occurred.

➢ **toString()** displays the exception type.

➢ Sometimes a **catch** block will throw another exception.

▪ To find information about the first exception that was thrown, use the **getCause()** method.

❋ Only **Throwable** objects can be thrown.

➢ **Error** objects are not supposed to be caught and handled; they are considered fatal.

➢ **RuntimeException**s are unchecked, they do not have to be declared or caught to compile, but will still terminate program execution when encountered.

➢ For practical purposes, you only need to deal with **Exception**, and subclasses of **Exception**, in your code.

```
public   class   java.lang.Throwable extends   java.lang.Object {
   // Constructors
   public  Throwable();
   public  Throwable(String   message);

   // Methods
   public Throwable fillInStackTrace();
   public Throwable getCause();
   public String getLocalizedMessage();
   public String getMessage();
   public StackTraceElement[] getStackTrace();
   public Throwable initCause(Throwable t);
   public void printStackTrace();
   public void printStackTrace(PrintStream  s);
   public void printStackTrace(PrintWriter  s);
   public void setStackTrace(StackTraceElement[] stackTrace);
   public String toString();
}
```

```
public   class   java.lang.Exception extends java.lang.Throwable {
   // Constructors
   public Exception();
   public Exception(String  s);
}
```
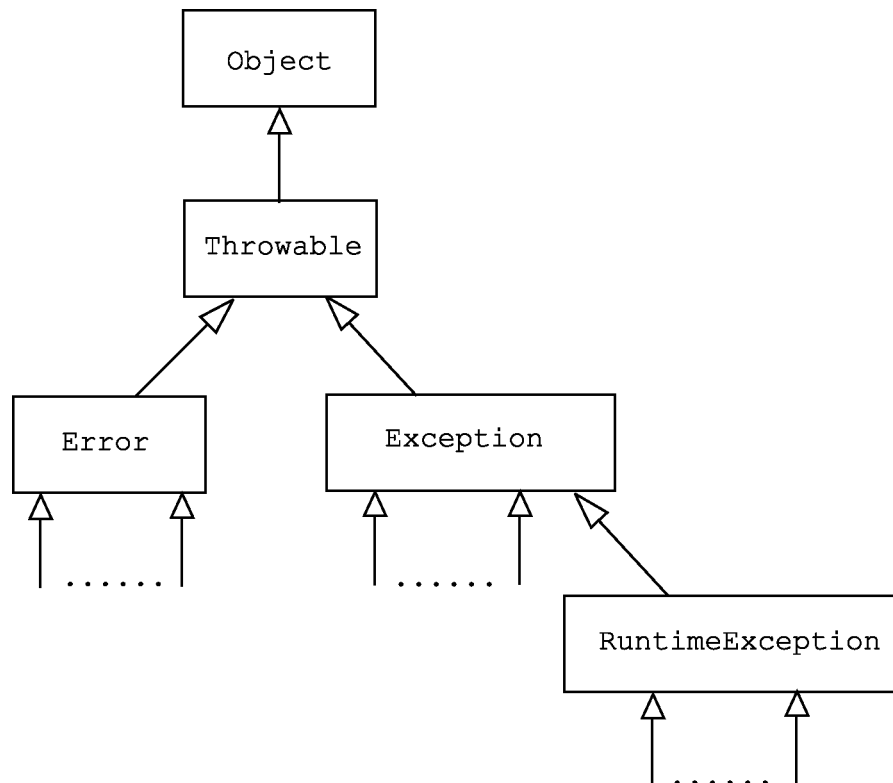
# Declaring Exceptions

❋ A method with code that may generate an exception must either:

 ➢ Encapsulate the code in a **try** block followed by appropriate **catch** blocks,

 OR

 ➢ Declare that it may throw the exception.

```
public float parseIt(String s)
   throws ParseException
```

❋ Users of your method will understand what exceptions they need to either **catch** or declare.

 ➢ The exceptions that a method may throw are part of its **public** interface.

❋ Code that can generate uncaught, undeclared, checked exceptions will not compile!

❋ To find out what exceptions a standard Java method might throw, look for the **throws** clause in the documentation.

ParseDeclare.java

```java
package examples;

import java.text.NumberFormat;
import java.text.ParseException;

public class ParseDeclare {

    public float parseIt(String s) throws ParseException {
        NumberFormat format = NumberFormat.getCurrencyInstance();
        Number num = format.parse(s); // may generate exception

        return num.floatValue();
    }

    public static void main(String[] args) {
        ParseDeclare parser = new ParseDeclare();
        String s;
        s = "$45.67";
        // s = "hi mom";

        try {
            System.out.println("Float value = " + parser.parseIt(s));
        }
        catch (ParseException e) {
            System.err.println("Invalid string \"" + s + "\"");
        }
        finally {
            System.out.println("Original string was \"" + s + "\"");
        }
    }
}
```

Try It:

Run *ParseDeclare.java* with the **"hi mom" String** to see the exception.

# Defining and Throwing Exceptions

❊ To explicitly throw an exception:

1. Instantiate an exception object.

2. Invoke the **throw** statement using the exception object.

   ```
   throw new InvalidDataException();
   ```

❊ If no existing exception class is appropriate, define a new one.

```
class InvalidDataException extends Exception {...
```

➢ User-defined exception classes should extend **Exception**.

➢ You can add new methods and data which can be used in a catch handler.

▪ Often, just knowing the type of the exception that was thrown is all the information you need.

➢ You can write **catch** blocks to handle your new exception type.

```
catch(InvalidDataException e) {
   System.err.println(e.getMessage());
}
```

InvalidDataException.java

```
...
public class InvalidDataException extends Exception {
    public String getMessage() {
        return "Name must be provided.";
    }
}
```

Person.java

```
...
public class Person {
    private String name;

    public Person(String n) throws InvalidDataException {
        if (n == null || n.equals("")) {
            throw new InvalidDataException();
        }
        name = n;
    }
    ...
}
```

UsePerson.java

```
...
public class UsePerson {
    public static void main(String[] args) {
        Person p = null;
        String name = null;
        name = "Some Name"; // comment this line to see an exception

        try {
            p = new Person(name);
        }
        catch (InvalidDataException e) {
            System.err.println(e.getMessage());
        }

        System.out.println(p);
    }
}
```

## Try It:

Comment out the line which initalizes the **name** variable in *UsePerson.java*, then run it to see the
**InvalidDataException** thrown.

# Errors and RuntimeExceptions

❋ **Error** extends **Throwable** and is a peer of **Exception**.

❋ **Error**s are severe malfunctions that are not intended to be caught.

➢ Running out of memory or missing libraries are situations that could cause **Error**s to be thrown.

➢ The Java VM generates **Error**s; you should not create or throw them yourself.

➢ Handling of these problems is usually very difficult and is not expected; the program should terminate.

❋ **RuntimeException** is a child of **Exception** and has several subclasses.

➢ These exceptions should not occur in properly written code, therefore handling of them is not required by the compiler.

➢ For example, if you always check the length of an array, you will never get an **ArrayIndexOutOfBoundsException**.

```
for (int i = 0; i < people.length; i++)
   people[i].display();
```

❋ **Error**s and **RuntimeException**s do not have to be declared as part of a method signature; they are unchecked.

PeopleTest.java

```
package examples;

public class PeopleTest {
    public static void main(String[] args) {
        Person[] people;

        people = new Person[3];
        try {
            people[0] = new Person("Bob");
            people[1] = new Person("Jane");
            people[2] = new Person("John");
        }
        catch (InvalidDataException e) {
            System.err.println(e.getMessage());
        }

/*
        // produces ArrayIndexOutOfBoundsException, a RuntimeException
        for (int i = 0; i < 4; i++)
            System.out.println("people[" + i + "] contains "
                    + people[i]);
*/

        // Use the array length to avoid the RuntimeException
        for (int i = 0; i < people.length; i++)
            System.out.println("people[" + i + "] contains "
                    + people[i]);
    }
}
```

## Try It:
Uncomment the first **for** loop to see an **ArrayIndexOutOfBoundsException**.

# Labs

❶ Modify *ParseTest.java* by adding a **catch** block for **Exception**, before the **catch** block for **ParseException**. Why does this generate a compiler error?
(Solutions: *ParseTest2.java*, *ParseTest2.txt*)

❷ Create an exception class called **BadAgeException**. Modify the **Person** class to contain an **age** field. Add an **int** argument to the **Person** constructor for the age. If the age given to the constructor is less than 0 or greater than 120, throw the **BadAgeException**. Modify the **UsePerson** class to test this constructor.
(Solutions: *BadAgeException.java*, *InvalidDataException.java*, *Person2.java*, *UsePerson2.java*)

# Unit Testing Concepts

❋   *Unit testing* verifies that a particular piece of code works properly in isolation.

➢   In Java, you typically create a unit test for each class with individual test cases for each **public** method.

➢   Your unit tests verify whether the methods under test produce the expected results given different inputs.

❋   *Test-Driven Development* (TDD) is a programming methodology that encourages you to create unit tests before adding methods.

➢   Your tests will initially fail since the code that they intend to test is not yet written.

➢   You know that you are done coding when the test passes.

▪   This keeps you from adding additional functionality that wasn't originally anticipated when you created the requirements for your software.

❋   *Regression testing* involves re-running all of your unit tests to ensure that everything is still working.

➢   Run your regression tests every time you add more features to your software.

➢   Run your regression tests every time you refactor your code.

▪   *Refactoring* is improving the code without adding functionality.

➢   Some organizations run regression tests nightly to ensure quality.

Suggested References:

*The Art of Software Testing* by Glenford J. Myers.

*Effective Unit Testing* by Lasse Koskela.

# JUnit

✳    *JUnit* is a lightweight test harness for automating unit testing in Java.

➢    You can download JUnit for free from **http://junit.org**.

▪    Installation is as simple as adding *junit.jar* and *hamcrest-core.jar* to the Java classpath.

➢    Eclipse bundles JUnit in both the Java and Java EE editions.

Over the next several pages, we will demonstrate JUnit by testing the **Account** class.

examples/Account.java
```java
package examples;
public class Account {
    private static int nextId = 0;
    private double balance;
    private String id;

    public Account() {
        this(0.0);
    }

    public Account(double initialBalance) {
        setBalance(initialBalance);
        setAccountId(nextId++);
    }

    public double getBalance() {
        return balance;
    }

    private void setBalance(double b) {
        if (b < 0) {
            throw new IllegalArgumentException(
                    "Negative balance is not allowed");
        }
        balance = b;
    }

    public String getAccountId() {
        return id;
    }

    private void setAccountId(int value) {
        id = "Acct-" + value;
    }

    public void deposit(double amount) {
        setBalance(balance + amount);
    }

    public void withdraw(double amount) {
        setBalance(balance - amount);
    }
}
```

# assertEquals()

❋ A JUnit test fixture consists of a class with one or more methods preceded by the **@Test** annotation.

➢ Each method should return **void** and have no parameters, but has no other restrictions.

❋ Use the overloaded *org.junit.Assert.assertEquals(expected, actual)* method to test whether the actual result of the method under test matches the expectation.

➢ If the expected and actual are not equal, then you will get an **AssertionError**.

❋ For floats and doubles, the **assertEquals()** method has an additional parameter: *delta.*

➢ The delta value provides a tolerance for equality between the expected and actual values.

❋ Another overload allows you to pass a **String** as a message to embed within the **AssertionError** if the test fails.

❋ If the expected result of a method invocation is an exception, add the **expected** attribute to the **@Test** annotation.

examples/AccountTest.java

```java
package examples;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class AccountTest {
    @Test
    public void testAccountConstructor() {
        Account a = new Account();
        assertEquals("Acct-1", a.getAccountId());
    }

    @Test
    public void testDeposit() {
        Account a = new Account();
        a.deposit(100.0);
        assertEquals(100.0, a.getBalance(), .01);
    }

    @Test
    public void testWithdraw() {
        Account a = new Account();
        a.deposit(100.0);
        a.withdraw(1.0);
        String message = "Withdraw test failed.";
        assertEquals(message, 99.0, a.getBalance(), .01);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testNegativeWithdrawal()
        throws IllegalArgumentException {

        Account a = new Account();
        a.withdraw(1.0);
    }
}
```

## Try It:

Your instructor will show you how to test *Account.java* using JUnit.

# Additional Assert Methods

❋ In addition to the **assertEquals()** method, the **Assert** class contains many others overloaded methods including:

- **assertArrayEquals()**
- **assertFalse()**
- **assertNotEquals()**
- **assertNotNull()**
- **assertNull()**
- **assertTrue()**
- **...**

➢ See the full JavaDoc documentation at: ***http://junit.org/javadoc/latest/ index.html***

# @Before and @After

✳ If you have setup code that should be run before each test case, you can place that code in a method annotated with **@Before**.

➢ Similarly, a method marked with **@After** will run after each test case.

examples/AccountTest2.java

```
package examples;

import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;

public class AccountTest2 {
    private Account a;

    @Before
    public void setup() {
        a = new Account();
    }

    @Test
     public void testAccountConstructor() {
        assertEquals("Acct-0", a.getAccountId());
    }

    @Test
    public void testDeposit() {
        a.deposit(100.0);
        assertEquals(100.0, a.getBalance(), .01);
    }
    ...
}
```

## Try It:

Run **AccountTest2** to make use of the method annotated with **@Before** that instantiates the **Account** object that is under test.

# Example: Testing a Triangle

examples/Triangle.java

```
package examples;

public class Triangle {
    private int side1;
    private int side2;
    private int side3;

    public Triangle(int s1, int s2, int s3) {
        // We don't allow negative or zero for the value of the sides
        if (s1 < 0 || s2 < 0 || s3 < 0)
            throw new IllegalArgumentException(
                    "A side cannot have a negative value");

        if (s1 == 0 || s2 == 0 || s3 == 0)
            throw new IllegalArgumentException(
                    "A side cannot have a zero value");

        // The length of two sides has to be greater than or equal to
        // the third side for it to be a valid triangle
        if ((s1 + s2 <= s3) || (s1 + s3 <= s2) || (s2 + s3 <= s1))
            throw new IllegalArgumentException("This isn't a triangle");

        side1 = s1;
        side2 = s2;
        side3 = s3;
    }

    public boolean isEquilateral() {
        if (side1 == side2 && side2 == side3)
            return true;
        else
            return false;
    }

    public boolean isScalene() {
        if (side1 != side2 && side2 != side3 && side1 != side3)
            return true;
        else
            return false;
    }
```

```
    public boolean isIsosceles() {
        if ((side1 == side2 && side2 != side3)
                || (side1 == side3 && side2 != side3)
                || (side2 == side3 && side1 != side3))
            return true;
        else
            return false;
    }
}
```

examples/TriangleTest.java

```
package examples;

import static org.junit.Assert.*;

import org.junit.Test;

public class TriangleTest {
    @Test
    public void testEquilateral() {
        Triangle t = new Triangle(3, 3, 3);
        assertTrue(t.isEquilateral());
    }

    @Test
    public void testIsosceles() {
        Triangle t = new Triangle(3, 3, 4);
        assertTrue(t.isIsosceles());
        t = new Triangle(3, 4, 3);
        assertTrue(t.isIsosceles());
        t = new Triangle(4, 3, 3);
        assertTrue(t.isIsosceles());
    }

    @Test
    public void testScalene() {
        Triangle t = new Triangle(3, 4, 5);
        assertTrue(t.isScalene());
    }

     @Test(expected = IllegalArgumentException.class)
    public void testZeroSide1() {
        new Triangle(0, 2, 2);
    }
    ...
```

**Try It:**

Test *Triangle.java* with *TriangleTest.java*.

# Labs

❶       Create a **Calculator** class. The class should have methods to add, subtract, multiply, and divide two numbers. Overload the **divide()** method with versions that take **int**s as well as **double**s. (Solution: *Calculator.java*)

❷       Create a set of test cases for the **Calculator** class. (Solution: *CalculatorTest.java*)

❸       Create a set of test cases for the **java.lang.String** class. Test the following **String** methods: **charAt(), concat(), contains(), endsWith(), indexOf(), isEmpty(), lastIndexOf(), length(), replace(), split(), startsWith(), substring(), toLowerCase(), toUpperCase(),** and **trim().** (Solution: *StringTest.java*)

❹       (Optional) Create unit tests for one or more classes that you have previously created.

# APPENDIX A - ECLIPSE SHORTCUTS

## OBJECTIVES

✸    Use various shortcuts to increase your
      productivity in Eclipse.

# Shortcut Key Sequences

❋     <Command><Space> — Content Assist

    ➤     Use this key sequence to have Eclipse generate a list of possible matches for the current entry in the code.

❋     <Command><Shift>O — Organize Imports

    ➤     Click on a class that has been flagged as "not resolved" and use this key sequence to have Eclipse automatically discover and add an **import** for the class as well as eliminate any **import**s that aren't being used.

❋     <Control>M — Maximize Active View or Editor

    ➤     This will take the current view or editor you are working in and maximize it; use the sequence again to go back to the original size.

❋     <Command><Shift>F — Format Code

    ➤     After you have configured Eclipse to understand your coding standards (Eclipse→Preferences→Java→Code Style), this handy key sequence will modify the current open Java file to match them.

❋     <Command>/ — Quick Comment

    ➤     This key sequence will comment out the current line or lines with a // comment; press it again to uncomment.

❋     <Command><Control>/ — Quick Block Comment

    ➤     Use this key sequence to comment out the current block of lines with a block comment **/* */**; use a backslash to remove a block comment.

Eclipse is loaded with features that tend to be hidden deep beneath menu options. To see a list of all of the shortcut keys, use <Command><Shift>L.

Another handy set of shortcuts to remember can be found under the Source menu. For example, the Source→Generate Getters and Setters menu item will create gets and sets for any of the fields in your class.

# More Shortcut Key Sequences

✳ <Command><Shift>T — Open Type

➢ Use the Open Type dialog to quickly jump to a class or interface in your project or in the Java libraries.

➢ For example, type **Str** for all classes that start with these three characters (**String**, **StringBuffer**, etc.)

✳ <Command><Shift>R — Open Resource

➢ Use the Open Resource dialog to find any kind of file in your project.

✳ <F3> — Open Declaration

➢ Any time you have clicked on a method call in code, you can click F3 to jump to the method declaration in the Java editor.

✳ <Command>O — Quick Outline

➢ Use this key sequence to launch a dialog that helps you quickly find a method within the current file by simply typing in the first few letters of the method you are looking for.

✳ <Command><Alt>↓ — Duplicate line.

✳ <Alt>↑ — Move line up.

✳ <Alt>↓ — Move line down.

In addition to key sequences, Eclipse also provides templates that allow you to quickly insert common code snippets. Type the template, followed by <Control><Space> to see the template replaced with the full syntax.

- **sysout** — **System.out.println()**
- **syserr** — **System.err.println()**
- **try** — **try/catch** block
- **catch** — **catch** block
- **main** — **main()** method

For more templates, see Eclipse→Preferences→Java→Editor →Templates.

# Index

# Skill Distillery

**7400 E. Orchard Road,  Suite 1450 N**
**Greenwood Village, Colorado  80111**
**303-302-5234**
**www.SkillDistillery.com**