Programming in Java™

Rex Jaeschke

Programming in Java

Edition: 4.0 (matches JDK1.6/Java 2)

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

Welcome to the world of Java. Throughout this book, we will look at the statements and constructs of the Java programming language as defined by the definitive text "The Java Language Specification", 3rd edition, by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, ISBN 0-321-24678-0, Addison-Wesley, 2005. Each statement and construct will be introduced by example with corresponding explanations, and except where errors are intentional, the examples will be complete programs or subroutines that are error-free. I encourage you to run, and modify these examples, because the only realistic way to master a language is to write programs in it, run them, and debug them.

All program examples have been taken directly from the manuscript text and tested using a number of compilers. While almost all material presented will be processor- and operating system-independent, system-specific aspects will be mentioned where applicable.

This book was written with teaching in mind and evolved from an earlier series on C and C++. It was written for use both in a classroom environment as well as for self-paced learning.

Java is a robust, general-purpose, high-level language that supports object-oriented programming (OOP). From a grammatical viewpoint, Java is a relatively simple language having only about 50 keywords. Because of the power of the basic statements and constructs, and the fact that the programmer can effectively extend the language by using callable methods while still maintaining portability, there really is no need for language extensions especially considering the fact that Java supports calls to procedures written in other languages. There are numerous commercial and shareware Java class libraries available, so when you look at other people's code, make sure you understand whether it involves classes outside those defined in the Java specification.

Depending on their language backgrounds, programmers new to Java may initially find programs hard to read because, traditionally, most code is written in lowercase. Lower- and uppercase letters are treated by the compiler as distinct. In fact, Java language keywords must be written in lowercase. Keywords are also reserved words.

If you have ever wondered what all those special punctuation marks on most keyboards are for, the answer may well be "to write Java programs!" Java uses almost all of them for one reason or another and sometimes combines them for yet other purposes. Java supports the 16-bit character set ISO 10646 UCS-2, commonly known as Unicode, of which ASCII is a proper subset. (Character codes 0–127 represent the same characters in both sets.) As such, it does not suffer from many of the difficulties faced by other languages that were designed primarily to support a "USA-English" mode of programming.

Java encourages a structured, modular approach to programming using classes containing callable subroutines, called *methods*. Classes can be compiled separately, with external references being resolved at runtime.

Java is an architecturally neutral language that when compiled, results in the generation of Java *bytecode*. This bytecode is then interpreted by a Java Virtual Machine (JVM), which may be running on a different hardware and/or operating system platform than the one on which the source was compiled.

Java lends itself to writing terse code. However, there is a fine (and subjective) line between writing code that is terse and code that is cryptic. It is easy to write code that is unreadable and, therefore, unmaintainable. In fact, that's what you will get by default. However, with care and a small dose of discipline, you can produce nicely formatted code that is easy to read and maintain. But remember, good code doesn't happen automatically—you have to work at it. Throughout the book, I will make numerous comments and suggestions regarding style. Perhaps the best advice I can give is "Remember that the poor fool who has to read your code in the future just might be you!" Above all, be overt and be consistent.

Numerous "Tips" and "Style Tips" have been added throughout to highlight important suggestions. Considerable attention has been paid to using and teaching a consistent, popular, and overt programming style. Both kinds of tips are highlighted so they stand out. For example:

> **Tip:** The simplest way to convert the value of an expression to its corresponding string representation is to concatenate that expression with an empty string; for example, `"" + 123` results in the string `"123"`.

> **Style Tip:** Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

Java is not all things to all people; nor does it claim to be. For many applications, assembly language, Pascal, COBOL, FORTRAN, and BASIC, for example, will do just fine. In any event, compared to other high-level languages, Java is a relatively inexpensive language to learn and master. It is certainly much less expensive to master—and far less error-prone—than C or C++.

Java's Design Goals

The Java language, library, and run-time environment were designed to deliver the following:

- Code reuse and reduced development effort via object-oriented programming support.
- Extremely broad portability by the elimination of implementation-defined and unspecified behavior, by the use of an architecturally neutral language interpreted by a host-specific virtual machine.
- The specification of a standard GUI library.
- A comprehensive library including support for networking, Internet access, audio, database, and data structures.
- Strong type checking.
- Support for threading.
- No unsafe constructs.
- Easier to learn, read, and write.
- Support for internationalization.

Reader Assumptions

I assume that you know how to use your particular text editor, Java compiler, and debugger. Comments on the use of these utility programs will be limited to points of particular interest to the Java programmer.

To fully understand and exploit the material, you should be conversant with the following topics:

- The basic purpose of a compiler and linker.
- Number system theory.
- Bit operations such as AND, inclusive-OR, exclusive-OR, complement, and left- and right-shift.
- Data representation.
- Communication between procedures by passing arguments and/or by returning a value, as well as via global variables.
- Use of single- and multi-dimensional arrays.
- Creation and use of sequential files and how to do formatted and unformatted I/O.
- Basic data structures such as linked lists.

Many of Java's more powerful capabilities, particularly exception handling, inheritance, and threads, require advanced programming experience before they can be understood and exploited fully.

Despite the many similarities between Java and C/C++, you do *not* need to know either of these languages to use this book. If you do know either or both, simply read quickly over those sections that seem familiar to you. I say, "read quickly" rather than "skip" because you may well find that Java defines things more fully or a little differently than do C and C++. For example, you might already "know all about the types `int` and `long`", but if you skip the coverage on that topic, you'll miss learning that their actual size and representation is defined by Java, unlike C/C++, in which these things are implementation-defined.

So if you already know C++, the Java learning curve should be short and not too steep. If you know C but not C++, you have all the OO stuff to learn as well. If you know some OO language other than C++, you'll already be familiar with the OO concepts but not the syntax that is largely common to Java, C, and C++. And if your programming background is in some procedural language such as Pascal, FORTRAN, or BASIC, you'll need to read the whole book closely and do all the exercises. As for those of you having only a COBOL background, you'll have some catching-up to do with respect to thinking in binary versus decimal when it comes to data representation, as well as the use of methods and their associated argument passing and value returning.

Limitations

This book covers almost all of the Java language. It also introduces the library of *core classes*. However, only a few percent of these classes are covered in detail. Another few percent are mentioned in passing. The Java library contains so many methods that books have been written about that subject alone.

For most of its early life, Java was used to implement *applets*, programs that execute under the control of a *browser*, and that often are resident on a remote site on the world wide web. Later, implementations became available that allow us to write stand-alone programs called *applications*. This book makes almost no mention of applets, browsers, or programming for the web—that is another whole series of Java-related topics. Instead, this book is directed at teaching the Java language proper, by writing applications, since without a thorough knowledge of the language, you will not be able to understand and implement other than very trivial applets.

Applets, GUI, calling non-Java routines, threading, interprocess communications, and a number of other advanced features are covered in a separate text.

Source Code, Exercises, and Solutions

The programs shown in the text are made available electronically in a directory tree named Source, where each chapter—as well as each example within that chapter—has its own subdirectory.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided on disk in a directory tree named Labs, in which each chapter has its own subdirectory.[1] Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Numerous exercises contain a statement of the form "(See directory *xx*.)". This indicates the corresponding solution or test file in the Labs subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

Refer to Annex A for a discussion of compilation and execution of Java applications.

The Java Development Kit

Sun's initial production release of Java was the Java Development Kit (JDK) version 1.0. Versions 1.1 through 1.5 contained numerous bug fixes, and language and library enhancements. The latest version can be downloaded from Sun's website (www.sun.com).

While the language has remained very stable, along the way, several features were added along with numerous new packages and classes and new methods to existing classes. Also, some existing method names have been changed. In these latter cases, the old names continue to be acceptable, but are flagged by the compiler as *deprecated*, meaning that support for them might well be removed in future versions. If your compiler issues such a warning, consult Sun's on-line documentation to find the recommended replacement.

From an internationalization (I18N) viewpoint, one of the most significant additions made by V1.1 was the completion of support for dealing with non-US, non-English environments, including those involving very large alphabets and non-Latin writing systems.

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my introductory Java seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, September *2009*

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

1. The Basics

In this chapter, we will learn about a number of fundamental constructs and language elements. The main topics covered include basic program structure, user-defined names, basic formatted output, primitive data types, literals, operator precedence, and type conversion.

1.1 Basic Program Structure

A correct source program is made up of an acceptable combination of tokens, where a *token* is the smallest possible unit of source. Java has five different kinds of tokens: identifiers, keywords, literals, separators, and operators.

For the most part, Java is a free-format language, and space between tokens is optional. However, in some cases, something is needed between tokens so they can be recognized the way they were intended. White space performs this function. *White space* consists of one or more consecutive characters from the set: space, horizontal tab, form feed, *newline* (the ASCII lf character), and return (the ASCII cr character). Typically, the return and/or newline characters are entered by pressing the RETURN or ENTER key. An arbitrary amount of white space is permitted before the first token, between any two adjacent tokens, or after the last token.

Let's look at the basic structure of a Java program (see directory Ba01):

```
/*–
Ba01.java - A sample Java program

In case 1 below, the text "Welcome to Java" is written to the
standard output device.

In case 2, the main method returns to its caller.
–*/

public class Ba01
{
        public static void main(String[] args)  // start of program
        {                                       // start of block
/*1*/           System.out.println("Welcome to Java");
/*2*/           return;                         // redundant
        }                                       // end of block
}
```

The output produced is

```
Welcome to Java
```

There are several ways to write a comment. The first form involves both a comment start and end delimiter, represented by /* and */, respectively. These delimiters and all the characters contained between them are ignored. This form of comment can span an arbitrary number of source lines.

Note the unusual comments, /*1*/ and /*2*/, in the example above. Throughout this book, such comments are used to give source lines pseudo-labels, so they can be referenced directly in the narrative. In production code, this approach can also be used, as follows: A comment prior to a method provides a general introduction and describes the steps used to implement the solution. By giving each step a label, we can place a comment containing that label prior to the first statement that implements that step. For example, in the program above, the introductory comment mentions case 1 (which corresponds to step 1), and the comment /*1*/ shows where that step (or case) is implemented.

The second form is a line-oriented comment, begun by // and terminated by the end of that source line. Both these forms of comment are treated as a single space allowing a comment to occur anywhere white space can be used; that is, between any two adjacent tokens. Comments of the same form do not nest; however, a comment of the form /* ... */ can contain // as text, and vice-versa. A third form of comment (not shown here) is known as a *documentation comment*.[1]

A Java program consists of one or more *methods* that can be defined in one or more source files (or *compilation units*, as they are formally known). (A method corresponds to what some other languages refer to as a function, a procedure, or a subroutine.) A program must contain at least one method, called `main`. This specially named method indicates where the program is to begin execution.

Each method can have one or more *modifiers*; `main` has two: `public` and `static`. Names having the `public` modifier are visible outside their parent class. We'll learn about the `static` modifier in §1. The keyword `void` preceding the method name `main` indicates that this method does not return a value. Return values are discussed in §3.1. For now, suffice it to say that the definition of `main` must always contain these three keywords, as shown.

The parentheses following the method name `main` surround that method's formal parameter list.[2] The parentheses are required, even if no arguments are expected. Although the argument passed to `main` is not used in this example, it must be declared as shown. The meaning and purpose of this argument are discussed in §4.6.

The body of a method is enclosed within a pair of matching braces. All executable code resides within the body of some method or other.[3] Statements are executed in sequential order unless branching or looping statements dictate otherwise.

A program terminates when it returns from `main`, either by dropping into the closing brace of that method–—which acts as an implicit `return` `return`:implicit statement——or via an explicit `return` statement, as shown in case 2 above.[4]

---

[1] Documentation comments have the form /** ... */ and contain tags of the form *@name*, as well as text. These tags and the program components they precede are recognized by the javadoc program, which uses them to produce program documentation as a series of HTML files.

[2] A method uses parameters to declare what it is expecting to be passed, via an argument list, at run time.

[3] We'll see two exceptions to this requirement in §4.11 and §5.9.

[4] A program can also be terminated from within any method by an explicit call to the library method `System.exit`.

As we can see, `main` is defined inside a block preceded by `class` *class-name*, where the name of the class is `Ba01`, a somewhat arbitrary choice.[1] Note that the class is declared as `public`. While this is permitted, it is not necessary. We'll discuss this further in §8.2; however, for now, suffice it to say that if you declare a class to be public, it must be contained in a source file whose name is exactly the same as that class, case included.

Every method must be defined inside some class or another. While a *class* has numerous attributes, the one of importance here is namespace control. The name `main` is really qualified by its parent class, in this case `Ba01`. For that reason, when we wish to run this program, we must specify the name of the class that contains the `main` method of interest, since each class can have such a method.

The call to method `println` in case 1 above, writes a line of text to the standard output device, automatically adding a newline. As we can see, a string literal is delimited by double-quote characters. Note that we cannot call `println` using that simple name; instead, we must specify that this method belongs to an object called `out`, which, in turn, belongs to a class called `System`. We separate each of these names with the dot (.) member selection operator. (Formatted I/O is discussed in §1.4.)

> **Style Tip:** Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

We can write any given correct program in an infinite number of ways simply by using different amounts and kinds of white space (including comments). There are very good, overt styles and very bad, cryptic styles. Then there is the large and subjective gray area in between.

> **Style Tip:** Be overt; pick a style that isn't too far outside the mainstream, and stick with it. Above all, be consistent. And remember, the style you develop when learning a language is the one with which you will likely stay, so give some thought to programming style from the outset.

The style used in this book is very common. If you choose to adopt a different style, note that no matter how great you think your particular style is, every minute you spend arguing about whose style is superior and why—or worse yet, changing other people's code to your style—is generally time wasted. If the people on a multi-person project cannot agree on a common style, the project manager should dictate one.

> **Exercise 1-1:** What happens if you try to run a program that has no `main` method? Try omitting `main`'s argument list and/or one or more of the modifiers.
>
> **Exercise 1-2:** Using `/*` and `*/`, comment out a block of code that already contains comments of that type, and see how your compiler reacts. What happens if you leave off the closing `*/` from a comment?

[1] Actually, this name was picked because this example is the first in the chapter entitled "Basics".

4. References, Strings, and Arrays

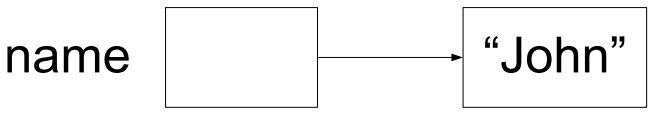
In this chapter, we will learn about reference variables, the allocation of memory at runtime using `new`, garbage collection, array allocation and manipulation, and the string classes `String` and `StringBuffer`.

4.1 Introduction

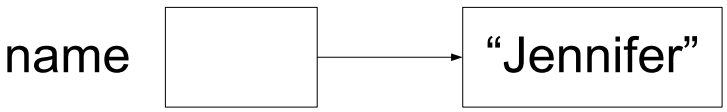
There are two categories of types in Java: primitive and reference. We have already seen examples of the primitive types `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`.

In the definition `int i;`, `i` is a *variable* and the name `i` designates some specific storage location in memory. We say that the value of `i` is some `int`. Throughout its life, that variable is always associated with the same location. (This may seem obvious based on your experience with other languages but it's most important to understand when we look at reference types.) To help us understand reference types, let's look at the following example (see directory Rf01):

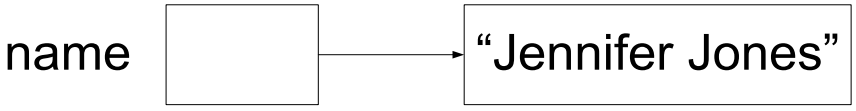
```
public class Rf01
{
        public static void main(String[] args)
        {
/*1*/           String name = "John";
/*2*/           System.out.println("name is " + name);
```



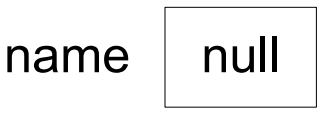

```
/*3*/           name = "Jennifer";
                System.out.println("name is " + name);
```




```
/*4*/           name = "Jennifer" + " Jones";
                System.out.println("name is " + name);
```




```
/*5*/           name = null;
/*6*/           System.out.println("name is " + name);
        }
}
```



The output produced is:

```
name is John
name is Jennifer
name is Jennifer Jones
name is null
```

`String` is an object type defined in the Java library, and a string literal refers to an object of this type. An object type is often called a *class type*. Unlike some languages, a string is not an array of `byte` nor is it an array of `char`; it is a sequence of Unicode characters. **The contents of an object of type String cannot be modified.**[1]

In case 1, we define a variable called `name`. At first glance, we might think that `name` is a variable of type `String` and that it contains the value `"John"` directly; however, that is not the case. Instead, `name` is a reference to that string; that is, `name` does not actually contain that data, it simply points to that data, which resides elsewhere. (In the case of a string literal, the data representing that string really has no name.) We say that `name` is a *reference variable* and that its type is "reference to `String`".

We have seen that the + operator can concatenate string literals and, since a string literal is really a string, that operator really concatenates strings. However, when doing so, it cannot simply add the second string to the end of the first (remember, strings are read-only), so in case 2, it creates a new string with the concatenated value. This new string is then displayed.

In case 3, `name` is made to refer to a different string. The important thing to understand here is that `name` "has been made to point" to a different string; the value of the string `"Jennifer"` has *not* been copied anywhere.

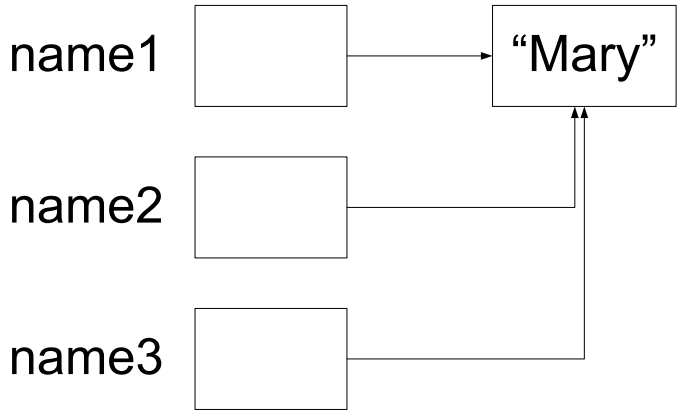
In case 4, the two strings are concatenated resulting in a third string to which `name` is made to point. Because a string's value is read-only, the second string cannot simply be appended to the end of the first one.

A primitive variable can only ever contain a value of its type. However, a reference variable can hold either a reference to an object that is assignment compatible with the type of that variable, or the *null reference*. Case 5 introduces the name `null`. While this looks like a keyword, it really is referred to as the null literal; in any event, the word `null` is reserved. By initializing a reference with the value `null`, we make it point "nowhere". The output produced by case 6 shows us that when the value of a reference containing `null` is used in string concatenation, the text "null" results.

There can be any number of references to the same object. For example (see directory Rf02):

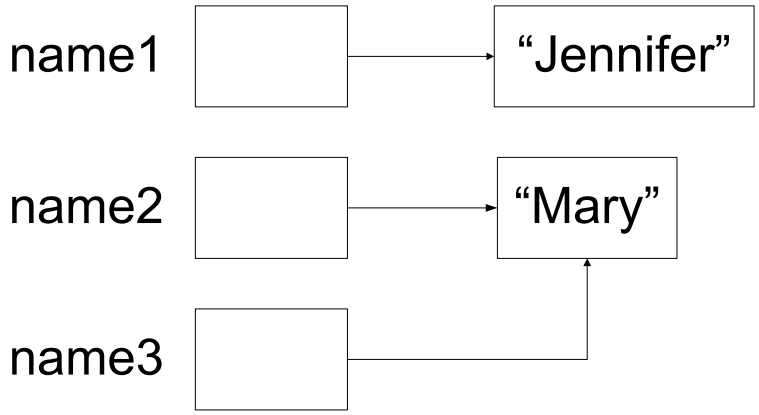
[1] The library class `StringBuffer` is similar to String except that objects of type StringBuffer can have their contents changed. See §4.10.


```
public class Rf02
{
        public static void main(String[] args)
        {
                String name1 = "Mary";
/*1*/           String name2 = name1;
/*2*/           String name3 = name2;
```




```
                System.out.println("name1 is " + name1);
                System.out.println("name2 is " + name2);
                System.out.println("name3 is " + name3);

/*3*/           name1 = "Jennifer";
```




```
                System.out.println("\nname1 is " + name1);
                System.out.println("name2 is " + name2);
                System.out.println("name3 is " + name3);
        }
}
```

The output produced is:

5. Classes

Classes are the key foundation stone of object-oriented programming. By using classes, we can take advantage of encapsulation, data hiding, inheritance, and polymorphism, the first two of which are discussed in this chapter.

5.1 Introduction

Until now, we have used the keyword `class` primarily as a means for namespace control; however, now we'll see how to exploit its full potential. We'll do this by showing how we can define our own object types.

So far, we've been creating variables of primitive types, arrays of those types, and strings. And while we can do useful programming with these types, eventually we'll need more complex and sophisticated data types. For example, in a personnel system, we might want an Employee type, containing employee name, address, date of birth, veteran status, and other descriptive information. In a library catalog system, we'll probably want to keep track of each book's call number, author, title, year of publication, and the like, in some kind of a Publication type. Neither of these types can be accommodated directly by the types we have seen.

Throughout this and future sections, we'll use `Point` as our user-defined type. A Point represents a location in a two-dimensional plane. This simple type is something with which we can easily relate and it serves nicely to introduce the class support machinery. Even if you don't write graphics programs, it should be very easy to extrapolate from the principles learned here.

Let's define our new Point type, and create and manipulate a number of objects of that type (see directory Cl01):

```
public class Point
{
// instance variables

/*1*/   public int x;
/*2*/   public int y;
}
```

The first thing to notice is that in cases 1 and 2, the keyword `static` has been omitted. Back in §1.15, we learned that variables defined inside a class (rather than inside a method) and having the modifier `static`, are class variables; that is, they are variables belonging to the class as a whole. What we now have is a pair of *instance variables*, called `x` and `y`. We can create an arbitrary number of Point objects each of which contains an x- and y-coordinate pair; that is, each instance of class Point contains a unique pair of instance variables, since each Point's representation is separate from those of all other Points, even for Points that happen to have the same coordinate values.

```
public class Cl01
{
        public static void main(String[] args)
        {
/*3*/           Point p1 = new Point();
/*4*/           System.out.println("p1 = (" + p1.x + "," + p1.y + ")");

/*5*/           p1.x = 5;
/*6*/           p1.y = 7;
                System.out.println("p1 = (" + p1.x + "," + p1.y + ")");

/*7*/           Point p2 = new Point();

/*8*/           p2.x += -4;
/*9*/           p2.y += 12;
                System.out.println("p2 = (" + p2.x + "," + p2.y + ")");
        }
}
```

In case 3, we allocate memory for a Point using new, just like we did for strings in §4.1. Creating an instance of a class is known as *instantiation*. By default, all instance variables take on a zero, false, or null value, depending on their type. Like strings and Points, all objects of user-defined types must be allocated on the heap using new. To access the instance variables, we simply prefix their names with the name of their parent, as in case 4. Note that this is different to the way in which we have been accessing class variables, which use the parent class name as their prefix. A class can have both class and instance variables, as we'll see later; in fact, many of the library classes do.

It is useful to be able to change the coordinates of a Point after it has been created. This operation is often referred to as *moving* the Point, and is done in cases 5 and 6. In case 7, we define a second Point and we *translate* that Point in cases 8 and 9. (Translation involves moving by an offset rather than to an absolute new location.)

The output produced by this program is:

```
p1 = (0,0)
p1 = (5,7)
p2 = (-4,12)
```

When we have a general-purpose class such as `Point`, we very quickly realize that it makes no sense to define our application program as a method within that class. After all, we likely will write many programs that use this class and we can't define all our programs inside that class. In any event, we can only have one method called `main` having some given signature. As a result, not only do we need to define our application and Point in different classes, these classes need to be in separate source files.

When defining user-defined types, it is customary to precede the keyword `class` with the keyword `public`. While this is permitted, it is not necessary. We'll discuss this further in §8.2; however, for now, suffice it to say

that if you declare a class to be public, it must be contained in a source file whose name is exactly the same as that class, case included.

5.2 Class-Specific Methods

It is tedious to write out the steps to move, translate, and display a Point each time. Also, every application relies on the fact that a Point contains an x- and a y-coordinate of type `int`, called `x` and `y`, respectively. Therefore, any change in the way that type is represented will negatively affect those applications. By making the instance variable private and adding some public methods to class `Point`, we can deal with it in a more abstract manner. Now the Point class looks like the following (See directory Cl04):

```
public class Point
{
// instance variables
        private int x;
        private int y;

// constructors
        public Point(int xor, int yor)
        {
                x = xor;
                y = yor;
        }

        public Point()
        {
                x = 0;
                y = 0;
        }
```

By making the instance variables `private` they can only be accessed by methods within their parent class. This is known as *data hiding*, since we hide the representation details of `Point` from all programs that use it.

The main reason for data hiding is to cater for unanticipated changes. Since we can never say the representation of an object won't change and we can never say exactly how it might or will change, we should cater for as much flexibility as possible. That is, assume everything within reason will change.

In the program above, we also see an example of *encapsulation*, the process by which we associate variables and methods by defining them in the same class. This allows us to control the access to those variables and methods.

The methods called `Point` are special. Strictly speaking, they are not really methods but rather, constructors; however, for all practical purposes, they look like methods—they just have a special name. From this, we can deduce that, syntactically, a *constructor* is nothing more than a method that has the same name as its parent class. The first constructor expects two `int` arguments, which represent the x- and y-coordinates, respectively, and it initializes the private fields using those values. A constructor cannot have a return type declared, not even `void`. It can contain one or more `return` statements, however, provided they do not contain an expression.

