Second AP* Edition
— with GridWorld

Java Methods

Object-Oriented Programming
and
Data Structures

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810

web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Copyright © 2011 by Maria Litvin, Gary Litvin, and Skylight Publishing

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the authors and Skylight Publishing.

Library of Congress Control Number: 2010915303

ISBN 978-0-9824775-7-1

* AP and Advanced Placement are registered trademarks of The College Board, which was not involved in the production of and does not endorse this book.

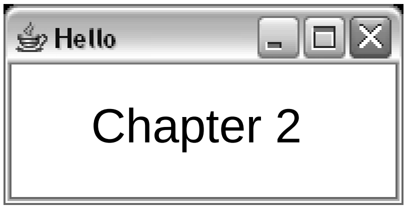
The names of commercially available software and products mentioned in this book are used for identification purposes only and may be trademarks or registered trademarks owned by corporations and other commercial entities. Skylight Publishing and the authors have no affiliation with and disclaim any sponsorship or endorsement by any of these product manufacturers or trademark owners.

Oracle, Java, and Java logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates in the U.S. and other countries.

SCRABBLE® is the registered trademark of HASBRO in the United States and Canada and of J.W. Spear and Sons, PLC, a subsidiary of Mattel, Inc., outside the United States and Canada.

1 2 3 4 5 6 7 8 9 10 16 15 14 13 12 11

Printed in the United States of America



Chapter 2

An Introduction to Software Engineering

Copyright © 2011 by Skylight Publishing

2.1 Prologue

One of the first computers, ENIAC,[eniac] developed in 1942-1946 primarily for military applications, was programmed by people actually connecting hundreds of wires to sockets (Figure 2-1) — hardly a "software development" activity as we know it. (ENIAC occupied a huge room, had 18,000 vacuum tubes, and could perform 300 multiplications per second.) In 1946, John von Neumann developed the idea that a computer program can be stored in the computer memory itself in the form of encoded CPU instructions, together with the data on which the program operates. Then the modern computer was born: a "universal, digital, program-stored" computer that can perform calculations and process information.

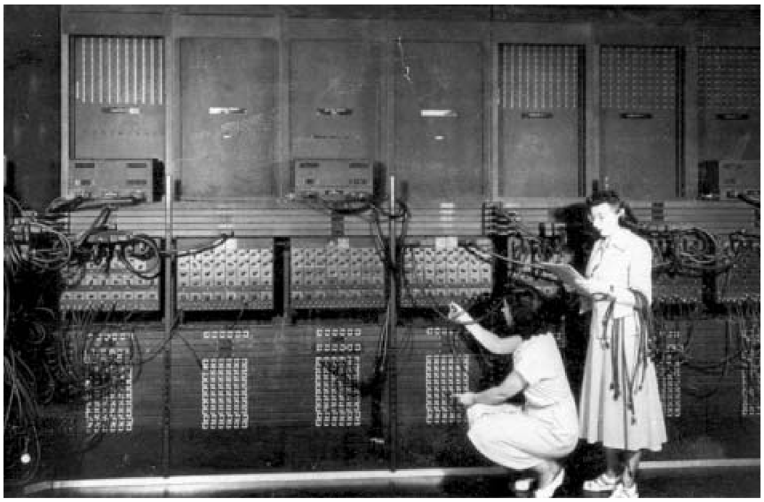


Figure 2-1. Two technicians wiring the right side of ENIAC

(Courtesy of U. S. Army Research Laboratory)

Copyright © 2011 by Skylight Publishing

Once program-stored computers were developed, it made sense to talk about programs as "written." In fact, at the beginning of the computer era, programmers wrote programs in pencil on special forms; then technicians punched the programs into punch cards[punchcard] or perforated tape. A programmer entering a computer room with a deck of punch cards was a common sight. Fairly large programs were written entirely in machine code using octal or hexadecimal instruction codes and memory addresses. It is no coincidence that the same word, "coding," is used for writing programs and encrypting texts. Programmers were often simply mathematicians, electrical engineers, or scientists who learned the skill on their own when they needed to use a computer for their work.

In those days computers and “computer time” (that is, the time available for running programs) were very expensive, much more expensive than a programmer’s time, and the high computer costs defined the rules of the game. For instance, only fairly important computer applications could be considered, such as military and scientific computations, large information systems, and so on. Programmers strove to make their programs run faster by developing efficient *algorithms* (the concept of an algorithm is described in Chapter 4). Often one or two programmers wrote the entire program and knew all about it, while no one else could understand it. Computer users were happy just to have access to a computer and were willing to learn cryptic instructions and formats for using programs.

Now, when computers are so inexpensive that they have become a household appliance, while programmers are relatively scarce and expensive, the rules of the game have changed completely. This change affects which programs are written, how they are created, and even the name by which programmers prefer to be called — “software engineers.” There is still a need, of course, for understanding and optimizing algorithms. But the emphasis has shifted to programmers’ productivity, professionalism, and teamwork — which requires using standard programming languages, tools, and software components.

Copyright © 2011 by Skylight Publishing

Software applications that run on a desktop computer are loaded with features and must be very interactive and “*user-friendly*,” (that is, have an intuitive and fairly conventional user interface). They must also be *portable* (that is, able to run on different computer systems) and internationalized (that is, easily adaptable for different languages and local conventions). Since a large team may work on the same software project, it is very important that teams follow standard development methodologies, and that the resulting programs be understandable to others and well documented. Thus software engineering has become as professionalized as other engineering disciplines: there is a lot of emphasis on knowing and using professional tools in a team environment, and virtually no room for solo wizardry.

A typical fairly large software project may include the following tasks:

- Interaction with customers, understanding customer needs, refining and formalizing specifications
- General design (defining a software product’s parts, their functions and interactions)
- Detailed design (defining objects, functions, algorithms, file layouts, etc.)
- Design/prototyping of the user interface (designing screen layouts, menus, dialog boxes, online help, reports, messages, etc.)

- Coding and debugging
- Performance analysis and code optimization
- Documentation
- Testing
- Packaging and delivery
- User technical support

And, in the real world:

- Bug fixes, patches and workarounds, updated releases, documentation updates, and so on.

Of course there are different levels and different kinds of software engineers, and it is not necessary that the same person combine all the skills needed to design and develop good software. Usually it takes a whole team of software designers, programmers, artists, technical writers, QA (Quality Assurance) specialists, and technical support people.

In this chapter we will first discuss general topics related to software development, such as high-level programming languages and software development tools. We will discuss the difference between compilers and interpreters and Java's hybrid compiler + interpreter approach. Then we will learn how to compile and run simple Java applications and applets and take a first look at the concepts involved in object-oriented programming.

2.2 Compilers and Interpreters

Computer programmers very quickly realized that the computer itself was the perfect tool to help them write programs. The first step toward automation was made when programmers began to use *assembly languages* instead of numerically coded CPU instructions. In an assembly language, every CPU instruction has a short mnemonic name. A programmer can give symbolic names to memory locations and can refer to these locations by name. For example, a programmer using assembly language for Intel's 8086 microprocessor can write:

Copyright © 2011 by Skylight Publishing

```
index   dw     0          ; "Define word" -- reserve 2 bytes
                          ;  for an integer and call it "index".
        ...
        mov    si,index   ; Move the value of index into
                          ;   the SI register.
        ...
```

A special program, called the *assembler*, converts the text of a program written in assembly language into the *machine code* expected by the CPU.

Obviously, assembly language is totally dependent on a particular CPU; *porting* a program to a different type of machine would require rewriting the code. As the power of computers increased, several *high-level* programming languages were developed for writing programs in a more abstract, machine-independent way. FORTRAN (Formula Translation Language) was defined in 1956, COBOL (Common Business Oriented Language) in 1960, and Pascal and C in the 1970s. C++ gradually evolved from C in the 1980s, adding OOP (Object-Oriented Programming) features to C.[languagehistory] Java was introduced in the mid-1990s and eventually gained popularity as a fully object-oriented programming language for platform-independent development, in particular for programs transmitted over the Internet. Java and OOP are of course the main subjects of this book, so we will start looking at them in detail in the following chapters.

A program written in a high-level language obeys the very formal *syntax* rules of the language. This syntax produces statements so unambiguous that even a computer can interpret them correctly. In addition to strict syntax rules, a program follows *style* conventions; these are not mandatory but make the program easier to read and understand for fellow programmers, demonstrating its author’s professionalism.

❖ ❖ ❖

A programmer writes the text of the program using a software program called an *editor*. Unlike general-purpose word-processing programs, program editors may have special features useful for writing programs. For example, an editor may use colors to highlight different syntactic elements in the program or have built-in tools for entering standard words or expressions common in a particular programming language.

The text of a program in a particular programming language is referred to as *source code*, or simply the *source*. The source code is stored in a file, called the *source file*.

Copyright © 2011 by Skylight Publishing

Before it can run on a computer, a program written in a high-level programming language has to be somehow converted into CPU instructions. One approach is to use a special software tool called a *compiler*. The compiler is specific to a particular programming language and a particular CPU. It analyzes the source code and generates appropriate CPU instructions. The result is saved in another file, called the *object module*. A large program may include several source files that are compiled into object modules separately. Another program, a *linker*, combines all the object modules into one *executable* program and saves it in an executable file (Figure 2-2).

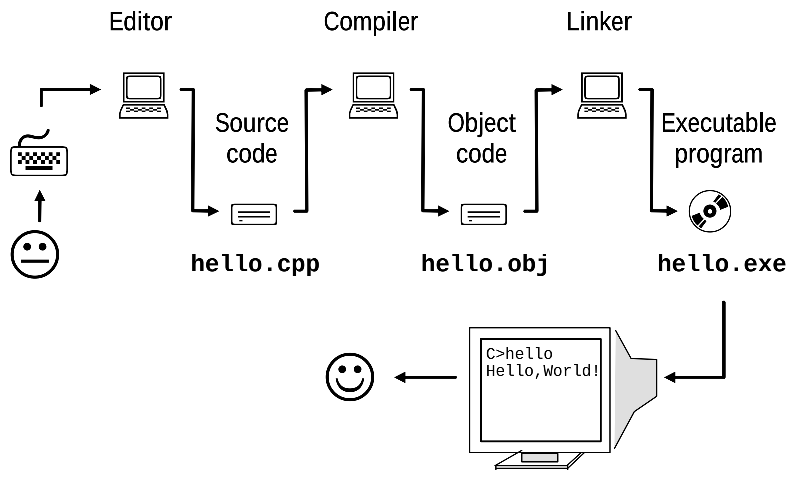


Figure 2-2. Software development cycle for a compiled program: edit-compile-link-run

For a compiled program, once it is built and tested, the executable file is distributed to program users. The users do not need access to the program's source code and do not need to have a compiler.

Java also uses a compiler, but, as we will explain shortly, the Java compiler does not generate object code.

In an alternative approach, instead of compiling, a program in a high-level language can be *interpreted* by a software tool called an *interpreter*. The difference between a compiler and an interpreter is subtle but important. An interpreter looks at the high-level language program, figures out what instructions it needs to execute, and executes them. But it does not generate an object-code file and does not save any

Copyright © 2011 by Skylight Publishing

compiled or executable code. A user of an interpreted program needs access to the program's source code and an interpreter, and the program has to be interpreted again each time it is run. It is like a live concert as opposed to a studio recording, and a live performance needs all the instruments each time.

❖ ❖ ❖

A particular programming language is usually established as either a compiled language or an interpreted language (that is, is either more often used with a compiler or an interpreter, respectively). FORTRAN, COBOL, Ada, C++ are typically compiled; BASIC, Perl, Python are interpreted. But there is really no clear-cut distinction. BASIC, for example, was initially an interpreted language, but soon BASIC compilers were developed. C is usually compiled, but C interpreters also exist.

Java is different: it uses a mixed compiler-plus-interpreter approach. A Java compiler first compiles the program into *bytecode*, instructions that are pretty close to a machine language. But a machine with this machine language does not exist! It is an abstract computer, a *Java Virtual Machine* (*JVM*). The bytecode is then <u>interpreted</u> on a particular computer by the Java interpreter for that particular CPU. A program in bytecode is not object code, because it is still platform-independent (it does not use instructions specific to a particular CPU). It is not source code, either, because it is not readable by humans. It is something in between.

Copyright © 2011 by Skylight Publishing

Why does Java use a combination of a compiler and an interpreter? There is no reason why a regular Java compiler couldn't be created for a particular type of computer. But one of the main purposes of Java is to deliver programs to users via the Internet. A *Java-enabled* browser (that is, a browser that has a Java interpreter built into it) can run little Java programs, called *applets* (miniature applications). The many applets available free on the Internet, often with their source code, are one of the reasons why Java has become so popular so fast. When you connect to a web site and see some elaborate action or interactive features, it may mean that your computer has received a Java applet and is running it.

Java designers had to address the key question: Should users receive Java source code or executable code? The answer they came up with was: neither. If users got source, their browsers would need a built-in Java compiler or interpreter. That would make browsers quite big, and compiling or interpreting on the user's computer could take a long time. Also, software providers may want to keep their source confidential. But if users got executables, then web site operators would somehow need to know what kind of computer each user had (for example, a PC or a Mac) and deliver the right versions of programs. It would be cumbersome and expensive for web site operators to maintain different versions of a program for every different platform. There would also be a security risk: What if someone delivered a malicious program to your computer?

Bytecode provides an intermediate step, a compromise between sending source code or executables to users (Figure 2-3). On one hand, the bytecode' language is platform-independent, so the same version of bytecode can serve users with different types of computers. It is not readily readable by people, so it can protect the confidentiality of the source code. On the other hand, bytecode is much closer to the "average" machine language, and it is easier and faster to interpret than "raw" Java source. Also, bytecode interpreters built into browsers get a chance to screen programs for potential security violations (for example, they can block reading of and writing to the user's disks).

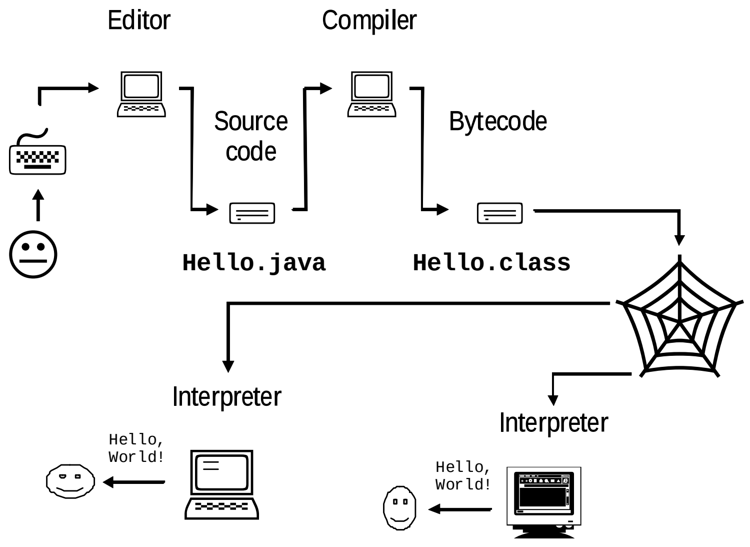


Figure 2-3. Java software development and distribution through the Internet

Copyright © 2011 by Skylight Publishing

To speed up the loading of applets, a new software technology has emerged, called *JIT* (Just-In-Time) compilers. A JIT compiler combines the features of a compiler and an interpreter. While interpreting bytecode, it also compiles it into executable code. (To extend our music analogy, a JIT compiler works like a recording of a live concert.) This means an applet can be interpreted and start running as soon as it is downloaded from the Internet. On subsequent runs of the same applet, it can be loaded and run from its executable file without any delay for reinterpreting bytecode.

Naturally, bytecode does not have to travel through the Internet to reach the user: a Java program can be compiled and interpreted on the same computer. That is what we will do for testing Java applications in our labs and exercises. We do not even have to use a browser to test an applet: the standard Java Development Kit (JDK) has a program, called *Applet Viewer*, that runs applets.

❖ ❖ ❖

Modern software development systems combine an editor, a compiler, and other tools into one *Integrated Development Environment* (*IDE*). Some of the software development tools (a program editor, for example) are built into the IDE program itself; larger tools (a compiler, an interpreter) are usually stand-alone programs, for which the IDE only serves as a *front end*. An IDE has a convenient *GUI* (*Graphical User Interface*) — one mouse click on an icon will compile and run your program.

Modern programs may be rather complex, with dozens of different types of objects and functions involved. *Structure analyzers* and viewers built into an IDE create graphical views of source files, objects, their functions, and the dependencies between them. GUI *visual prototyping and design tools* help a programmer design and implement a graphical user interface.

Few programs are written on the first try without errors or, as programmers call them, *bugs* (Figure 2-4).

Copyright © 2011 by Skylight Publishing

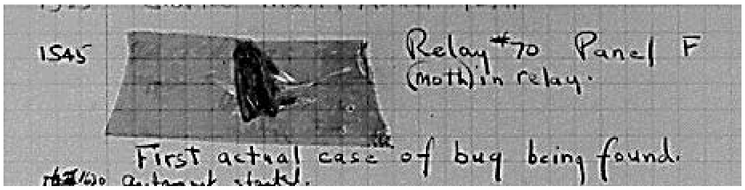


Figure 2-4. The term "bug" was popularized by Grace Hopper,❀hopper a legendary computer pioneer, who was the first to come up with the idea of a compiler and who created COBOL. One of Hopper's favorite stories was about a moth that was found trapped between the points of a relay, which caused a malfunction of the Mark II Aiken Relay Calculator (Harvard University, 1945). Technicians removed the moth and affixed it to the log shown in the photograph.

Programmers distinguish *syntax errors* and *logic errors*.

Syntax errors violate the syntax rules of the language and are caught by the compiler. Logic errors are caused by flawed logic in the program; they are not caught by the compiler but show up at "run-time," that is when the program is running. Some run-time errors cause an *exception*: the program encounters a fatal condition and is aborted with an error message, which describes the type of the exception and the program statement that caused it. Other run-time errors may cause program's unexpected behavior or incorrect results. These are caught only in thorough testing of the program.

It is not always easy to correct bugs just by looking at the source code or by testing the program on different data. To help with this, there are special *debugger* programs that allow the programmer to trace the execution of a program "in slow motion." A debugger can suspend a program at a specified break point or step through the program statements one at a time. With the help of a debugger, the programmer can examine the sequence of operations and the contents of memory locations after each step.

2.3 Software Components and Packages

Writing programs from scratch may be fun, like growing your own tomatoes from seeds, but in the present environment few people can afford it. An amateur, faced with a programming task, asks: What is the most original (elegant, efficient, creative, interesting, etc.) way to write this code? A professional asks: What is the way to <u>not</u> write this code but use something already written by someone else? With billions of lines of code written, chances are someone has already implemented this or a similar task, and there is no point duplicating his or her efforts. (A modern principle, but don't try it with your homework!) Software is a unique product because all of its production cost goes into designing, coding and testing <u>one</u> copy; manufacturing multiple copies is virtually free. So the real task is to find out what has been done, purchase the rights to it if it is not free, and reuse it.

There are many sources of reusable code. Extensive software packages come with your compiler. Other packages may be purchased from third-party software vendors who specialize in developing and marketing reusable software packages to developers. Still other packages may be available for free in the spirit of the open source opensource philosophy. In addition, every experienced programmer has accumulated his or her own collection of reusable code.

Reusability of software is a two-sided concept. As a programmer, you want to be more efficient by reusing existing code. But you also want to write reusable code so that you yourself, your teammates, your enterprise, and/or the whole world can take advantage of it later. Creating reusable code is not automatic: your code must meet certain requirements to be truly reusable. Here is a partial list of these requirements:

Copyright © 2011 by Skylight Publishing

- Your code must be divided into reasonably small parts or components (modules). Each component must have a clear and fairly general purpose. Components that implement more general functions must be separated from more specialized components.
- Your software components must be well documented, especially the interface part, which tells the user (in this case, another programmer) what this component does and how exactly to use it. A user does not necessarily always want to know how a particular component does what it does.
- The components must be robust. They must be thoroughly tested under all possible conditions under which the component can be used, and these conditions must be clearly documented. If a software module encounters conditions under which it is not supposed to work, it should handle such situations gracefully, giving its user a clue when and why it failed instead of just crashing the system.
- It should be possible to customize or extend your components without completely rewriting them.

Individual software components are usually combined into *packages*. A package combines functions that deal with a particular set of structures or objects: a graphics package that deals with graphics capabilities and display; a text package that manipulates strings of text and text documents; a file package that helps to read and write data files; a math package that provides mathematical functions and algorithms; and so on. In Chapter 20, we will talk about Java collections classes, which are part of the `java.util` package from the standard Java library. Java programmers can take advantage of dozens of standard packages that are already available for free; new packages are being developed all the time. At the same time, the plenitude of available packages and components puts an additional burden on the software engineer, who must be familiar with the standard packages and keep track of the new ones.

Copyright © 2011 by Skylight Publishing

2.4 *Lab:* Three Ways to Say Hello

A traditional way to start exploring a new software development environment is to write and get running a little program that just prints “Hello, World!” on the screen. After doing that, we will explore two other very simple programs. Later, in Section 2.6, we will look at simple GUI applications and a couple of applets.

In this section, we will use the most basic set of tools, JDK (Java Development Kit). JDK comes from Sun Microsystems, Inc., the makers and owners of Java.

JDK includes a compiler, an interpreter, the *Applet Viewer* program, other utility programs, the standard Java library, documentation, and examples.

JDK itself does not have an IDE (Integrated Development Environment), but Sun and many third-party vendors and various universities and other organizations offer IDEs for running Java. *Eclipse, BlueJ, JCreator* are some examples, but there are dozens of others. This book's companion web site, `www.skylit.com/javamethods`, has a list of several development environments and *FAQs* (*Frequently Asked Questions*) about installing and using some of them.

In this lab the purpose is to get familiar with JDK itself, without any IDE. However, if you don't feel like getting your hands dirty (or if you are not authorized to run command-line tools on your system), you can start using "power" tools right away. Just glance through the text and then use an IDE to type in and test the programs in this lab.

We assume that by now you have read Sun's instructions for installing and configuring JDK under your operating system and have it installed and ready to use. In this lab you can test that your installation is working properly. If you are not going to use command-line tools, then you need to have an IDE installed and configured as well.

This lab involves three examples of very simple programs that do not use GUI, just text input and output. Programs with this kind of old-fashioned user interface are often called *console applications* (named after a teletype device called a console, which they emulate). Once you get the first program running, the rest should be easy.

Our examples and commands in this section are for *Windows*.

1. Hello, World

JDK tools are UNIX-style *command-line tools*, which means the user has to type in commands at the system prompt to run the compiler, the interpreter, or *Applet Viewer*. The compiler is called `javac.exe`, the interpreter is called `java.exe`, and *Applet Viewer* is called `appletviewer.exe`. These programs reside in the `bin` subfolder of the folder where your JDK is installed. This might be, for example, `C:\Program Files\Java\jdk1.6.0_21\bin`. You'll need to make these programs accessible from any folder on your computer. To do that, you need to set the *path environment variable* to include JDK's `bin` folder. There is a way to make

Copyright © 2011 by Skylight Publishing

a permanent change to the path, but today we will just type it in once or twice, because we don't plan on using command-line tools for long.

Create a work folder (for example, `C:\mywork`) where you will put your programs from this lab. You can use any editor (such as *Notepad*) or word processor (such as *Wordpad* or *MS Word*) or the editor from your IDE to enter Java source code. If you use a word processor, make sure you save Java source files as "Text Only." But the file extension should be `.java`. Word processors such as *Word* tend to attach the `.txt` extension to your file automatically. The trick is to first choose `Save as type: Text-Only (*.txt)`, and only after that type in the name of your file with the correct extension (for example, `HelloWorld.java`).

In your editor, type in the following program and save it in a text file `HelloWorld.java`:

```
/**
 *  Displays a "Hello World!" message on the screen
 */
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello, World!");
  }
}
```

In Java, names of files are case sensitive.

This is true even when you run programs in a *Command Prompt* window. Make sure you type in the upper and lower cases correctly.

In the little program above, `HelloWorld` is the name of a class as well as its source file. (Don't worry if you don't quite know what that means, for now.)

The name of the file that holds a Java class must be exactly the same as the name of that class (plus the extension `.java`).

This rule prevents you from having two runnable versions of the same class in the same folder. Make sure you name your file correctly. There is a convention that the name of a Java class (and therefore the name of its Java source file) always starts with a capital letter.

Copyright © 2011 by Skylight Publishing

The Java interpreter calls the `main` method in your class to start your program. Every application (but not an applet) must have a `main` method. The one in your program is:

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

For now, treat this as an idiom. You will learn the meaning of the words `public`, `static`, `void`, `String`, and `args` later.

`System` is a class that is built into all Java programs. It provides a few system-level services. `System.out` is a data element in this class, an object that represents the computer screen output device. Its `println` method displays a text string on the screen.

Examine what you have typed carefully and correct any mistakes — this will save time.

Save your file and close the editor. Open the *Command Prompt* window (you'll find it under *All Programs/Accessories* on your *Start* menu). Navigate to the folder that contains your program (for example, `mywork`) using the `cd` (change directory) command, and set the path:

```
C:\Documents and Settings\Owner>cd \mywork
C:\mywork> path C:\program files\java\jdk1.6.0_21\bin;%PATH%
```

Now compile your program:

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

If you have mistyped something in your source file, you will get a list of errors reported by the compiler. Don't worry if this list is quite long, as a single typo can cause several errors. Verify your code against the program text above, eliminate the typos, and recompile until there are no errors.

Type the `dir` (directory) command:

```
C:\mywork> dir
```

You should see files called `HelloWorld.java` and `HelloWorld.class` in your folder. The latter is the bytecode file created by the compiler.

Copyright © 2011 by Skylight Publishing

Now run the Java interpreter to execute your program:

```
C:\mywork> java HelloWorld
```

Every time you make a change to your source code, you'll need to recompile it. Otherwise the interpreter will work with the old version of the `.class` file.

2. Greetings

A Java application can accept "command-line" arguments from the operating system. These are words or numbers (character strings separated by spaces) that the user can enter on the command line when he runs the program. For example, if the name of the program is *Greetings* and you want to pass two arguments to it, "Annabel" and "Lee", you can enter:

```
C:\mywork> java Greetings Annabel Lee
```

If you are using an IDE, it usually has an option, a dialog box, where you can enter command-line arguments before you run the program.

If you are already using your IDE and do not feel like figuring out how to enter command-line arguments in it, skip this exercise.

The following Java program expects two command-line arguments.

```
/**
 *  This program expects two command-line arguments
 *  -- a person's first name and last name.
 *  For example:
 *  C:\mywork> java Greetings Annabel Lee
 */
public class Greetings
{
  public static void main(String[] args)
  {
    String firstName = args[0];
    String lastName = args[1];
    System.out.println("Hello, " + firstName + " " + lastName);
    System.out.println("Congratulations on your second program!");
  }
}
```

Type this program in using your editor and save it in the text-only file `Greetings.java`. Compile this program:

Copyright © 2011 by Skylight Publishing

```
C:\mywork> javac Greetings.java
```

Now run it with two command-line arguments: your first and last name.

3. More Greetings

Now we can try a program that will *prompt* you for your name and then display a message. You can modify the previous program. Start by saving a copy of it in the text file `Greetings2.java`.

```
/*
    This program prompts the user to enter his or her
    first name and last name and displays a greeting message.
    Author: Maria Litvin
*/

import java.util.Scanner;

public class Greetings2
{
  public static void main(String[] args)
  {
    Scanner kboard = new Scanner(System.in);
    System.out.print("Enter your first name: ");
    String firstName = kboard.nextLine();
    System.out.print("Enter your last name: ");
    String lastName = kboard.nextLine();
    System.out.println("Hello, " + firstName + " " + lastName);
    System.out.println("Welcome to Java!");
  }
}
```

Our `Greetings2` class uses a Java library class `Scanner` from the `java.util` package. This class helps to read numbers, words, and lines from keyboard input. The `import` statement at the top of the program tells the Java compiler where it can find `Scanner.class`.

Compile `Greetings2.java` —

```
C:\mywork> javac Greetings2.java
```

— and run it:

```
C:\mywork> java greetings2
```

What do you get?

Copyright © 2011 by Skylight Publishing

```
Exception in thread "main" java.lang.NoClassDefFoundError: greetings2 (wrong
name: Greetings2)
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:620)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:124)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:260)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:56)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:195)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:268)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
        at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:319)
```

Wow! The problem is, you entered `greetings2` with a lowercase "G", and the Java interpreter cannot find a file called `greetings2.class`. Remember: Java is case-sensitive. You can see now why you might want some help from an IDE!

Try again:

```
C:\mywork> java Greetings2
```

Now the program should run: it prompts you for your first and last name and displays a greeting message:

```
C:\mywork> java Greetings2
Enter your first name: Virginia
Enter your last name: Woolf
Hello, Virginia  Woolf
Welcome to Java!
```

Copyright © 2011 by Skylight Publishing

2.5 Object-Oriented Programming

In von Neumann computer architecture, a program is a sequence of instructions executed by a CPU. Blocks of instructions can be combined into *procedures* that perform a certain calculation or carry out a certain task; these can be called from other places in the program. Procedures manipulate some data stored elsewhere in computer memory. This *procedural* way of thinking is suggested by the hardware architecture, and naturally it prevailed in the early days of computing. In *procedural programming*, a programmer has an accurate picture of the order in which instructions might be executed and procedures might be called. High-level *procedural languages* don't change that fact. One statement translates into several CPU instructions and groups of statements are combined into functions, but the nature of programming remains the same: the statements are executed and the

functions are called in a precise order imposed by the programmer. These procedures and functions work on separately defined data structures.
In the early days, user interface took the form of a dialog: a program would show prompts asking for data input and display the results at the end, similar to the *Greetings2* program in the previous section. This type of user interface is very orderly — it fits perfectly into the sequence of a procedural program. When the concept of *graphical user interface* (*GUI*) developed, it quickly became obvious that the procedural model of programming was not very convenient for implementing GUI applications. In a program with a GUI, a user sees several GUI components on the screen at once: menus, buttons, text entry fields, and so on. Any of the components can generate an event: things need to happen whenever a user chooses a menu option, clicks on a button, or enters text. A program must somehow handle these events in the order of their arrival. It is helpful to think of these GUI components as animated objects that can communicate with the user and other objects. Each object needs its own memory to represent its current state. A completely different programming model is needed to implement this metaphor. *Object-oriented programming* (*OOP*) provides such a model.

The OOP concept became popular with the introduction of Smalltalk,[smalltalk] the first general-purpose object-oriented programming language with built-in GUI development tools. Smalltalk was developed in the early 1970s by Alan Kay[kay] and his group at the Xerox Palo Alto Research Center. Kay dreamed that when inexpensive personal computers became available, every user, actually every child, would be able to program them; OOP, he thought, would make this possible. As we know, that hasn’t quite happened. Instead, OOP first generated a lot of interest in academia as a research subject and a teaching tool, and then was gradually embraced by the software industry, along with C++, and later Java, as the preferred way of designing and writing software.

One can think of an OOP application as a virtual world of active objects. Each object has its own “memory,” which may contain other objects. Each object has a set of *methods* that can process messages of certain types, change the object’s state (memory), send messages to other objects, and create new objects. An object belongs to a particular class, and each object’s functionality, methods, and memory structure are determined by its class. A programmer creates an OOP application by defining classes.

Two principles are central to the OOP model: *event-driven* programs and *inheritance*.

In an OOP program many things may be happening at once, and external events (for example, the user clicks the mouse or types a key, the application’s window is

Copyright © 2011 by Skylight Publishing

resized, etc.) can determine the order of program execution. An OOP program, of course, still runs on sequential von Neumann computers; but the software simulates parallelism and asynchronous handling of events.

An OOP program usually defines many different types of objects. However, one type of objects may be very similar to another type. For instance, objects of one type may need to have all the functionality of another type plus some additional features. It would be a waste to duplicate all the features of one class in another. The mechanism of *inheritance* lets a programmer declare that one class of objects *extends* another class. The same class may be extended in several different ways, so one *superclass* may have several *subclasses* derived from it (Figure 2-5). A subclass may in turn be a superclass for other classes, such as `Music` is for `Audio` and `MP3`. An application ends up looking like a branching tree, a hierarchy of classes. Classes with more general features are closer to the top of the hierarchy, while classes with more specific functionality are closer to the bottom.

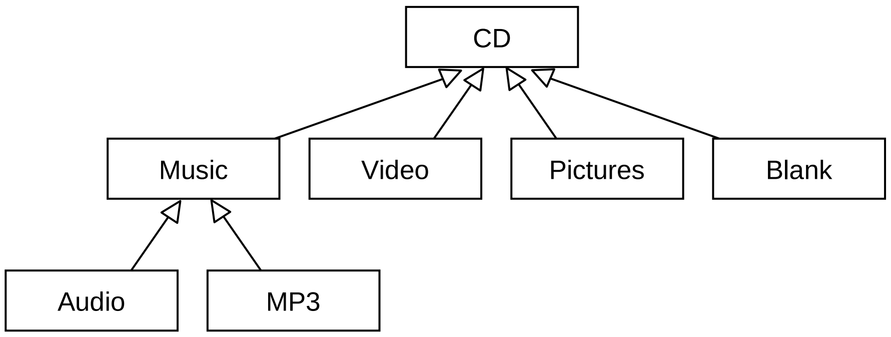


Figure 2-5. A hierarchy of classes that represent compact disks with different content

Copyright © 2011 by Skylight Publishing

Object-oriented programming aims to answer the current needs in software development: lower software development and documentation costs, better coordinated team development, accumulation and reuse of software components, more efficient implementation of multimedia and GUI applications, and so on. Java is a fully object-oriented language that supports inheritance and the event-driven model. It includes standard packages for graphics, GUI, multimedia, events handling, and other essential software development tools.

Our primary focus in this book is working with hierarchies of classes. Event-driven software and events handling in Java are considered to be more advanced topics. For example, they are not included in the Advanced Placement Computer Science course description. We will discuss events handling in Java and provide examples in Chapters 17 and 18.

2.6 *Lab:* More Ways to Say Hello

In Section 2.4 we learned how to run very simple console applications. These types of programs, however, are not what makes Java great: they can be easily written in other programming languages.

The features that distinguish Java from other languages are its built-in support for GUI and graphics and its support for object-oriented programming.

In this section we will consider four more examples: two applications (one with a simple GUI object, another with graphics), and two applets (one with graphics and one with animation). Of course at this stage you won't be able to understand all the code in these examples — we have a whole book ahead of us! This is just a preview of things to come, a chance to get a general idea of what is involved and see how these simple programs work.

1. A GUI application

In this program, `HelloGui.java`, we create a standard window on the screen and place a "Hello, GUI!" message in it. Our `HelloGui` class *extends* the `JFrame` library class, which is part of Java's *Swing* package. We are lucky we can reuse `JFrame`'s code: it would be a major job to write a class like this from scratch. We would have to figure out how to show the title bar and the border of the window and how to support resizing of the window and other standard functions. `JFrame` takes care of all this. All we have left to do is add a label to the window's *content pane* — the area where you can place GUI components.

Our `HelloGui` class is shown in Figure 2-6. In this program, the `main` method creates one object, which we call `window`. The type of this object is described as `HelloGui`; that is, `window` is an object of the `HelloGui` class. This program uses only one object of this class. `main` then sets `window`'s size and position (in pixels) and displays it on the screen. Our class has a *constructor*, which is a special procedure for constructing objects of this class. Constructors always have the same name as the class. Here the constructor calls the superclass's constructor to set the text displayed in the window's title bar and adds a label object to the window's content pane.

Copyright © 2011 by Skylight Publishing

```
/**
 *  This program displays a message in a window.
 */

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

public class HelloGui extends JFrame
{
  public HelloGui()   // Constructor
  {
    super("GUI Demo");    // Set the title bar
    Container c = getContentPane();
    c.setBackground(Color.CYAN);
    c.setLayout(new FlowLayout());
    c.add(new JTextField(" Hello, GUI!", 10));
  }

  public static void main(String[] args)
  {
    HelloGui window = new HelloGui();

    // Set this window's location and size:
    // upper-left corner at 300, 300; width 200, height 100
    window.setBounds(300, 300, 200, 100);

    window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    window.setVisible(true);
  }
}
```

Figure 2-6. `JM\Ch02\HelloGui\HelloGui.java`

The code in Figure 2-6 is a little cryptic, but still we can see roughly what's going on. Do not retype the program — just copy `HelloGui.java` form the `JM\Ch02\HelloGui` folder into your current work folder. Set up a project in your favorite IDE, and add the class, `HelloGui`. Compile and run the program using menu commands, buttons, or shortcut keys in your IDE.

Copyright © 2011 by Skylight Publishing

2. Hello, Graphics

We will now change our program a little to paint some graphics on the window instead of a text label. The new class, `HelloGraphics`, is shown in Figure 2-7.

```
//  This program shows simple graphics in a window.

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

public class HelloGraphics extends JPanel
{
  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);  // Call JPanel's paintComponent method
                              //  to paint the background
    g.setColor(Color.RED);

    // Draw a 150 by 45 rectangle with the upper-left
    // corner at x = 25, y = 40:
    g.drawRect(20, 40, 150, 45);

    g.setColor(Color.BLUE);

    // Draw a string of text starting at x = 60, y = 25:
    g.drawString("Hello, Graphics!", 55, 65);
  }

  public static void main(String[] args)
  {
    JFrame window = new JFrame("Graphics Demo");
    // Set this window's location and size:
    // upper-left corner at 300, 300; width 200, height 150
    window.setBounds(300, 300, 200, 150);
    window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    HelloGraphics panel = new HelloGraphics();
    panel.setBackground(Color.WHITE);  // the default color is light gray
    Container c = window.getContentPane();
    c.add(panel);

    window.setVisible(true);
  }
}
```

Figure 2-7. `JM\Ch02\HelloGui\HelloGraphics.java`

Copyright © 2011 by Skylight Publishing

`HelloGraphics` extends a library class `JPanel`. Each `JPanel` object has a `paintComponent` method that generates all the graphics contents for the panel. `paintComponent` is called automatically whenever the window is opened, resized, or repainted. These events are reported to the program by the operating system.

By default, `JPanel`'s `paintComponent` method only paints the background of the panel. Our class `HelloGraphics` redefines (overrides) `paintComponent` to add a blue message inside a red box. `paintComponent` receives an object of the type `Graphics`, often called `g`, that represents the panel's graphics context (its position, size, etc.).

The graphics coordinates are in pixels and have the origin (0, 0) at the upper-left corner of the panel (the *y*-axis points down).

We have placed the `main` method into the same class to simplify things. If you wish, you can split our `HelloGraphics` class into two separate classes: one, call it `HelloPanel`, will extend `JPanel` and have the `paintComponent` method; the other, call it `HelloGraphics` will have `main` and nothing else (it doesn't have to extend any library class). Your project should include both classes.

3. Hello, Applet

Applets are small programs embedded in web pages and distributed over the Internet. From a programmer's point of view, the difference between an applet and a GUI application is minor. Instead of extending `JFrame`, your applet's class extends `JApplet`, a *Swing* library class that represents applet objects. An applet does not need a `main` method because the browser (or *Applet Viewer*) automatically constructs the applet object and displays it as part of a web document. Instead of a constructor, your applet class uses the `init` method to initialize your applet. Figure 2-8 shows the `HelloApplet` class adapted from the `HelloGraphics` class. This applet redefines `JApplet`'s `paint` method to show graphics.

Copyright © 2011 by Skylight Publishing

```
/*
 * This applet shows a string of text inside a box.
 */

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

public class HelloApplet extends JApplet
{
  public void init()
  {
    Container c = getContentPane();
    c.setBackground(Color.WHITE);
  }

  public void paint(Graphics g)
  {
    super.paint(g);    // call JApplet's paint method
                       //  to paint the background
    g.setColor(Color.RED);
    g.drawRect(25, 40, 150, 45);  // draw a rectangle 150 by 45
    g.setColor(Color.BLUE);
    g.drawString("Hello, Applet!", 60, 65);
  }
}
```

Figure 2-8. **`JM\Ch02\HelloGui\HelloApplet.java`**

Copyright © 2011 by Skylight Publishing

The code for this applet is shorter than `HelloGraphics`. But now we need another file that describes a web page that presents the applet. The contents and layout of web pages are usually described in HTML (Hypertext Mark-Up Language). You can find a brief HTML tutorial in Appendix C.⚝ Here we can use a very simple HTML file (Figure 2-9). Let's call it `TestApplet.html`. As you can see, some of the information — the size of the applet's content pane — has shifted from Java code into the HTML file. The title bar is no longer used because an applet does not run in a separate window — it is embedded into a browser's (or *Applet Viewer*'s) window. An applet does not have an exit button either (the browser's or *Applet Viewer*'s window has one).

```
<html>

<head>
<title>My First Java Applet</title>
</head>

<body>
<applet code="HelloApplet.class" width="300" height="100"
   alt="Java class failed">
Java is disabled
</applet>
</body>

</html>
```

Figure 2-9. **`JM\Ch02\HelloGui\TestApplet.html`**

You can either test your applet directly in your IDE or open it in your Internet browser. If you have a website, you can upload the `TestApplet.html` page to your site, along with the `HelloApplet.class` file, for the whole world to see.

You can adapt `TestApplet.html` to run another applet by replacing `HelloApplet.class` in it with the name of your new applet class and adjusting the applet's size, if necessary.

4. Hello, Action

And now, just for fun, let's put some action into our applet (Figure 2-10). Compile the `Banner` class from `JM\Ch02\HelloGui` and open the `TestBanner.html` file (from the same folder) in the *Applet Viewer* or in the browser to test this applet.

Look at the code in `Banner.java`. The `init` method in this applet creates a `Timer` object called `clock` and starts the timer. The timer is programmed to fire every 30 milliseconds. Whenever the timer fires, it generates an event that is captured in the `actionPerformed` method. This method adjusts the position of the banner and repaints the screen.

You might notice that unfortunately the animation effect in this applet is not very smooth: the screen flickers whenever the banner moves. One of the advantages of Java's *Swing* package is that it can help deal with this problem. We will learn how to do it in later chapters.

Copyright © 2011 by Skylight Publishing

```
/* This applet displays a message moving horizontally
   across the screen. */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Banner extends JApplet
  implements ActionListener
{
  private int xPos, yPos;  // hold the coordinates of the banner

  public void init()
  {
    Container c = getContentPane();
    c.setBackground(Color.WHITE);
    xPos = c.getWidth();
    yPos = c.getHeight() / 2;
    Timer clock = new Timer(30, this);  // fires every 30 milliseconds
    clock.start();
  }

  // Called automatically after a repaint request
  public void paint(Graphics g)
  {
    super.paint(g);
    g.drawString("Hello, World!", xPos, yPos);
  }

  // Called automatically when the timer fires
  public void actionPerformed(ActionEvent e)
  {
    Container c = getContentPane();

    // Adjust the horizontal position of the banner:
    xPos--;
    if (xPos < -100)
    {
      xPos = c.getWidth();
    }

    // Set the vertical position of the banner:
    yPos = c.getHeight() / 2;

    repaint();
  }
}
```

Figure 2-10. `JM\Ch02\HelloGui\Banner.java`

Copyright © 2011 by Skylight Publishing

2.7 Summary

In the modern development environment, programmers usually write programs in one of the *high-level programming languages* such as C++, Python, or Java. A program written in a high-level language obeys the very precise syntax rules of that language and must also follow stylistic conventions established among professionals. For compiled languages, such as C or C++, a software program called the *compiler* translates the source code for a program from the high-level language into machine code for a particular CPU. A compiler creates object modules that are eventually linked into an executable program. Alternatively, instead of compiling, a program in a high-level language, such as Python, can be interpreted by a software tool called an *interpreter*. An interpreter does not generate an executable program but instead executes the appropriate CPU instructions immediately.

Java takes a mixed compiler + interpreter approach: the source code is compiled into code (called *bytecode*) for the *Java Virtual Machine* (*JVM*). JVM is not a real computer; it is an abstract model of a computer with features typical for different computer models. Bytecode is still independent of a particular CPU, but is much closer to a machine language and easier to interpret than the source code. A Java interpreter installed on a specific computer then interprets the bytecode and executes the instructions appropriate for that specific CPU.

An *IDE* (*Integrated Development Environment*) combines many tools, including an editor, a compiler, and a debugger, under one convenient *GUI* (*Graphical User Interface*).

The software development profession has evolved from an individual artisan craft into a highly structured engineering discipline with its own methodology, professional tools, conventions, and code of ethics. Modern applications are built in part out of standard reusable components from available packages. Programmers strive to produce and document new reusable components that meet the reliability, performance, and style requirements of their organization.

One can think of an *OOP* (*Object-Oriented Programming*) application as a virtual world of active objects. Each object holds its own memory and has a set of *methods* that can process messages of certain types, send messages to other objects, and create new objects. A programmer creates an OOP application by defining classes of objects. OOP is widely believed to lower software development costs, help coordinate team projects, and facilitate software reuse.

Copyright © 2011 by Skylight Publishing

Exercises

Sections 2.1-2.3

1. Which of the following are the advantages of using a high-level programming language, as opposed to a machine language? Mark true or false:

(a) It is easier to write programs. ______
(b) It is easier to read and understand programs. ______
(c) Programs run more efficiently. ______ ✓
(d) Programs can be ported more easily from one hardware platform to another. ______

2. Name four commonly used programming languages besides Java.

3. Mark true or false and explain:

(a) The operating system compiles source files into bytecode or executable programs. _____

(b) Each modern computer system is equipped with a compiler. ______ ✓

4. (MC) Which program helps programmers enter and modify source code?

A. Editor B. Compiler C. Linker D. Interpreter
E. None of the above

5. (MC) What is a debugger used for?

A. Removing comments from the source code
B. Running and tracing programs in a controlled way
C. Running diagnostics of hardware components
D. Removing syntax errors from Java programs
E. Removing dust from the computer screen

6. True or false: a modern IDE provides a GUI front end for an editor, compiler, debugger, and other software development tools. _____ ✓

Copyright © 2011 by Skylight Publishing

7. Describe the differences between a compiler, a JIT compiler, and an interpreter.

Section 2.4

8. (a) Replace the forward slash in the first line of the `HelloWorld` program with a backslash. Compile your program and observe the result.

(b) Remove the first three lines altogether. Compile and run your program. What is the purpose of the `/*` and `*/` markers in Java programs?

9. Write a program that generates the following output: ✓

```
   xxxxx
  x     x
((  o o  ))
  |  V  |
  | === |
   -----
```

10. Navigate your browser to Oracle's Java *API* (*Application Programming Interface*) documentation web site (`http://download.oracle.com/javase/6/docs/api/index.html`) or, if you have the JDK documentation installed on your computer, open the file <*`JDK base folder`*>`/docs/api/index.html` (for example, `C:/Program Files/Java/jdk1.6.0_21/docs/api/index.html`).

Find the description of the `Color` class. What color constants (`Color.RED`, `Color.BLUE`, etc.) are defined in that class? ✓

Copyright © 2011 by Skylight Publishing

11.■ (a) Write a program that prompts the user to enter an integer and displays the entered value times two as follows:

```
Enter an integer: 5
2 * 5 = 10
```

⋛ Hint: You'll need to place

```
import java.util.Scanner;
```

at the top of your program. The `Scanner` class has a method `nextInt` that reads an integer from the keyboard. For example:

```
    Scanner keyboard = new Scanner(System.in);
    ...
    int n = keyboard.nextInt();
```

Use

```
    System.out.println("2 * " + n + " = " + (n + n));
```

to display the result. ⋛

(b) Remove the parentheses around `n + n` and test the program again. How does the + operator work for text strings and for numbers? ✓

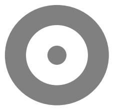
Copyright © 2011 by Skylight Publishing

Sections 2.5-2.7

12. Name the two concepts that are central to object-oriented programming.

13. (a) The program *Red Cross* (`JM\Ch02\Exercises\RedCross.java`) is supposed to display a red cross on a white background. However, it has a bug. Find and fix the bug.

(b)■ Using `RedCross.java` as a prototype, write a program that displays



in the middle of the window. ⪡ Hint: the `Graphics` class has a method `fillOval`; its parameters are the same as in the `drawRect` method for an oval inscribed into the rectangle. ⪢

14.■ Modify *HelloApplet* (`JM\Ch02\HelloGui\HelloApplet.java`) to show a white message on a blue background. ⪡ Hint: `Graphics` has a method `fillRect` that is similar to `drawRect`, but it draws a "solid" rectangle, filled with color, not just an outline. ⪢ ✓

15.■ Modify the *Banner* applet (`JM\Ch02\HelloGui\Banner.java`) to show a solid black box moving from right to left across the applet's window.

Copyright © 2011 by Skylight Publishing

16.◆ Using the *Banner* applet (`JM\Ch02\HelloGui\Banner.java`) as a prototype, write an applet that emulates a banner ad: it should display a message alternating “East or West” and “Java is Best” every 2 seconds.

⋠ Hints: At the top of your class, define a variable that keeps track of which message is to be displayed. For example:

```
    private int msgID = 1;
```

In the method that processes the timer events, toggle `msgID` between 1 and `-`1:

```
      msgID = -msgID;
```

Don’t forget to call `repaint`.

In the method that draws the text, obtain the coordinates for placing the message:

```
      Container c = getContentPane();
      int xPos = c.getWidth() / 2 - 30;
      int yPos = c.getHeight() / 2;
```

Then use a conditional statement to display the appropriate message:

```
      if (msgID == 1)
      {
        ...
      }
      else  // if msgID == -1
      {
        ...
      }
```
⋡

Copyright © 2011 by Skylight Publishing