Programming in Java™

Rex Jaeschke

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", by James Gosling, et al. 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. 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, other languages 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.
- Easy 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.
- Number system theory (such as binary and hexadecimal).
- 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.
- 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 absolutely 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 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 small number of these classes are covered in detail. Others 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 allowed programmers 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 separate texts.

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 C for a discussion of compilation and execution of Java applications.

The Java Development Kit

The 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. Over the years, the numbering system changed, with 1.6 being known as Java 6. Then came editions 7, 8, 9, 10, and 11. The latest version can be downloaded from Oracle's website.

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 the JDK 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, February 2019

[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 begins 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 §5. 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 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` statement—or via an explicit `return` statement, as shown in case 2 above.[4]

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.[5] 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

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

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?

Let's move on to the more common case of a program having multiple methods (see directory Ba03a):

```
public class Ba03a
{
        public static void main(String[] args)
        {
/*1*/           welcome();
        }
        private static void welcome()
        {
                System.out.println("Welcome to Java");
        }
}
```

Java supports modularization via methods. Unlike some languages, in Java, method definitions cannot be nested; that is, each method's definition must be outside the braces delimiting the definitions of all other methods. There is no syntactic difference between `main` and any other Java method; they all have the same basic structure.

We call a method by using its name followed by a possibly empty argument list, as shown in case 1. The parentheses used in the call represent the method invocation operator. Each statement must be terminated by a semicolon separator. When method `welcome` terminates, control is returned to its caller; we learned earlier that dropping into the closing brace of a method is an implicit `return`.

Like `main`, `welcome` does not return any value, so it too is declared using `void`. `welcome` must also have the `static` modifier. Since this method need be visible only within its parent class, it has the modifier `private`.

Style Tip: Indent every line inside the body of a method definition by at least one tab. Also, line up the opening and closing braces, one above the other, but not indented with the block they delimit. This way, you can see the shape of the program at a glance by looking at the brace pairs, and if you are interested in the details, you can quite easily see where they are.

Exercise 1-3: See how your compiler reacts to having one method defined inside another. (This can easily happen if you forget the closing brace from the first method, and if you don't line up your brace pairs, you can't easily see which one is missing.)

Exercise 1-4: What happens when you omit the semicolon from the end of a statement? Add an extra one and see what happens.

Exercise 1-5: What happens if the keyword `static` is omitted from the definition of `welcome`?

Within a class, methods can be defined in any order; for example, the program Ba03b (which is not shown here) defines `welcome` before `main`. Let's see how we can define each method in a separate class (see directory Ba03c):

```
class Ba03c1
{
        public static void main(String[] args)
        {
/*1*/           Ba03c2.welcome();
        }
}
```

```
class Ba03c2
{
        public static void welcome()
        {
                System.out.println("Welcome to Java");
        }
}
```

Note that the call to `welcome` in case 1 has been changed to `Ba03c2.welcome`. As we learned earlier, names defined inside a class belong to that class, allowing different classes to contain methods by the same name. Since `welcome` is defined in class `Ba03c2`, its parent class name is added to the method call. Note that `welcome` must be `public` if it is to be accessed outside its parent class.

As we might expect, the classes can be defined in separate source files (see directory Ba03d):

```
// source file Ba03d1.java

public class Ba03d1
{
        public static void main(String[] args)
        {
                Ba03d2.welcome();
        }
}

// source file Ba03d2.java

public class Ba03d2
{
        public static void welcome()
        {
        }
}
```

All methods that belong to a given class must be defined in the same source file. There is no way to have contributions to a single class from multiple source files. Therefore, the use of class names as prefixes is especially important when working on large projects or when using the standard or class libraries. However, given their simplicity, most examples in this text involve only one source file with all methods being defined in the same class.

1.2 Java's Compilation Model

Java's compilation model is different to that used by other mainstream compiled languages, which, typically, generate a single object file from a source file, and where the name of the object file corresponds to that of its source file. The end result of compiling a single Java source file is one or more .class files whose names correspond exactly to the classes defined in that source file.

Consider the example above that involves the source files Ba03d1.java and Ba03d2.java. When Ba03d1.java is compiled, the file Ba03d1.class is produced, *not* because that's the source file name, but because the only class in that file is called Ba03d1 and the suffix .class was added. Similarly, when Ba03d2.java is compiled, the file Ba03d2.class is produced.

A source file can have any name, since that name is not used except by the compiler to locate the text to be compiled. For example, the file Ba03c.java contains both of the classes Ba03c1.java and Ba03c2.java, yet what comes out of the compiler are two class files, named Ba03c1.class and Ba03c2.class, respectively. So, the distribution of classes among source files is not particularly relevant.

With the addition of enum types (§1.7) by JDK1.5, each enum type also compiles to its own class file.

To execute a Java application, we must tell the Java runtime system the name of the class to load, that contains the `main` method at which we wish to begin executing the program. This allows us to have more than one `main` method in a program, provided each is defined in a different class.

For a more detailed discussion of this topic, refer to Annex C.

1.3 Identifiers

One kind of token is an *identifier*, a name usually invented by the application programmer. Identifiers are used to name variables, methods, and classes, among other things. Since keywords (such as `public`, `static`, `void`, and `return`) are reserved, they cannot be used as identifiers.

An identifier can be spelled using the following characters: Upper- and lowercase letters, the decimal digits 0–9, and the underscore (_);[1] however, an identifier cannot begin with a digit. An identifier can be arbitrarily long, all characters are significant, and case is distinct.

While we can spell identifiers using letters in either case, or combinations of upper- and lowercase, the recommended style of naming variables and methods is to use all lowercase characters with the first letter in the second and subsequent words being uppercase. Examples are `total`, `maximumNumber`, and `computeTotalCost`. For class names, use uppercase leading characters on all words with no inter-word separator. Examples from the core class library are `CheckboxGroup`, `Choice`, and `BorderLayout`. Identifiers spelled in all uppercase (with _ as the inter-word separator) are often used for other purposes, as we shall see in §1.10. For a detailed discussion of identifier naming conventions, see "Naming Conventions" in the book "The Java Language Specification" mentioned in the Preface.

Keywords are only recognized as such if they are spelled entirely in lowercase. The complete list of keywords is shown in Annex B.

Style Tip: It is very bad style to invent an identifier such as `If`, `Else`, or `FOR`, since such names are too easily confused by the reader (but never by the compiler) with keywords having the same names but spelled in lowercase.

Exercise 1-6*: Which of the following are valid identifiers : name, _Xyz_, _, Total.count, Today'sdate, first-name, day_of_week, X32BITS, TOTAL$COST, 3WiseMen, LastNameOfMyFathersGrandfatherInLaw? (See lab directory Lbba01.)

[1] The dollar sign $ is also permitted but its use is discouraged. Symbols from non-Roman alphabets and non-Arabic number systems are also permitted; however, that is beyond of the scope of this text.
Starting with Java 9, a name consisting of a single underscore changed from being an identifier to be a keyword.

1.4 Introduction to Formatted Output

Unlike many older languages, Java has no input or output statements. Instead, a set of I/O methods is provided as part of the standard library. Two such library methods are `PrintStream:print` and `PrintStream:println`. These methods write formatted output to the *standard output* device, which, typically, is directed to the user's terminal. For example (see directory Ba04):

```
public class Ba04
{
        public static void main(String[] args)
        {
/*1*/           System.out.print("Hello.\n");
/*2*/           System.out.print("\nWelcome");
/*3*/           System.out.print(" to Java.\n");

/*4*/           System.out.print("Hello.\n\nWelcome to Java.\n");
/*5*/           System.out.println("Hello.\n\nWelcome to Java.");
/*6*/           System.out.println("Three "
                        + "separate "
                        + "words");
        }
}
```

which produces the following output:

```
Hello.

Welcome to Java.
Hello.

Welcome to Java.
Hello.

Welcome to Java.
Three separate words
```

In all six calls above, the output method name is prefixed with `System:out`. As we learned earlier, a method name is unique within its parent class and since the `print` and `println` methods operate on the public variable `out` defined in class `System`, we must qualify these method names as shown. We will learn more about `System` and `out` in §9.

In case 1, `print` is called with one argument, a *string literal*. (A string literal has type `String`, a type defined in the standard library. We'll learn more about this type in §4.1.) The characters enclosed in double quotes are written out verbatim, except for certain *escape sequences*, which begin with a backslash. \n is Java's notation for a *newline*, the character that moves the print position to the first character on the next output line.[1] A newline is not automatically appended by `print`, so `print` can be invoked multiple times to print an output line a piece at

[1] While a newline may actually generate more than one character (such as a carriage return/line feed pair) on output, within a Java program it is considered a single character.

a time, as shown in cases 2 and 3. `print` can also output multiple lines at a time. If we want double spacing, we must use two consecutive newlines, as shown in cases 4 and 5.

Calling `print` three times to put out three parts of some output seems rather inefficient and it is. Case 4 achieves the same thing with only one such call. Unlike `print`, `println` appends a newline to the string being written; in all other respects, these two methods are identical.

In cases where we have long string literals, or our style requires considerable indenting, it may be necessary or convenient to break a string literal over multiple lines. Now while we can put an arbitrary amount of white space between two adjacent tokens, we cannot split a single token, such as a string literal. To handle this case, we can break the string literal into a number of smaller strings, separating them with white space as desired as well as with the string concatenation operator +, as shown in case 6.

Inside string literals, the backslash is used as an escape character prefix. It indicates that the following one or more characters are to be interpreted with other than their literal meaning. Each escape sequence represents a single character (and often one that is non-printable). The complete set of escape sequences is:

Table 1-1: Escape Sequences

Sequence	Value	Meaning
`\b`	0x0008	Backspace
`\f`	0x000C	Form feed
`\n`	0x000A	Line-feed
`\r`	0x000D	Carriage return
`\t`	0x0009	Horizontal tab
`\\`	0x005C	Backslash
`\'`	0x0027	Single quote
`\"`	0x0022	Double quote
`\`*ooo*	0x0000–00FF	Octal bit pattern
`\u`*hhhh*‡	0x0000–FFFF	Unicode character in hex

‡ Strictly speaking, this is a *Unicode sequence*; not only can it be used inside character and string literals, it can be used anywhere in a source file to designate the corresponding Unicode character. Note that *hhhh* is exactly four hex digits long having leading zeroes as necessary.

For the most part, the escape sequences are self-explanatory; however, a few need further explanation. Carriage return is a holdover from the early days of C. It is unlikely it will ever be needed in mainstream programming; a newline will always suffice as a line terminator. Sometimes it is necessary to escape a single quote, as we shall see in §1.6.2. To enclose a double quote in a string literal, we must use `\"`.

The octal bit pattern sequences allow special commands to be sent to output devices. For example, to clear the screen of a terminal that understands ANSI hardware escape sequences, we print the character sequence `\33[2J`, where `\33` represents the ESCAPE character. Likewise, we can use escape sequences to set a dot-matrix printer to bold, italic, or underline mode, for example. An octal escape sequence can contain between one and three octal digits.

While a hexadecimal escape sequence can perform the same task as an octal escape sequence, it can represent a much larger range of values. In fact, it can represent any character in the Unicode character set. For example, \u03A0 and \u03C0 represent the Greek letters Π and π, respectively, while \u2020 and \u2021 represent † and ‡. A hexadecimal escape sequence must contain exactly four (upper-, lower-, or mixed-case) hexadecimal digits.

Both `print` and `println` can output the value of one or more variables as well as text and escape sequences, as the next example demonstrates (see directory Ba05):

```
public class Ba05
{
        public static void main(String[] args)
        {
/*1*/           int year;

/*2*/           year = 1992;
/*3*/           System.out.println("Year " + year + " was a leap year.");

/*4*/           int i = 10, j = i + i;

/*5*/           System.out.println(i + " + " + i + " = " + j);
        }
}
```

The output produced is:

```
Year 1992 was a leap year.
10 + 10 = 20
```

In case 1, the variable `year` is defined to be of type `int`, a signed integer type that can represent a range of values that includes 1992. (We will discuss the range of each integer type in §1.5.2.)

Variables must be declared explicitly before their first use. Unlike some languages, when an unknown variable name is used, there is no implicit creation or typing; it is diagnosed as an error. Like statements, each definition must be terminated by a semicolon.

We assign values to variables using the assignment operator =, as shown in case 2. The value assigned can be a literal, the value of another variable, the result returned from a method, or the value of any expression having *compatible type*.[1] If necessary, the type of the result of the right-hand expression is converted to match that of the left-hand expression.

In case 3, `println` is called with an argument involving two string literals and a variable of type `int`. Whenever one of the operands of the binary + operator has type `String` and the other does not, that other one is implicitly

[1] All arithmetic types are compatible with one another.

converted to a string via an implicit call to a library method. As a result, the value of `year` is automatically formatted as a string and that string is concatenated with its predecessor and its successor before being printed. The string produced for the value of `year` contains decimal digits and uses only the number of print positions needed. A leading sign is present only if the value is negative.

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"`.

The definition of a variable can contain an *initializer* (such as `= 10` in case 4) thereby eliminating the need for a separate assignment statement. As shown, multiple variables can be defined at the same time, a variable's name can be used immediately after it has been defined, and the initializer of a variable can be an expression involving non-constant terms.

Provided a variable is defined before its first use, its definition can be interspersed with statements.

Style Tip: By placing each variable in its own separate definition, with one definition per source line, you leave room for a trailing comment. You can also cut and paste lines more easily in a full-screen text editor.

Initially, the basic output machinery did *not* provide direct support for controlling the format of output; however, the class `DecimalFormat` (and its siblings in package `java.text`, which was introduced in JDK1.1) provides a lot of formatting capability. Note that understanding and using this facility requires familiarity with most of the chapters in this book; that's why it is provided as an appendix. A somewhat simpler, alternate, approach to formatted output was added in JDK1.5. For more information on these facilities, see Annex D.

Exercise 1-7*: What happens if you try to use the value of a variable that hasn't been initialized? (See lab directory Lbba02.)

Exercise 1-8*: What happens if you use an undefined escape sequence such as `\x` or `\*`? (See lab directory Lbba03.)

1.5 The Primitive Data Types

The *primitive types* are those built-in to the compiler; they are the logical type `boolean` and the numeric types. The numeric types are further broken down into integer and floating-point types.

1.5.1 The boolean Type

Variables of type `boolean` can contain the value `true` or `false`. Strictly speaking, these two names are Boolean literals; however, they behave like keywords in that they are reserved. Here's an example involving this type (see directory Ba06):

```
public class Ba06
{
        public static void main(String[] args)
        {
                boolean complete = true;

                System.out.println(complete + ", " + false);
                complete = false;
        }
}
```

which produces the following output:

```
true, false
```

As we can see, the value of each expression of type `boolean` is converted to a string before being concatenated to the string(s) adjacent.

1.5.2 The Integer Types

All integer types except `char` are signed, use twos-complement representation, and have the following sizes and value ranges:

Table 1-2: The Integer Types

Type	Size	Value Range
`byte`	8 bits	-128 to +127
`short`	16 bits	-32,768 to +32,767
`int`	32 bits	-2,147,483,648 to +2,147,483,647
`long`	64 bits	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
`char`	16 bits	0x0000 to 0xFFFF (0 to 65,535)

Strictly speaking, there are no unsigned integer types although `char` really does represent an unsigned value.

Here's an example that uses the integer types (see directory Ba07):

```
public class Ba07
{
        public static void main(String[] args)
        {
                byte b = 127;
                short s = -56;
                int i = 2147483647;
                long l = 1234;
/*1*/           System.out.println(b);
/*2*/           System.out.println(s);
/*3*/           System.out.println(i);
                l = l * i;
/*4*/           System.out.println(l);

/*5*/           i = i + 1;
                System.out.println(i);

/*6*/           l = i = s = b = 100;

/*7*/           char c = 65;    // 'A' has value 65 in Unicode
                System.out.println(c);
        }
}
```

which produces the following output:

```
127
-56
2147483647
2649994820398
-2147483648
A
```

As we can see, `println` knows how to format and display values of the integer types. In case 5, we add 1 to `i` causing its value to overflow. However, rather than produce a runtime error or result in undefined behavior, Java requires that such operations "wrap around". For signed integer operations this means the value goes from the largest positive value to the smallest negative or, for subtraction, vice versa; for `char` operations, this means we go from the largest value to zero or, for subtraction, vice versa.

Since variables of type `byte`, `short`, `int`, and `long` represent increasingly larger value ranges, case 6 is perfectly well formed. That is, any value that can be stored in a `byte` can safely be stored in a `short`, and so on up to `long`.

In case 7, we are initializing a `char` with an integer value; and while this works just fine, it doesn't make the code particularly readable. In §1.6.2, we'll see how to write this more clearly using a character literal.

We can manipulate the bits in an integer value via a series of operators; for example, we can shift bits left or right using the binary operators <<, <<=, >>, >>=, >>>, and >>>=, we can mask them using the binary operators &, &=, |, |=, ^, and ^=, and we can complement them using the unary operator ~.

The classes `BigDecimal` and `BigInteger` (both of which were introduced in JDK1.1) provide support for arbitrary-precision signed-decimal and binary arithmetic, respectively.

Exercise 1-9*: Write a program that has three `int` variables named `cost`, `markup`, and `quantity`, and initialize them to $20, $2, and 123, respectively. Print the result of *retail cost* × *quantity*, where *retail cost* equals *cost* + *markup*. (See lab directories Lbba04 and Lbba04b.)

1.5.3 The Floating-Point Types

All floating-point types are signed, use IEEE 754 representation, and have the following sizes and value ranges:

Table 1-3: The Floating-Point Types

Type	Size	Value Range
`float`	32 bits	$\pm 1.4\times10^{-45}$ to $3.4028235\times10^{+38}$
`double`	64 bits	$\pm 4.9\times10^{-324}$ to $1.7976931348623157\times10^{+308}$

Java supports IEEE 754's notion of ±∞ as well as NaN. (NaN is an abbreviation for Not-a-Number and is a value used to represent special cases such as zero divided by zero.)

Here's an example that uses the floating-point types (see directory Ba08):

```
public class Ba08
{
        public static void main(String[] args)
        {
                int i = 1;
                float f = i;
                double d = i;

/*1*/           System.out.println(f/3);
/*2*/           System.out.println(f/126);
/*3*/           System.out.println(f/126932);
/*4*/           System.out.println(d * 123456789);

                i = 2147483647;
/*5*/           d = f = i;
                System.out.println(i);
                System.out.println(f);
                System.out.println(d);
        }
}
```

which produces the following output:

```
0.33333334
0.007936508
7.878234E-6
1.23456789E8
2147483647
2.14748365E9
2.147483648E9
```

1.6 Literals

1.6.1 Boolean Literals

There are only two, `true` and `false`; we learned about them in §1.5.1.

1.6.2 Character Literals

Since characters really are represented as integers, variables of type `char` can be initialized with integer expressions, as follows:

```
char c = 65;    // 'A' has value 65 in Unicode
```

However, this requires us to know that the Unicode value of A is 65. To allow for readability, we can write such values in the form of *character literals* using the form `'`*x*`'`, for example:

```
'A'  '+'  'ß'  'ñ'  'æ'  '\n'  '\"  'Π'  '\u03A0'
```

As we can see, any Unicode character can be specified either by its printable form, by its escape sequence equivalent, or as a Unicode sequence. The single quote is represented by its equivalent escape sequence.

1.6.3 String Literals

A *string literal* consists of zero or more characters enclosed in double quotes. As we have seen, it can contain escape and Unicode sequences. **String literals cannot be modified.** The following are examples of string literals:

```
"Hello\n"   ""   "abc\tdef"   "\"quoted text\""   "\u00abcd"   "\uabcd"
```

Note that `"\u00abcd"` is subtly different to `"\uabcd"`; the former contains three characters, `'\u00ab'`, `'c'`, and `'d'`, while the latter contains only one, `'\uabcd'`. Remember, a Unicode sequence must always contain four hex digits.

1.6.4 Integer Literals

An integer literal has a base (or radix), as determined by the presence of a prefix. If it has a prefix of 0x or 0X, it is interpreted as a hexadecimal (base 16) number and is permitted to contain the characters a–f and A–F, as well as the digits 0–9. If the prefix is just 0 (zero), the number is interpreted as octal (base 8) and only the digits 0–7 are permitted. All other integer literals are deemed to be decimal (base 10) and can contain only the digits 0–9. Java 7 added support for binary (base 2) literals, which have a prefix of `0b` or `0B`.

Since a literal is an expression, and each expression has a type, a literal has a type as well as a value. An integer literal can explicitly be given the type `long` by the addition of a suffix of L or l.

Style Tip: When writing literals of type `long`, use L instead of l since the latter can easily be mistaken for the digit 1.

Consider the following code fragment:

```
/*1*/   long l1 = 2147483647;   // okay
/*2*/   long l2 = 2147483648;   // error
/*3*/   long l3 = 2147483648L;  // okay
/*4*/   long l4 = -2147483648;  // okay
/*5*/   long l5 = -2147483649;  // error
```

In the absence of an L or l suffix, an integer constant has type `int`; however, that constant's value cannot exceed the range of values for that type. As such, case 2 is an error even though the variable being initialized is large enough to store the specified value. To rectify this, we must add an explicit suffix as shown in case 3.

In case 4, we have an expression involving the unary minus operator and a non-negative constant. Note that 2147483648 is the magnitude of the smallest value that can be represented in a signed 32-bit integer. This number can only be written without a suffix if it is the operand of the unary minus operator; otherwise, it must have a suffix. Similarly, 9223372036854775808 is the magnitude of the smallest value that can be represented in a signed 64-bit integer. This number can only be written if it is the operand of the unary minus operator and has a suffix; otherwise, it is invalid.

Case 5 is invalid because integer constants that exceed the range of values representable by an `int` must have a suffix regardless of their base.

The following are examples of integer literals:

```
0b101           // binary, int
123456          // decimal, int
01234           // octal, int
0x09aB          // hex, int
0XfFc5          // hex, int
-2147483648     // decimal, int
5L              // decimal, long
013677777777L   // octal, long
0xAb2L          // hex, long
```

Consider the following initializer:

```
int i = 0L;     // error, possible loss
```

The type of the constant is `long`, which supports a larger range than the type of `i`, which is `int`. Therefore, the compiler issues an error even though the actual value of the initializer in this case can be represented in the variable.

There is no such thing as a negative-valued integer literal; for example, the expression -5 consists of the literal 5 and the unary minus operator.

Java 7 added support for having underscore character (“_”) separators between adjacent digits in an integer literal; for example:

```
long creditCardNumber = 9876_5432_1012_3456L;
long bytes = 0b11100011_00011100_10101010_01010101;
```

1.6.5 Floating-Point Literals

A floating-point literal must contain a decimal point, an exponent, or both. It may optionally contain whole number and fractional parts. The exponent can be written using either an upper- or lowercase E, and it can contain a leading sign. Both the value and exponent parts are interpreted as decimal.

A suffix of F or f indicates type `float`, while a suffix of D or d indicates type `double`. An unsuffixed floating-point literal has type `double`.

The following are examples of floating-point literals:

```
.952            // double (implicit)
3e34            // double     "
1.23E5          // double     "
```

```
345.D           // double (explicit)
3.456e+4d       // double     "
345.F           // float
3.456E-27f      // float
```

Consider the following initializer:

```
float f = 0.0;  // error, possible loss
```

The type of the constant is `double`, which supports a larger range and precision than the type of `f`, which is `float`. Therefore, the compiler issues an error even though the actual value of the initializer in this case can be represented in the variable.

Java 7 added support for having underscore character ("_") separators between adjacent digits in a floating-point literal; for example:

```
float pi =      3.14_15_26F;
```

1.7 Enum Types

An *enum type* is made up of a set of named constant values each of which is called an *enum member*. Each distinct enum constitutes a different enum type. The use of enum types and their enum members provides a useful amount of abstraction. For example (see directory Ba12):

```
/*1*/ enum CarColor {Black, White, Red};
/*2*/ enum HouseColor {Red, Green, Blue, Yellow};
```

In case 1, we define an enum called `CarColor` with three enum members, `Black`, `White`, and `Red`. (Enum member names are identifiers.) In case 2, we define an enum `HouseColor`, giving it four enum members.

```
public class Ba12
{
        public static void main(String[] args)
        {
/*3*/           CarColor c1 = CarColor.Black;
/*4*/           HouseColor c2 = HouseColor.Green;

/*5*/           c1 = CarColor.White;    // okay
//*6*/          c1 = HouseColor.Blue;   // error, incompatible types
//*7*/          c2 = CarColor.Red;      // error, incompatible types
/*8*/           c2 = HouseColor.Blue;   // okay
//*9*/          c1 = c2;                // error, incompatible types

/*10*/          System.out.println("c1 = " + c1 + ", c2 = " + c2);
        }
}
```

The output produced is:

```
c1 = White, c2 = Blue
```

To access enum members, we must prefix them with their parent enumeration type and the dot operator, as shown in cases 3 and 4. This allows multiple enumerations to have the same enum member names, as is the case with Red.

Since operations involving enumerations and enum members are strongly checked with respect to type compatibility, situations such as those occurring in cases 6, 7, and 9, are rejected.

In case 10, the values of `c1` and `c2` are written out as strings.

The use of an enumeration permits compile-time checking to be performed when a variable is permitted to take on any one of an enumerated set of values only. Since this is a common situation in programming, we should use enumerations whenever we can.

We can share an enum type definition between source files by defining that type in its own source file (having the same name as the enum type) and by declaring that type to be `public`.

1.8 Operator Precedence

When an expression involves more than one operator, the compiler determines the order in which it groups terms, based on the *precedence table*.[1] Consider the following example (see directory Ba09):

```
public class Ba09
{
        public static void main(String[] args)
        {
                int i = 5;
                long l = 500;
                float f = 3.58345f;
                double d;

/*1*/           d = i + l * f;              // d = i + (l * f)
                System.out.println("d = " + d);

/*2*/           d = (i + l) * f;
                System.out.println("d = " + d);

/*3*/           d = i + l + f;              // d = (i + l) + f
                System.out.println("d = " + d);

/*4*/           System.out.println("i squared = " + i * i);
/*5*/           System.out.println("twice i = " + i + i);
/*6*/           System.out.println("twice i = " + (i + i));
        }
}
```

The output produced is:

```
d = 1796.72509765625
d = 1809.642333984375
```

[1] This table is shown in Annex A, along with a discussion of operator precedence and associativity.

```
d = 508.58343505859375
i squared = 25
twice i = 55
twice i = 10
```

In case 1, multiplication has higher precedence than addition, resulting in the value of `l` being multiplied by the value of `f` and the result being added to the value of `i` then stored in `d`. The default precedence and associativity can be overridden via the use of grouping parentheses, as shown in case 2. Here, the values of `i` and `l` are added, and the result is multiplied by the value of `f`.

In case 3, the two addition operators have the same precedence, since they are the same operator. However, this operator has left-to-right *associativity*; that is, the left one is given higher precedence. Therefore, the values of `i` and `l` are added, and the result is then added to the value of `f`.

In case 4, the multiplication has higher precedence than addition so the square of `i` is computed before being written. However, case 5 does not give the result we might have expected. Here, the first `i` is concatenated to the string literal forming a new string. Then the second `i` is concatenated and that string is output, resulting in 55 instead of 10 as produced by case 6. The fact that + has meaning both for string literals and numeric types matters not, the precedence is the same for both situations.

Unlike some languages (such as C and C++), Java guarantees that the order in which terms of an expression are evaluated is left-to-right. Consider the following:

```
f() + g() * h()
```

Here, the precedence is quite clear; multiplication wins out over addition, as demonstrated by the corresponding parse tree:

```
    +
   / \
  /   \
f()    *
      / \
     /   \
   g()   h()
```

However, in what order are the three terms themselves evaluated? That is, in what order are the three methods called? Terms (which are, of course, subexpressions) are evaluated left-to-right in the expression as written. For example, although the following expressions can be represented mathematically by the same parse tree, each has a different order of evaluation of terms, as indicated by the corresponding comments:

```
f() + g() * h()         // f, g, then h
f() + h() * g()         // f, h, then g
g() * h() + f()         // g, h, then f
h() * g() + f()         // h, g, then f
```

1.9 Type Conversion

Like most languages, Java permits expressions to contain terms of different types, provided the types are compatible. Consider the following fragment taken from an earlier example; it contains a mixed-mode expression:

```
int i = 5;
long l = 500;
float f = 3.58345f;
double d;

d = i + l + f;     /* d = (i + l) + f */
```

The precedence table determines how pairs of terms are grouped. Because of that grouping, the type of each subexpression is determined. In the example above, since `i` and `l` have different (but compatible) types, implicit conversion of the value of `i` to `long` occurs, and the result of the addition has type `long`. Then that value is converted to type `float` so it can be added to `f`, producing a result of type `float`. The value of this result is then converted to type `double` during its assignment to `d`.

The order in which we write the terms in this assignment expression can be important. For example, the following are subtly different:

```
/*1*/   d = i + l + f;          /* d = (i + l) + f */
/*2*/   d = f + i + l;          /* d = (f + i) + l */
```

In case 2, the value of `i` is converted to type `float` and then added to `f`, producing a result of type `float`. Then, the value of `l` is converted to type `float` and added to the result of the previous addition. For many values of `i`, `l`, and `f`, the two versions will produce the same result. However, consider case 1 when `l` contains the largest value that can be stored in a variable of type `long`, and the value of `i` is 1 or more. When they are added, wrapping occurs, since the result cannot be represented in the type `long`. If, however, the expression is rearranged as in case 2, all intermediate results are performed in type `float`, so no integer wrapping can occur.

Controlling the type of intermediate results by rearranging the expressions is poor style. Instead, it is better to use explicit type conversion. This is done via the *cast* operator, which has the following general form:

(*type-name*) *operand*

This unary prefix operator consists of a type name inside parentheses. For example, in the case of `(int)x`, an unnamed variable of type `int` is created and its value is that of `x` converted to type `int`. The type and value of `x` is unchanged. We can use a cast in our previous example to avoid the possibility of integer wrapping, as follows:

```
/*1*/   d = (float)i + l + f;
/*2*/   d = i + (float)l + f;
/*3*/   d = (float)i + (float)l + f;
```

In case 1, the value of `i` is explicitly converted to type `float`, resulting in the value of `l` being implicitly converted to type `float` as well. In case 2, the value of `l` is explicitly converted to type `float`, resulting in the value of `i` being implicitly converted to type `float`. In case 3, both `i` and `l` are explicitly cast. All three cases are equivalent.

The value of an expression having any numeric type can be cast explicitly to any numeric type; however, it may result in a loss of precision, for example, when a `double` is cast to a `float` or to an `int`. An expression can be cast to its own type, although such a cast has no effect.

Style Tip: Avoid unnecessary casts because they detract from readability.

The following conversions can result in a loss of significant information and/or precision. Therefore, such conversions require an explicit cast:

- `byte` to `char`
- `short` to `byte` or `char`
- `char` to `byte` or `short`
- `int` to `byte`, `short`, or `char`
- `long` to `byte`, `short`, `char`, or `int`
- `float` to `byte`, `short`, `char`, `int`, or `long`
- `double` to `byte`, `short`, `char`, `int`, `long`, or `float`

Even though `char` and `short` both use 16 bits, the values of a `short` are signed while those of a `char` are not, so neither can represent the complete range of values of the other.

There are several exceptions to this conversion requirement; for example:

```
/*1*/   byte b1 = 10;
/*2*/   byte b2 = (byte)10;     // redundant cast
/*3*/   short s1 = 10;
/*4*/   short s2 = (short)10;   // redundant cast
```

In case 1, the type of the initializer is `int` yet the variable being initialized is a `byte`, so truncation is involved. However, since the `int` is a constant, the compiler allows such constructs if the value of the constant can be represented in a `byte`. Case 2 does explicitly what the compiler already allows implicitly in case 1. Similarly, in case 3 a `short` is being initialized with an `int` constant. Note, however, that unless the initializing value in cases 1 and 3 is a constant, it will be rejected. For example:

```
/*1*/   int i = 0;
/*2*/   byte b3 = i;            // error, possible loss
/*3*/   byte b4 = (byte)i;      // compiles
```

The remaining case involves the initialization of a `char`:

```
/*1*/   char c = 65;
/*2*/   char c = (char)65;
```

In case 1, the type of the initializer is `int` yet the variable being initialized is a `char`, so truncation is involved. However, since the `int` is a constant, the compiler allows such constructs if the value of the constant can be represented in a `char`. Case 2 does explicitly what the compiler already allows implicitly in case 1.

1.10 Introduction to Class Members

Earlier in this chapter, we saw an example involving one method, `welcome`, being called from another method, `main`, where the definitions of these methods resided in different classes. The following example (see directory Ba10) uses that same approach, and introduces some other features:

```
public class Ba10a
{
// class variables
/*1*/   public final static int NUM_MONTHS = 12;

/*2*/   private static float value;

// class methods
        public static void main(String[] args)
        {
/*3a*/          System.out.println("main:value = " + value);
/*3b*/          changeValue();
/*3c*/          System.out.println("main:value = " + value);

/*4a*/          Ba10b.display1();
/*4b*/          Ba10b.display2();

/*5*/           int value = 200;
/*6*/           System.out.println("main:value = " + value);
/*7*/           System.out.println("main:value = " + Ba10a.value);
        }

        private static void changeValue()
        {
/*8*/           value = -1.87F;
        }
}
```

Not only can we define methods in classes, we can also define variables there, outside of the bodies of all methods. Such variables are called *fields*; NUM_MONTHS and value are examples, and each has the type shown. Since NUM_MONTHS is public it is accessible from outside its parent class, provided its name is prefixed with its parent class, as shown in case 9 below. It can be accessed without that prefix from within methods in its own class. Since value is private, it cannot be accessed outside its parent class, although it can be accessed by any method in its class. Both fields are static indicating that they are *class variables*; that is, they are variables belonging to this class.[1]

NUM_MONTHS has been declared final, which means that once it has been initialized, its value cannot be changed. Note that an initializer need not be a constant expression. The recommended style of naming final fields is to use all capitals with an underscore (_) as a multiword separator, as shown above.

By default, arithmetic fields take on a value of 0 or 0.0, as appropriate, which is demonstrated by the output from case 3a. In case 3b, we call changeValue, and since this is a method in the same class, it has access to Ba10a's class variables, allowing it to change value in case 8. The new value is reflected by the output from case 3c.

Let's ignore the calls to display1 and display2 for now, since they have no impact on the program's output from this class.

[1] We'll learn more about class variables and instance variables in §5.

In case 5, we define a local variable whose name is the same as that of a field in the parent class. This is permitted and causes the field to be hidden by the local variable name (even though the two types differ) from the point at which that local variable is defined, to the point at which it goes out of scope, which happens at the closing brace of its enclosing block. As such, the output from case 6 reflects the local variable's value; however, as we can see in case 7, we can still access the field by using its class name as a prefix.

Ba10b.java contains:

```
public class Ba10b
{
/*9*/   private static float value = Ba10a.NUM_MONTHS * 10.5F;

        public static void display1()
        {
/*10*/          System.out.println("display:value = " + value);
/*11*/          value = 53.56F;
        }

        public static void display2()
        {
/*12*/          System.out.println("display:value = " + value);
        }
}
```

The class variable `value` defined in case 9 belongs to class `Ba11b` and is completely separate from the class variable having the same name in class `Ba11a`. As a result, references to `value` in cases 10, 11, and 12, refer to `Ba11b.value`.

The output produced is:

```
main:value = 0.0
main:value = -1.87
display:value = 126.0
display:value = 53.56
main:value = 200
main:value = -1.87
```

As we have seen, a class introduces a namespace, so we can define a class to contain *final* variables (constants), variables, and methods for general use. In fact, the standard Java library contains a number of classes that do just this. Here are excerpts from several of the more commonly used ones:

```
public class Character
{
// Fields
        public final static char MIN_VALUE = '\u0000';
        public final static char MAX_VALUE = '\uffff';

// Methods
        public static boolean isDigit(char ch);
        public static boolean isLetter(char ch);
        public static boolean isLetterOrDigit(char ch);
        public static boolean isLowerCase(char ch);
        public static boolean isWhitespace(char ch);
        public static boolean isUpperCase(char ch);
        public static char toLowerCase(char ch);
        public static char toUpperCase(char ch);
}
```

The constants indicate the smallest and largest value, respectively, that can fit into a variable of type `char`. The methods each operate on a `char` argument either to test that character's attributes or to produce the converted value of that character. Program Ba11 demonstrates the use of some of these methods and here are excerpts from that program, along with the corresponding output:

```
        char c = 'A';

/*1*/   System.out.println("isDigit:  " + Character.isDigit(c));
/*2*/   System.out.println("isLetter: " + Character.isLetter(c));
/*3*/   System.out.println("isLower:  " + Character.isLowerCase(c));
/*4*/   System.out.println("isWhite:  " + Character.isWhitespace(c));
/*5*/   System.out.println("isUpper:  " + Character.isUpperCase(c));
/*6*/   System.out.println("toLower:  " + Character.toLowerCase(c));
/*7*/   System.out.println("toUpper:  " + Character.toUpperCase('?'));
```

The output produced is:

```
isDigit:  false
isLetter: true
isLower:  false
isWhite:  false
isUpper:  true
toLower:  a
toUpper:  ?
```

The output is obvious except perhaps for the result of `toUpperCase`. This method and its companion, `toLowerCase`, return the converted value of their argument, or, if the argument cannot be converted, they return the value of the argument.

Note that the method `Character.isWhitespace` was a JDK1.1 invention. Prior to that release, method `isSpace` was used instead.

Class `Integer` provides constants and methods that pertain to the type `int`. Here are some of them:

```
public class Integer
{
// Fields
        public final static int MIN_VALUE = 0x80000000;
        public final static int MAX_VALUE = 0x7fffffff;

// Methods
        public static String toBinaryString(int i);
        public static String toHexString(int i);
        public static String toOctalString(int i);
        public static String toString(int i);
}
```

Here are more excerpts from Ba11:

```
/*8*/   System.out.println("toBinary: " + Integer.toBinaryString(100));
/*9*/   System.out.println("toHex:    " + Integer.toHexString(100));
```

The output produced is:

```
toBinary: 1100100
toHex:    64
```

In the following code fragment, the two cases are equivalent; while the first results in an implicit conversion of the `int` value to a string, the second does it by an explicit call to the `toString` method:

```
        int value = -45;
/*10a*/ System.out.println("string: " + value);
/*10b*/ System.out.println("string: " + Integer.toString(value));
```

The output produced is:

```
string: -45
string: -45
```

Class Double provides constants and methods that pertain to the type double. Here are some of them:

```
public class Double
{
// Fields
        public final static double MIN_VALUE = 2.2250738585072014E-308;
        public final static double MAX_VALUE = 1.79769313486231570e+308;
        public final static double NaN = 0.0/0.0;
        public final static double NEGATIVE_INFINITY = -1.0/0.0;
        public final static double POSITIVE_INFINITY = 1.0/0.0;

// Methods
        public static boolean isInfinite(double v);
        public static boolean isNaN(double v);
        public static String toString(double d);
}
```

Here are more excerpts from Ba11:

```
        int value = -45;
/*11a*/ System.out.println("string: " + (5.6 + value));
/*11b*/ System.out.println("string: " + Double.toString(5.6 + value));
```

The output produced is:

```
string: -39.4
string: -39.4
```

The code continues:

```
/*12*/  System.out.println("string: " + Double.NEGATIVE_INFINITY);
```

The output produced is:

```
string: -Infinity
```

As we have seen, the classes `Character`, `Integer`, and `Double` support the primitive types `char`, `int`, and `double`, respectively. There are also classes that perform similar operations on the types `byte`, `short`, `long`, and `float`. Here is the set of classes and their corresponding primitive types:

Class	**Primitive Type**
`Boolean`	`boolean`
`Character`	`char`
`Byte`	`byte`
`Short`	`short`
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`

Class `Math` provides constants and methods that pertain to mathematics. Here are some of them:

```
public class Math
{
// Fields
        public final static double E = 2.7182818284590452354;
        public final static double PI = 3.14159265358979323846;

// Methods

        public static float abs(float a);
        public static double abs(double a);
        public static double log(double a);
        public static double sin(double a);
        public static double sqrt(double a);
}
```

Note that most methods in the math library are only available in double precision.

Exercise 1-10: Look at the documentation for the classes that directly support for the primitive types and see what public final static variables and public static methods they provide. In particular, look at the `toString` methods in `Float` and `Double` to see how they format certain special and other values.

Exercise 1-11*: A circle's diameter is 10.3 cm. Compute and display its area and circumference given that $a = \pi r^2$ and $c = 2\pi r$. The area and circumference variables have type `float`. (See lab directories Lbba05 and Lbba05b.)

4. References, Strings, and Arrays

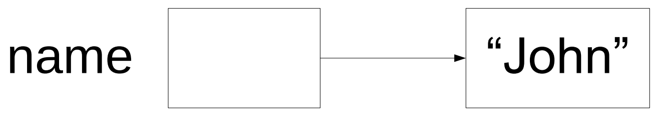
In this chapter, we will learn about reference variables, the allocation of memory at runtime using `new`, garbage collection, array allocation and manipulation, and the string classes `String` and `StringBuffer`.

4.1 Introduction

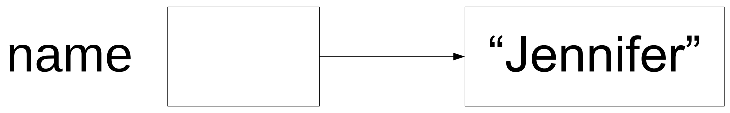
There are two categories of types in Java: primitive and reference. We have already seen examples of the primitive types `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`.

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):

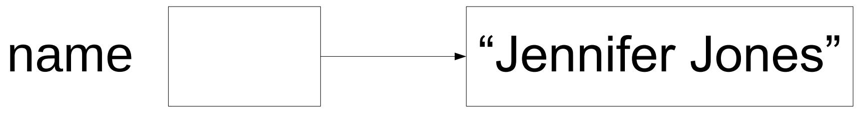
```
public class Rf01
{
        public static void main(String[] args)
        {
/*1*/           String name = "John";
/*2*/           System.out.println("name is " + name);
```



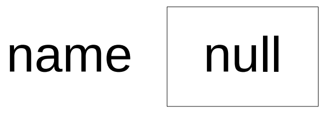
```
/*3*/           name = "Jennifer";
                System.out.println("name is " + name);
```



```
/*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.

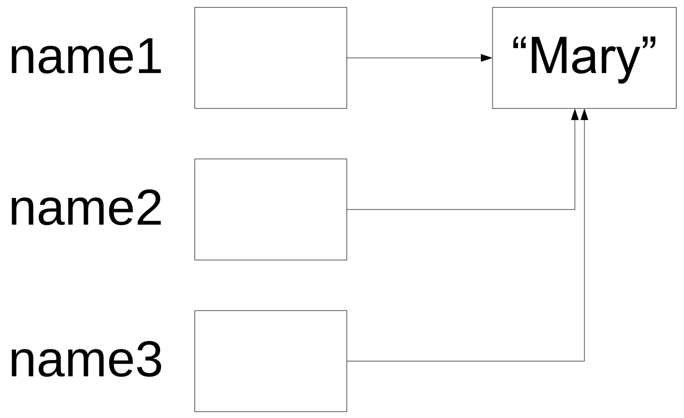
In case 4, the two strings are concatenated resulting in a third string to which `name` is made to point. Because a string's value is read-only, the second string cannot simply be appended to the end of the first one.

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.

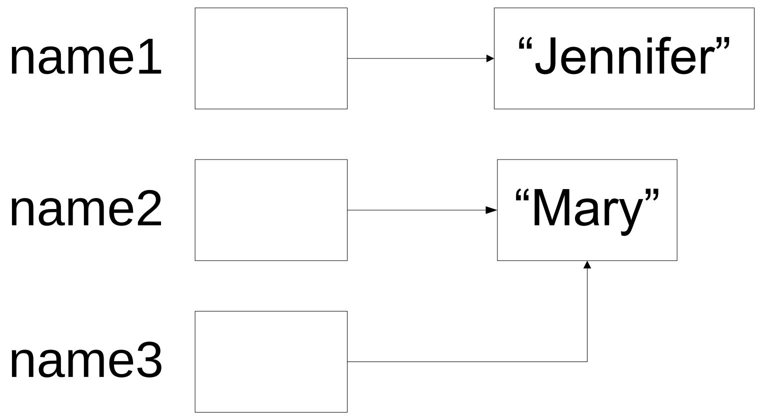
There can be any number of references to the same object. For example (see directory Rf02):

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

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



```
                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:

```
name1 is Mary
name2 is Mary
name3 is Mary

name1 is Jennifer
name2 is Mary
name3 is Mary
```

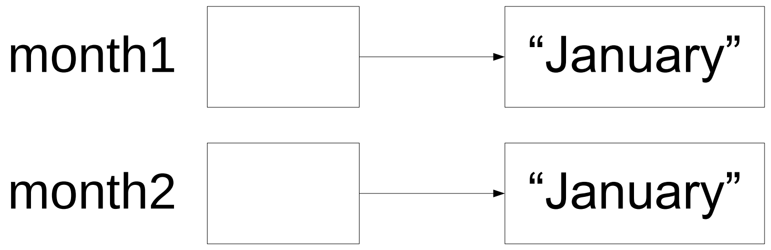
In cases 1 and 2, `name2` and `name3` are both made to refer to the same string, that also being referred to by `name1`; however, all three are separate reference variables—their values just happen to be the same. Therefore, when `name1` is made to refer to a different string, as in case 3, `name2` and `name3` still refer to `"Mary"`.

In §1.10, we learned that a class can have fields and methods and that there were a number of classes available in the Java library. For example, to test if the value of some `char c` is a digit, we can call the `isDigit` method of class `Character` using the notation `Character.isDigit(c)`. We prefix method names like `isDigit` with their parent class name because such methods belong to the class as a whole. A class can have other methods, which, instead of belonging to the class, are related to a particular instance of that class. For example, the `String` class contains a method called `length` that returns the length of some string. On its own, this method is useless; to work, it needs to operate on a specific string object. For example, to call it for a string called `name`, we use `name.length()`.

A method having the modifier `static` is a class method and belongs to the class as a whole. It can be called using the class name as a prefix. A method not having the modifier `static` is an instance method and can be called to operate on an instance of its parent class in which case, the name of that instance is used as the prefix.

Using this information, let's look at an example that calls some instance methods defined in class `String` (see directory Rf03):

```
public class Rf03
{
        public static void main(String[] args)
        {
/*1*/           String month1 = "January";
                String x = "u";
/*2*/           String month2 = "Jan" + x + "ary";
```



```
/*3*/           System.out.println("==          " + (month1 == month2));
/*4*/           System.out.println("==          " + month1 == month2);
/*5*/           System.out.println("equals      " + month1.equals(month2) + '\n');

                System.out.println("indexOf     " + month1.indexOf('a'));
/*6*/           System.out.println("indexOf     " + month1.indexOf('x'));
                System.out.println("indexOf     " + month1.indexOf("nu"));
                System.out.println("lastIndexOf " + month1.lastIndexOf('a'));
                System.out.println("substring   " + month1.substring(2, 5));
/*7*/           System.out.println("replace     " + month1.replace('a', '?'));
                System.out.println("month1      " + month1 + '\n');
```

```
/*8*/           int length = month1.length();
                for (int i = 0; i < length; ++i)
                {
/*9*/                   System.out.println("i = " + i + ", " + month1.charAt(i));
                }
        }
}
```

The output produced is:

```
==          false
false
equals      true
indexOf     1
indexOf     -1
indexOf     2
lastIndexOf 4
substring   nua
replace     J?nu?ry
month1      January
i = 0, J
i = 1, a
i = 2, n
i = 3, u
i = 4, a
i = 5, r
i = 6, y
```

In case 1, we define `month1` and make it refer to the string `"January"` created by the string literal. In case 2, we define `month2` and also make it refer to a string spelled `"January"`; and while these two strings just happen to contain the same characters, *they are not the same string!* This can be seen from the output from case 3, which shows that the two references do not refer to the same object. If we wish to compare the strings' content for equality, we must use the method `equals`, as shown in case 5. There is also a case-insensitive comparison method called `equalsIgnoreCase`.

Case 4 is worth a mention since it compiles but gives a (possibly) surprising answer. Since addition has a higher precedence than equality, the string literal and `month1` are concatenated and the resulting string's reference is compared with `month2` to see if they refer to the same string. Of course, they don't, so the result is false. To make the equality test go first, we need the grouping parentheses, as shown in case 3.

`indexOf` searches for the first occurrence of a given character in a string. Since character positions within a string start at zero, the first a is found at offset 1. If no match is found, -1 is returned, as in case 6. This method is overloaded to allow searching for a string. `lastIndexOf` searches for the first occurrence of a given character in a string, starting from the back end. `substring` extracts a substring with the given start and end positions; however, note that the end position specified is actually one more than the end character position actually used. In case 7, `replace` creates a new string whose contents have the given character changed as specified. As demonstrated, the contents of the string referenced by `month1` are unchanged.

To find the length of a string, we call its `length` method, as shown in case 8. We can then use that value to traverse the string, picking off one character at a time using `charAt`. Note that since a string is not an array, we cannot access its characters via subscripting.

In this example, we have ignored the possibility of error. For example, what if `charAt` is given an index that is out of bounds? Likewise, for `substring`. What if `indexOf` is given a null reference instead of a reference to a string? These kinds of errors generate what are known as *exceptions*, which are discussed in detail in §7. For now, it is only important to know that if an exception is produced and the program does not declare that it is ready to handle it (using special syntax that we'll see later), the program terminates abnormally. The good news is that such errors cannot go undetected.

Exercise 4-1*: What happens if you try to display the value of a reference variable that hasn't been initialized? (See lab directory Lbrf01.)

4.2 Sharing of Like Strings

A Java compiler is required to share copies of strings. Specifically, the "Java Language Specification", states that

- Literal strings within the same class in the same package represent references to the same String object.
- Literal strings within different classes in the same package represent references to the same String object.
- Literal strings within different classes in different packages likewise represent references to the same String object.
- Strings computed by constant expressions are computed at compile time and then treated as if they were literals.
- Strings computed at run time are newly created and therefore distinct.
- The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

Consider the following example (see directory Rf13):

```
class Rf13a
{
        public static void main(String[] args)
        {
                String color = "Green", en = "en";

/*1*/           System.out.println(color == "Green");
/*2*/           System.out.println(Rf13b.color == color);
/*3*/           System.out.println(color == "Gre" + "en");
/*4*/           System.out.println(color == "Gre" + en);
/*5*/           System.out.println(color == ("Gre" + en).intern());
        }
}

class Rf13b
{
        public static String color = "Green";
}
```

which produces the output

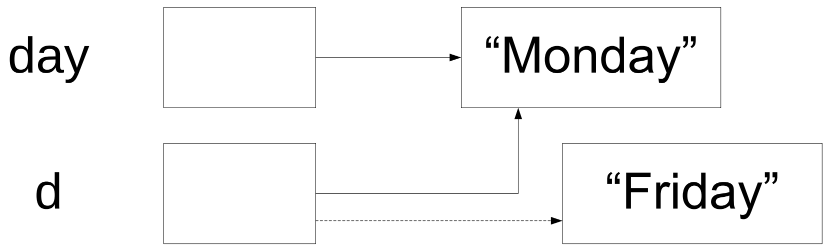
```
true
true
true
false
true
```

As we can see, the compiler is required to share like-spelled string literals in various ways. The programmer can also share these with strings created at runtime by using the method `intern`, as shown.

4.3 Passing and Returning References

Like variables of a primitive type, we can pass reference variables to a method and have a method return a reference. For example (see directory Rf04:

```
public class Rf04
{
        public static void main(String[] args)
        {
                String day = "Monday";
                System.out.println("day is " + day);

/*1*/           changeDay(day);
/*2*/           System.out.println("day is " + day);

/*3*/           day = newDay();
/*4*/           System.out.println("day is " + day);
        }
```



```
/*5*/   private static void changeDay(String d)
        {
/*6*/           d = "Friday";
        }

/*7*/   private static String newDay()
        {
/*8*/           return "Wednesday";
        }
}
```

The output produced is:

```
day is Monday
day is Monday
day is Wednesday
```

In case 1, we pass the value of day to changeDay. However, as we learned in §3.2, in Java all arguments are passed by value; that is, a copy of their value is passed to the called method. Therefore, what changeDay receives is a copy of the reference day, which, in case 5, is called d. In case 6, this copy is made to refer to some other string; however, when changeDay returns, the copy is destroyed, and day still refers to "Monday", as shown by the output produced by case 2.

Method newday is defined to return a reference to a string in case 7 and it does just that in case 8. This reference is then assigned to day is case 3 resulting in "Wednesday" being displayed in case 4.

There really are no surprises here; it's just a matter of remembering that all arguments are passed by value and that a reference only refers to an object, it is not the object itself.

4.4 Allocating Memory for Objects

When we define a primitive variable, memory is allocated for it at that time. However, when we define a reference variable, memory is only allocated for the reference, not for any object to which it might refer. So far, the only way we have seen to allocate memory for an object is to use a string literal. However, in many situations, we don't know the value of a string at compile-time, and, in any event, eventually we'll need object types other than strings. How then can we allocate memory for an object, on demand? We do this by using the new operator, which has the following general format:

new *class-type* ([*argument-list*])

Here's an example of its use (see directory Rf05):

```
public class Rf05
{
        public static void main(String[] args)
        {
/*1a*/          String str1 = "ABC";
/*1b*/          String str2 = new String("ABC");

                String str3 = str2;
/*2*/           System.out.println("str2 == str3 is " + (str2 == str3));

/*3*/           String str4 = new String(str2);
/*4*/           System.out.println("str2 == str4 is " + (str2 == str4));
/*5*/           System.out.println("str2.equals(str4) is " + str2.equals(str4));
        }
}
```

The output produced is:

```
str2 == str3 is true
str2 == str4 is false
str2.equals(str4) is true
```

In case 1a, we implicitly allocate memory for a string and store a reference to that in `str1`. In case 1b, we allocate memory explicitly for a string and store a reference to that in `str2`. What is really happening here is that `String()` results in a call to a special method in the `String` class, called a *constructor*. (Like other methods in a class, a constructor can be overloaded.) This method constructs a new string whose contents are "ABC".

As we learned earlier, `str2` and `str3` are references to the same string, so it should come as no surprise that the equality expression in case 2 produces true. However, in case 3, we call a constructor that takes one argument, of type `String`. This results in the value of the object being allocated, being initialized to the string specified by the argument; that is, `str4` refers to a copy of the string referred to by `str2` rather than to the same string as `str2`. So, the equality expression in case 4 produces false. Of course, the `equals` method still returns true since both strings have the same content.

A `new` expression produces a reference to an object of the given class type, and can be used in any context in which such a reference is permitted; for example:

```
String s1 = "Hello";
String s2 = new String(s1) + " there";

boolean b = (new String(s1)).equals("house");
```

The place from which `new` obtains memory is called the *heap*.

It is important to note that we *cannot* allocate memory for variables of primitive type directly using `new`. (See §4.9 for more information.)

4.5 Releasing Allocated Memory

As we have seen, we can allocate any number of objects using new, but we never seem to free up the memory we allocated. Don't we run the risk of running out of memory? While it is possible to run out of memory, the most likely answer is "No". At runtime, the Java environment runs what is called a *garbage collector* to gather up discarded memory and make it part of the heap again. So as long as we discard memory when we no longer need it, our program should be able to get more memory. (Of course, we do not have an infinite amount of memory.) While we can explicitly discard memory, in the vast majority of cases, memory is discarded as a by-product of some other action. Let's study the following example to see when this occurs (see directory Rf07):

```
public class Rf07
{
        public static void main(String[] args)
        {
/*1*/           String day = new String("Monday");
/*2*/           String anotherday = day;
/*3*/           day = "Friday";
/*4*/           anotherday = null;

/*5*/           String newday = test(new String(day));
/*6*/   }

/*7*/   private static String test(String s)
        {
/*8*/           String str = new String(s);
/*9*/           return str;
        }
}
```

Each object allocated by `new` has associated with it a *reference count* which simply keeps track of the number of reference variables that currently refer to that object. When an object's reference count is decremented to zero, that object becomes a candidate for garbage collection.

In case 1, we allocate a new object with value `"Monday"` and refer to it using `day`; the object's reference count is now 1. In case 2, we make `anotherday` refer to that same object, so the reference count goes to 2. In case 3, we make `day` refer to the string `"Friday"`, so Monday's reference count goes back to 1. In case 4, we make `anotherday` refer to the null reference, reducing Monday's reference count to zero making it eligible for garbage collection.

A new object is created in case 5 resulting in a reference count of 1, even though no named reference actually refers to that object. This reference is passed to `test` by value resulting in `s`'s being a reference to the same object, whose reference count is now 2. No other references are made to this object within `test` so when `test` returns, the reference `s` is destroyed as is that returned by the `new` operator before the method was called, resulting in the reference count being decremented by 2, to zero.

In case 8, we create a new object which is a copy of that referred to by `s`. (This action does not change the reference count of the object referred to by `s`, however.) When we return `str` in case 9, we are really returning a copy of that reference which we then assign to `newday` back in `main`. When `test` returns, the reference `str` is destroyed and the corresponding reference count is decremented, however, it doesn't become zero, since `newday` still refers to that string. When `main` returns, `newday` is destroyed thereby reducing the reference count of the object it refers to, to zero.

While we can decrement an object's reference count by one explicitly by assigning `null` to a reference to that object, in most cases, it doesn't really help, and it probably clutters up the source code unnecessarily.

When does garbage collection take place? For the most part, the collection of discarded memory is under the control of the Java run-time environment; however, there are three times it is done:

1. When the amount of memory left in the heap drops below some minimum threshold.
2. When cleanup is invoked explicitly by the program by calling the library method `System.gc`.

3. Otherwise, whenever the run-time environment gets around to it.

Having automatic garbage collection can remove a significant burden from certain kinds of programs. For example, consider the case in which we have a singly linked list containing several hundreds or even thousands of nodes. We have a root reference to the first node in that list, the first node has a reference to the second, and so on. Assuming that each object is referenced by only one reference variable, if we assign a new value to the root reference variable or that variable is destroyed, the whole list becomes eligible for garbage collection; we do not need to traverse the list freeing up each individual node!

Exercise 4-2*: Define a method called `compressString` that takes a String argument and strips off any leading and trailing white space from it, reduces any embedded consecutive series of white space characters to a single space, and then returns the resultant String. For example, an argument of `"\t ABC \f DE\n"` will be returned as `"ABC DE"`. Hint: Method `trim` in class `String` will help. Define `compressString` in public class `LocalTools` and write a program (in a different class) to test it. (See lab directory Lbrf02 and LocalTools.)

4.6 Arrays

Organizing data into arrays allows us to process all, or a range, of elements in that array quite easily. Since arrays are such common ways of representing lists, Java supports them. There is no limit on the number of dimensions an array can have.

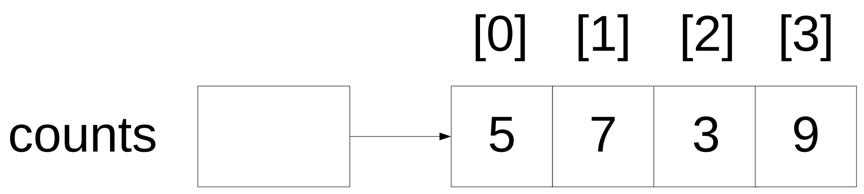
The reason we have deferred the discussion of arrays until now is that their usage involves references. Let's look at several array definitions (each taken from the file Rf08.java):

```
long[] sizes;
float[][] readings;
String[][][] names;
```

Unlike most other languages, when we define what looks like an array in Java, what we are really defining is a reference to an array. In this example, `sizes` is a reference to a one-dimensional array of `long`, `readings` is a reference to a two-dimensional array of `float`, and `names` is a reference to a three-dimensional array of `String`. As defined, these references are uninitialized; in short, there are no arrays in this example, only three reference variables each of which is capable of referring to an array of the corresponding number of dimensions and element type. Note that no dimension sizes are specified nor permitted.

In the following three examples, we not only define and allocate memory for three reference variables, we also allocate memory for, and initialize, the arrays to which they refer:

```
int[] counts = {5, 7, 3, 9};
```



`counts` is made to refer to an array of four `int`s, `counts[0]` through `counts[3]`.[1] Each of these is called a *component*, which refers to the corresponding `int` element in the underlying array.

Each array reference has a read-only public field called `length`, which maintains at runtime, the number of components in that array. In the case of `counts`, the length is 4. This allows us to loop through the array without having to remember the number of iterations, as follows:

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

The output produced is:

```
length of counts = 4
counts[0] = 5
counts[1] = 7
counts[2] = 3
counts[3] = 9
```

As we can see from the initializer list for `newcounts`, the initializer expressions need not be compile-time constants:

```
int[] newcounts = {6, counts[1], counts[3]/3};
```

For multidimensional arrays, the larger initializer list contains other initializers, as necessary. For example, a three-dimensional array initializer would be a list containing some lists, each of which contains yet more lists, as follows:

```
double[][] values = {
        {1.2, 3.4, 4.5},
        {5.6, 9.8, 5.4},
        {2.6, 9.2, 3.3},
        {7.6, 5.3, 0.6}
};
```

```
System.out.println("length of values    = " + values.length);
System.out.println("length of values[0] = " + values[0].length);
```

The output produced is:

```
length of values    = 4
length of values[0] = 3
```

We initialized `values` to refer to a 4×3 array. In reality, `values` refers to an array of 4 references, each of which refers to an array of 3 `doubles`. As such, the length of `values` is 4 (the row count) and the length of `values[0]`—and each other row—is 3 (the column count). As shown, we can use an array name with fewer than

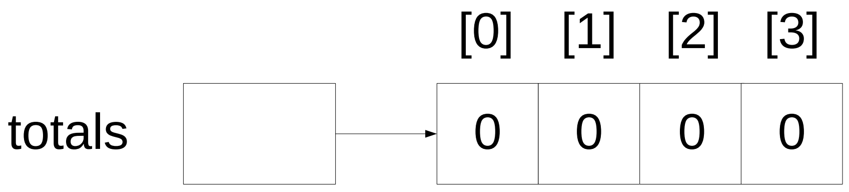
[1] An array of size *n* has subscripts in the range 0 to *n*-1. If we try to use a subscript value outside this range, an exception is produced at runtime. For an example, see §7.1.

the maximum number of subscripts. For example, `values` refers to the array of 4 references, `values[i]` refers to the ith array of 3 references, while `values[i][j]` refers to the `double` in row *i*, column *j*.

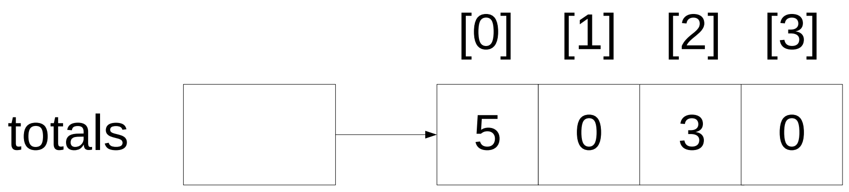
In all three examples above, the memory for the elements is allocated by the initializers.

We don't always know the number of elements we want in an array and even when we do, we don't necessarily know all their values; for example, we may be computing them. How then can we allocate an array containing a specific number of elements without providing an initializer list?

```
int[] totals = new int[4];
```



```
totals[0] = 5;
totals[2] = 3;
```



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

The output produced is:

```
totals[0] = 5
totals[1] = 0
totals[2] = 3
totals[3] = 0
```

Since references are involved, `new` was bound to show up eventually. In this example, we make `totals` refer to an array of 4 `int` which we allocate explicitly. We then initialize two elements; however, as shown by the output, all four have initial values with zero being used by default.

JDK1.1 added a feature called an *anonymous array*. For example (see directory Rf15):

```
public class Rf15
{
        public static void main(String[] args)
        {
                int i = 10, j = 20, k = 30;

/*1*/           int[] a = new int[] {i, j, k};

                f(a);
/*2*/           f(new int[] {i, -5, k, 123});
        }

        private static void f(int[] x)
        {
                System.out.println("Length = " + x.length);
        }
}
```

In cases 1 and 2, we create an unnamed array using a `new` expression.

When allocating memory for an array, the size we specify can be any non-negative integer value.[1] For example:

```
int[] xx1 = new int[1];
int[] xx2 = new int[0];
System.out.println("length of xx2 " + xx2.length);
int len = 24;
int[] xx3 = new int[len];
System.out.println("length of xx3 " + xx3.length);
```

The output produced is:

```
length of xx2 0
length of xx3 24
```

Not only can we have an array of size 1, we can even have an array of size zero! And, as shown, the allocation size need not be a compile-time constant. If we try to create an array with a negative size, however, an exception is produced at runtime.

One very powerful aspect of the way in which Java supports arrays is that array references can be changed. For example:

```
int[] yy = new int[4];
int[] aa = yy;
yy = new int[20];
yy = null;
```

The output produced is:

```
double[][] zz = new double[3][4];
zz = new double[6][2];
```

[1] The type of an array creation expression or a subscript expression cannot be `long`.

As the size of an array is not part of the reference type, `yy` can refer to *any* one-dimensional array of `int`. Initially, we make it refer to an array of 4 elements, then we make it refer to an array of 20, and then to the null reference. In the case of `zz`, it can be made to refer to any two-dimensional array of `double`. Therefore, while the number of elements in an array is fixed at the time that array is created, we can make an array reference refer to arrays of different sizes.

Since `yy` and `zz` are references, when we make them refer to new objects, the reference count of the objects to which they previously referred, are decremented, possibly making them eligible for garbage collection.

By being able to ignore the sizes of a dimension, we can write methods that can handle any array of a given number of dimensions and containing a given element type. For example, when we execute these calls:

```
dump1DIntArray(counts);
dump1DIntArray(newcounts);
```

the following method:

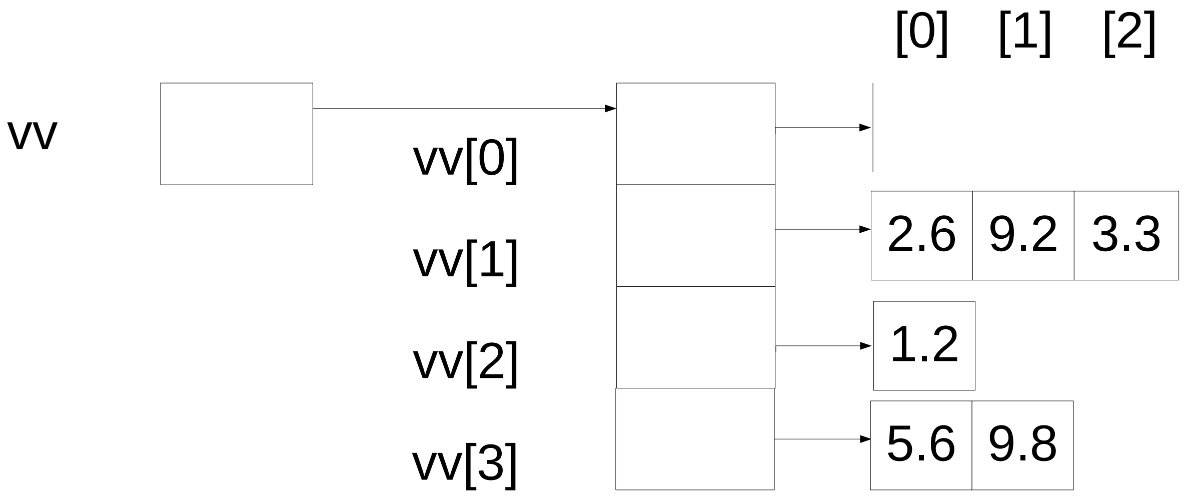
```
private static void dump1DIntArray(int[] x)
{
        if (x == null)
        {
                System.out.println("null array ref passed");
                return;
        }

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

produces this output:

```
4:  5 7 3 9
3:  6 7 3
```

A very powerful aspect of Java arrays is that a multidimensional array need not have the same size for each element of a dimension. For example, we can have a two-dimensional array in which some rows have more columns than others. (As a result, rows in a multidimensional array need not be stored in any predictable physical order relative to each other; however, components within the same row are stored contiguously.) For example:

```
double[][] vv = {
        {},
        {2.6, 9.2, 3.3},
        {1.2},
        {5.6, 9.8}
};
```



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

As we can see from the following output, each row has a different number of columns:

```
length of vv    = 4
length of vv[0] = 0:
length of vv[1] = 3:  2.6 9.2 3.3
length of vv[2] = 1:  1.2
length of vv[3] = 2:  5.6 9.8
```

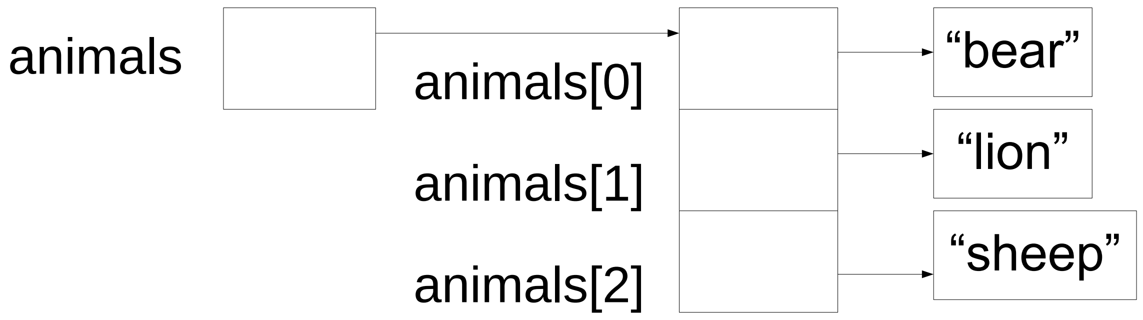
In the initializer above, the allocation of memory using new is implicit. Let's look at what the compiler is really doing here by initializing another array, called vv2, in exactly the same way as vv:

```
double[][] vv2 = new double[4][];

vv2[0] = new double[0];
vv2[1] = new double[3];
vv2[1][0] = 2.6;
vv2[1][1] = 9.2;
vv2[1][2] = 3.3;
vv2[2] = new double[1];
vv2[2][0] = 1.2;
vv2[3] = new double[2];
vv2[3][0] = 5.6;
vv2[3][1] = 9.8;
```

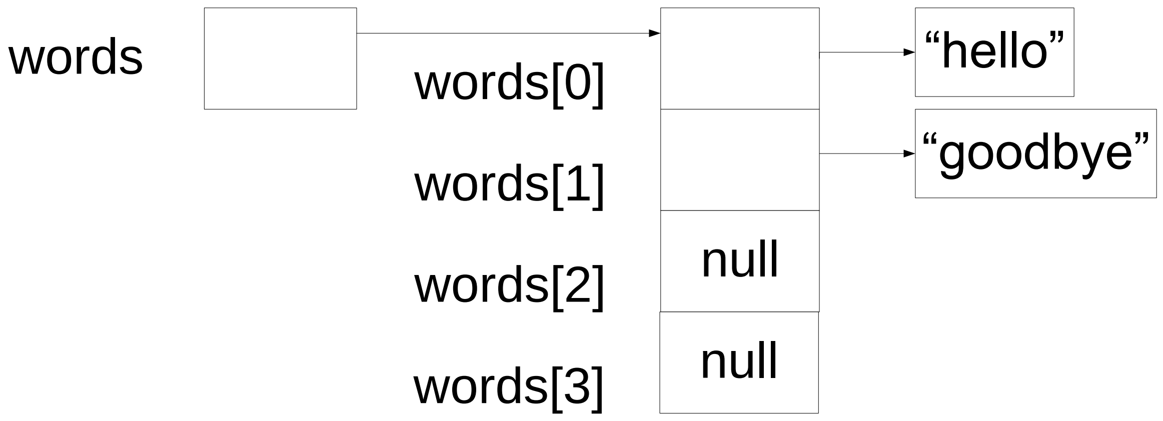
Unlike some languages, Java does not support partial initialization of an array with default values being applied to the rest. In Java, the number of initializers exactly determines the number of elements.

All the examples thus far in this section have involved arrays of primitive types. Of course, Java also supports arrays of reference types. For example:

```
String[] animals = {"bear", "lion", "sheep"};
```



```
String[] words = new String[4];
words[0] = "hello";
words[1] = "goodbye";
```



```
System.out.println("words[0] = " + words[0]);
System.out.println("words[3] = " + words[3]);
```

The output produced is:

```
words[0] = hello
words[3] = null
```

When an initializer list is provided, things are quite straightforward, however, in the case of `words`, we are allocating four references to strings, *not* four Strings themselves. For that, we need to use `new` on each component of `words`, individually. As we can see by the output, the two components not explicitly initialized (1 and 2), take on a default value of `null`.

When we pass an array to a method, what we are really passing is a reference to that array, and that reference is passed in by value. However, since the reference copy refers to the same array, changes made to elements through the copy do change the array. For example:

```
dump1DIntArray(counts);
change1DIntArray(counts);
dump1DIntArray(counts);

private static void change1DIntArray(int[] x)
{
        for (int i = 0; i < x.length; ++i)
        {
                x[i] += 10;
        }
}
```

The output produced is:

```
4:  5 7 3 9
4:  15 17 13 19
```

From the very first example in §1, we have been defining `main` as a method that takes an array of strings as an argument. Now that we know about arrays and strings, just what is the purpose of this argument? Most systems allow what are called *command-line arguments* to be passed to a program when it begins execution. The following example simply displays all the strings passed to it at program start-up (see directory Rf09):

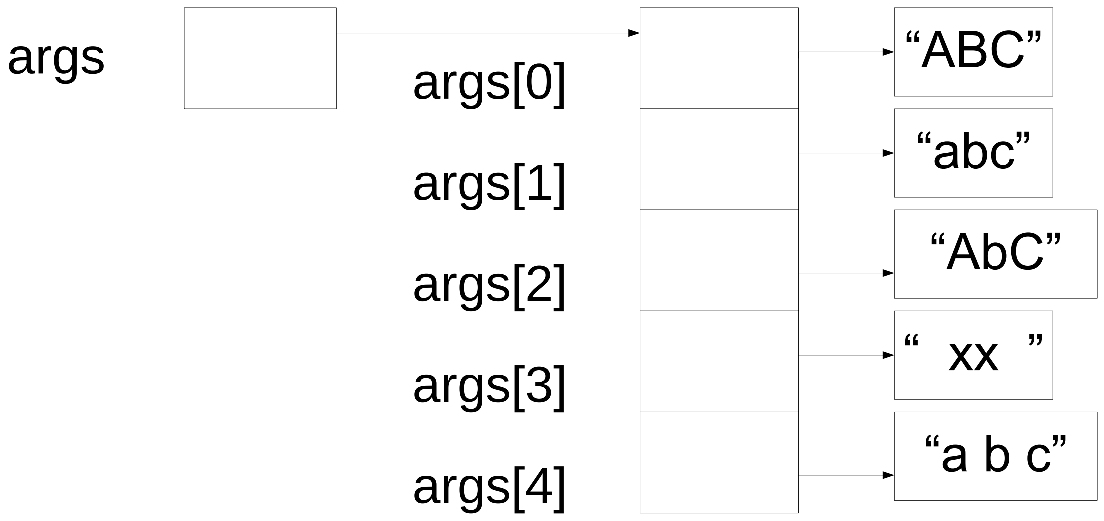
```
public class Rf09
{
        public static void main(String[] args)
        {
                for (int i = 0; i < args.length; ++i)
                {
                        System.out.println("args[" + i + "] = >" + args[i] + "<");
                }
        }
}
```

When this program was run on one system using the following command line:

```
ABC abc AbC "  xx  " "a b c"
```

the output it produced was:

```
args[0] = >ABC<
args[1] = >abc<
args[2] = >AbC<
args[3] = >  xx  <
args[4] = >a b c<
```



Since the behavior of command-line processors varies from one system to another, the output may vary. For example, this system stripped off double quotes, treating them as argument delimiters, allowing leading, trailing, and/or embedded white space to be made part of an argument. This system also preserved the case of non-quoted arguments; others might not unless they are enclosed in quotes. Systems that use some sort of delimiter often provide a way to include that character in an argument as well. Unlike some other languages, the name of the program being run is *not* made available through the argument list.

While most programmers use `args` as the name of the formal parameter in `main`, any identifier will do; it's just a local variable name.

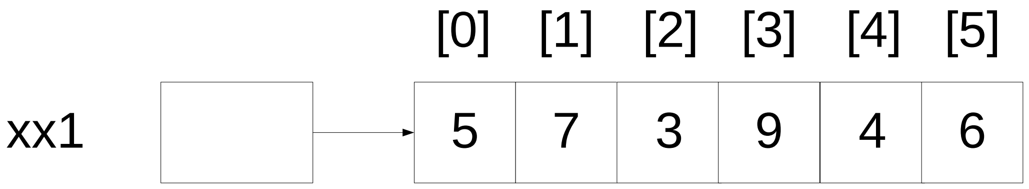
Exercise 4-3: Find out how to run a program on your system so that command-line arguments can be passed. Does your implementation preserve the casing of command-line arguments? Is white space embedded in double (or some other form of) quotes preserved? Can the delimiter characters be embedded inside an argument?

Exercise 4-4*: Have a program take each of its command-line arguments, pass them through the method `compressString` from the previous exercise, and display the compressed String. (See lab directory Lbrf03.)

4.7 Copying Arrays

At times, it is useful to be able to make a copy of a whole array, and since assigning one array reference to another copies only the reference, we need some other means of actually copying the array's elements. The following series of examples (each taken from the file Rf10.java) shows how. Let's begin by making a copy of a one-dimensional array of `int`:

```
int[] xx1 = {5, 7, 3, 9, 4, 6};
int[] xx2 = new int[xx1.length];
System.arraycopy(xx1, 0, xx2, 0, xx1.length);
```

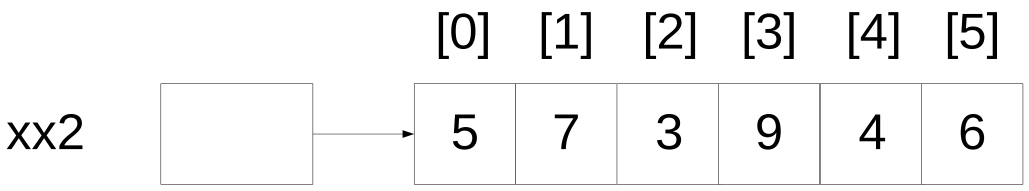


The output obtained from `dump1DIntArray(xx2)` is:

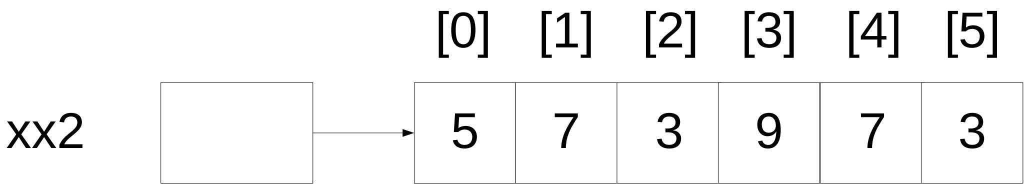
```
6:  5 7 3 9 4 6
```

indicating that the new array is an exact copy of the old. The method `arraycopy`, defined in class `System`, takes five arguments. In order, these are reference to the source array, the start index of the source array, reference to the destination array, the start index of the destination array, and the number of consecutive elements to copy. Invalid arguments cause exceptions. For example, the source and destination must be references to arrays of one dimension and have the same element type.

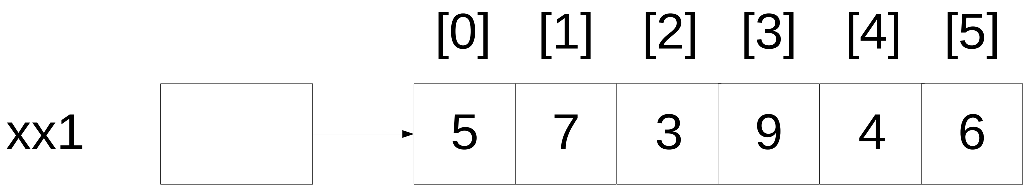
Not only can we copy one array to another, we can also copy from one part of an array to another. For example:



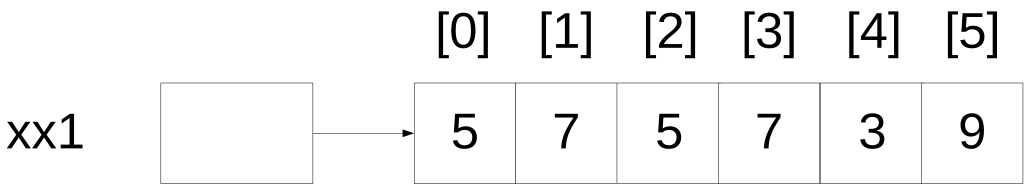
```
System.arraycopy(xx2, 1, xx2, 4, 2);
```



This results in xx2 containing the values 5, 7, 3, 9, 7, and 3. If the source and destination arrays overlap, the copy will be done as if it an intermediate storage area were used. For example:



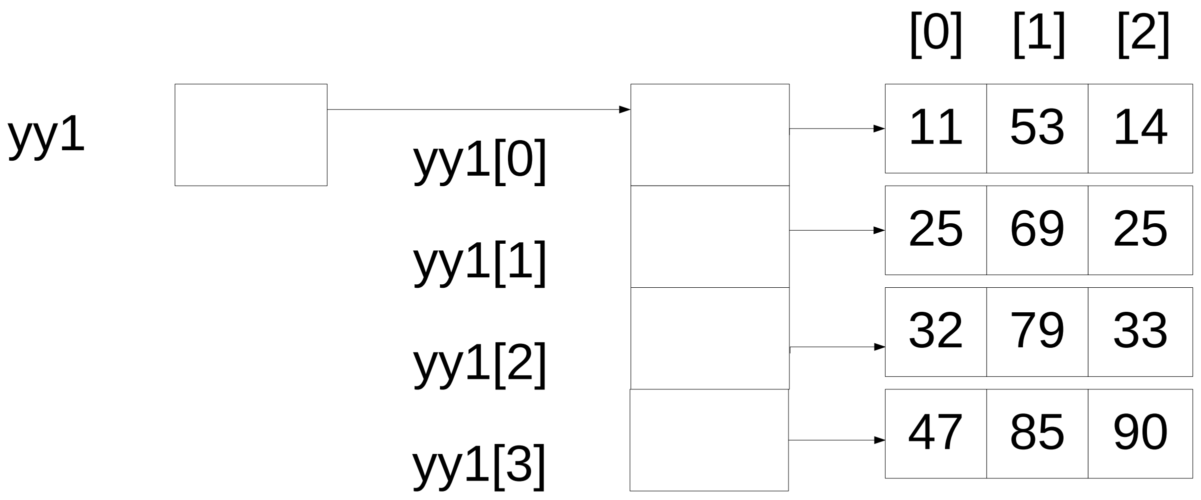
```
System.arraycopy(xx1, 0, xx1, 2, 4);
```



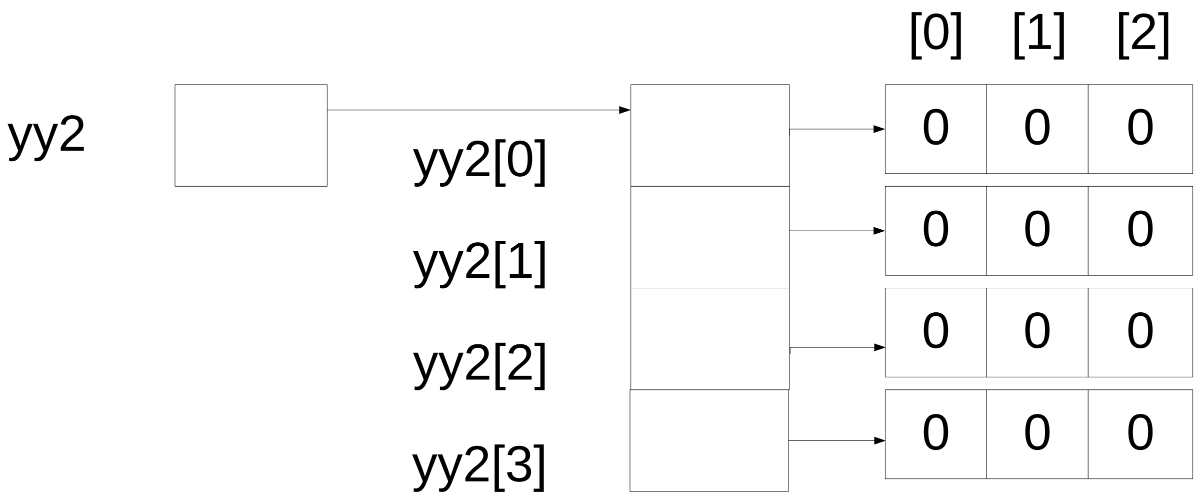
which results in xx2 containing the values 5, 7, 5, 7, 3, and 9.

Let's look at how we might copy a two-dimensional array of `int`:

```
int[][] yy1 = {
        {11, 53, 14},
        {25, 69, 25},
        {32, 79, 33},
        {47, 85, 90}
};
```



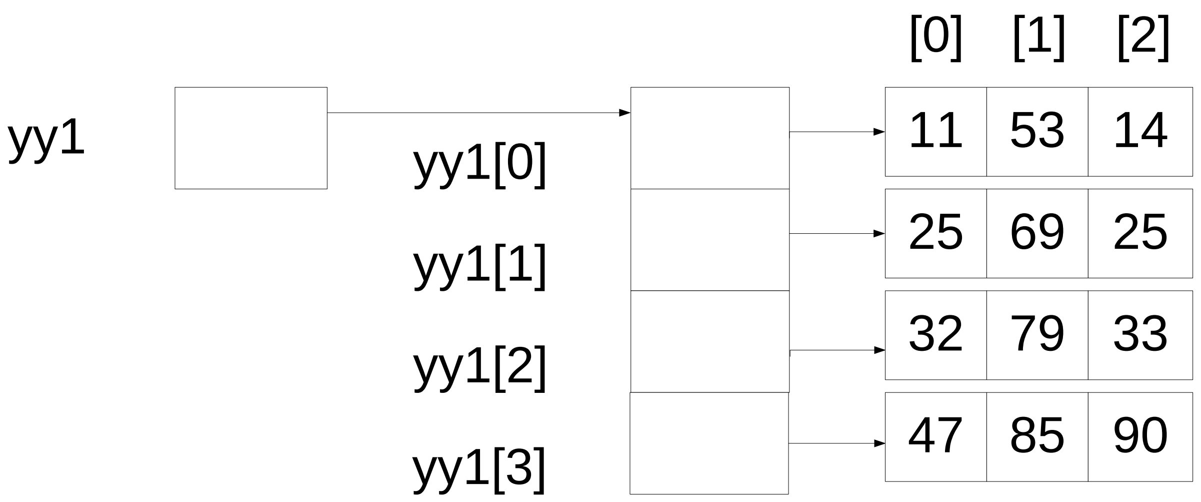
```
int[][] yy2 = new int[yy1.length][yy1[0].length];
```



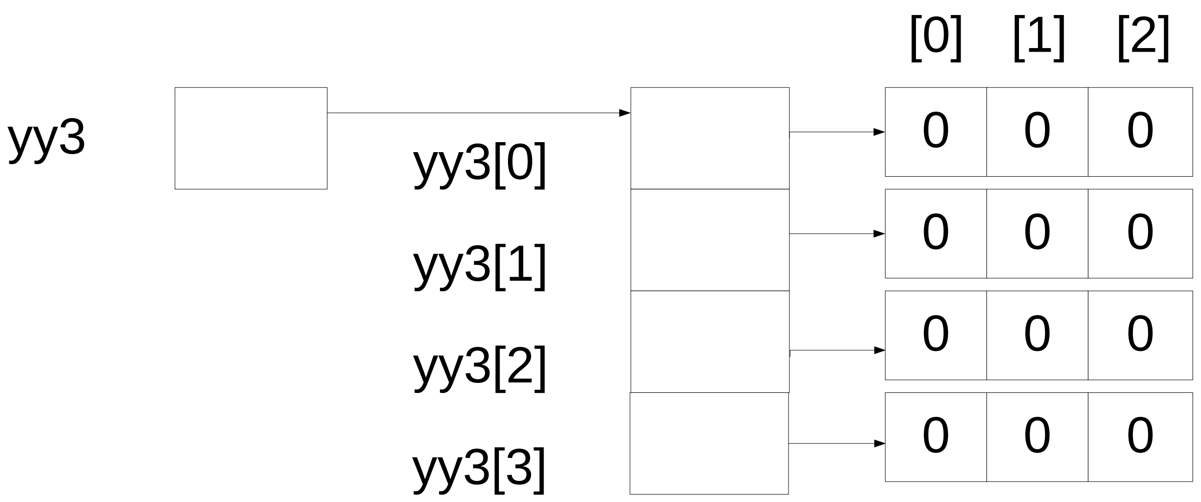
```
for (int i = 0; i < yy1.length; ++i)
{
        System.arraycopy(yy1[i], 0, yy2[i], 0, yy1[i].length);
}
```

First, we must allocate room for a copy of the two-dimensional array. Then **we must copy each row separately**. (In this case, each row has the same number of columns. If that were not the case, we'd have had to allocate memory for each row separately.) Now if we change the value of any element of yy1, those changes are not reflected in yy2.

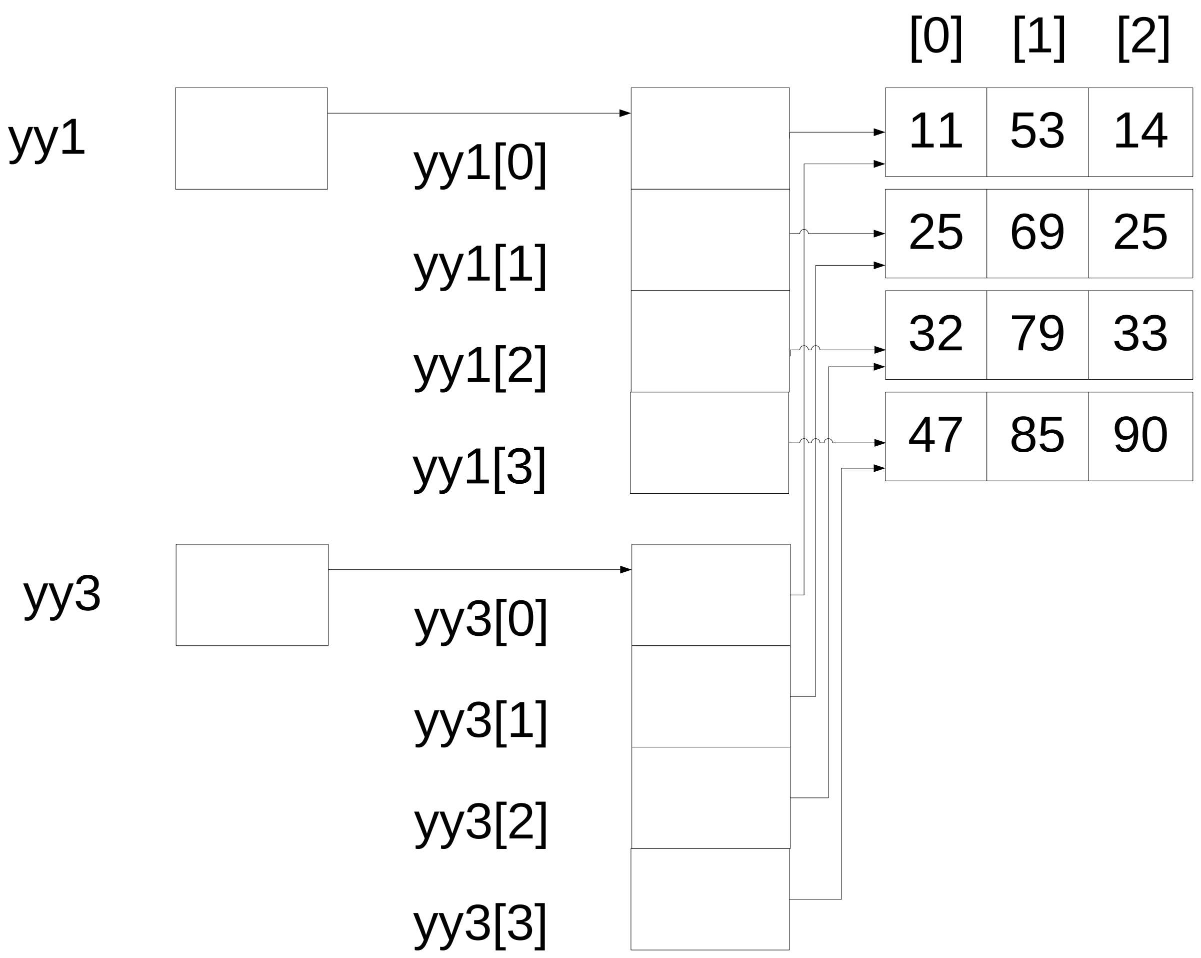
As we mentioned earlier, simply copying an array of references is insufficient. For example:



```
int[][] yy3 = new int[yy1.length][yy1[0].length];
```



```
System.arraycopy(yy1, 0, yy3, 0, yy1.length);
```



```
yy1[0][2] = -6;
yy1[1][1] = -7;
yy1[2][0] = -8;
yy1[3][2] = -9;
```

In this case, the changes made to the elements of yy1 are reflected in yy3 which when displayed contains:

```
11 53 -6
25 -7 25
-8 79 33
47 85 -9
```

As we should expect, copying an array of objects involves a bit more work than an array of primitive variables:

```
String[] zz1 = {"bear", "lion", "sheep"};
String[] zz2 = new String[zz1.length];
for (int i = 0; i < zz1.length; ++i)
{
        zz2[i] = new String(zz1[i]);
}
```

As before, we allocate memory for an array, but this is for the array of references. Then we must allocate memory for each string. And since we want each new string to be initialized with the corresponding element in the other array, we pass the constructor that string's value.

If strings are read-only, why do we need to make copies of them? Couldn't we just have one array with many references leading to it? Yes, we can, but it depends on what we are doing with the array that determines which approach we take. For example:

```
String[] zz1 = {"bear", "lion", "sheep"};
String[] zz3 = zz1;
String[] zz4 = new String[zz1.length];
System.arraycopy(zz1, 0, zz4, 0, zz1.length);
```

`zz3` simply refers to the same array as `zz1`; however, `zz4` refers to another array that has the same contents as the first. Note carefully that we have copied an array of references here, not the actual strings themselves; however, there is no need to copy the strings since they are read-only, and we can share them with other references. Consider what happens when we change an element of `zz1`:

```
zz1[0] = "parrot";
System.out.println("zz3[0] = " + zz3[0]);
System.out.println("zz4[0] = " + zz4[0]);
```

The output produced is:

```
zz3[0] = parrot
zz4[0] = bear
```

In the case of the second reference to the same array, the change is reflected whereas in the case of the second copy, it is not.

```
zz1 = null;
System.out.println("zz3[2] = " + zz3[2]);
System.out.println("zz4[2] = " + zz4[2]);
```

The output produced is:

```
zz3[2] = sheep
zz4[2] = sheep
```

In this case, when `zz1` is set to the null reference, the reference count to the array it was referring to, is decremented. However, `zz3` still refers to that array so its contents are still available.

One final, but important, note about arrays; as well as declaring a primitive to be `final` we can also declare an array reference to be `final`. However, **a final reference to an array says nothing about the finality of the elements of the array being referenced.** In fact, Java provides no way to write-protect these elements. For example:

```
private final static int[] values = {10, 30, 50, 34};

values = new int[5];    // prohibited
values[2] = 0;          // allowed
```

Exercise 4-5*: Write a method that performs a circular rotation of the elements in a one-dimensional array of `int`. For example, if an array contains the following elements:

```
1 2 3 4 5 6
```

before rotation, after it has been rotated right by two positions it contains the following elements instead:

```
5 6 1 2 3 4
```

The method is declared as `static void rotate(int[] a, int count)`. If `count` is positive, the rotation is to the right with right-hand values wrapping to the left. If `count` is negative, the rotation is to the left with left-hand values wrapping to the right. Of course, a rotation of zero has no effect. Handle counts greater than the array length by using modulo arithmetic. For example, a count of 10 for an array of 8 is really a count of 2. Correctly handle an array containing zero elements. How can such an array be created? Test your method. (See lab directory Lbrf04.)

Exercise 4-6*: Implement a method called `createArray` that takes two arguments, an integer count and a string. This method creates and initializes a two-dimensional array of strings having count rows, in which the first row has 1 element, the second has 2, and so on. All elements contain the string passed in. For example, the call `createArray(3, "XY")` results in an array that looks like the following:

	0	1	2
0	XY		
1	XY	XY	
2	XY	XY	XY

Return this array to `main` and have `main` pass it to another method, `printArray`, which displays its elements, one row per line. (See lab directory Lbrf05.)

4.8 Variable-Length Argument Lists

Starting with JDK1.5, a method can be defined to have a fixed number of (zero or more) arguments, followed by a variable number of arguments. Consider the following example (see directory Rf16), that computes and returns the maximum value of a set of values passed to it and displays the result. The `FindMaximum` method has no fixed argument list, only a variable one:

```
public class Rf16
{
        public static void main(String[] args)
        {
/*1*/           double max = FindMaximum(new double[] {3.4, 5.3, -3.4});
                System.out.println("Maximum = " + max);

/*2*/           max = FindMaximum(139.21, 10.5, 100.2, 121.54, -156.32);
                System.out.println("Maximum = " + max);
        }

/*3*/   private static double FindMaximum(double ... args)
        {
                double maximumValue = Double.MIN_VALUE;

                for (double d : args)
                {
                        if (d > maximumValue)
                        {
                                maximumValue = d;
                        }
                }

                return maximumValue;
        }
}
```

The output produced is:

```
Maximum = 5.3
Maximum = 139.21
```

In case 3, we define `FindMaximum` to have a variable-length argument list by using the punctuator `...` after the type of the final (in this case the only) parameter. This tells the compiler to accept a single argument that is an array of objects of that parameter's type, or a comma-separated list of separate arguments each having that type (or a type derived from that type). In the latter case, the list of separate values is arranged by the compiler into an array, which is then passed in. So, either way, the method is given an array for the final parameter.

We have seen examples of calls to the `PrintStream` method `printf`. Clearly, this method can take an arbitrary number of arguments having arbitrary type. It achieves this by having the following declaration:

```
printf(String format, Object ... args)
```

As we will learn in §6.10, all reference types are derived from type `System.Object`, and all primitive types can be implicitly converted to that type. As such, type `Object` is the "one type that fits all!"

4.9 Primitive Wrapper Classes

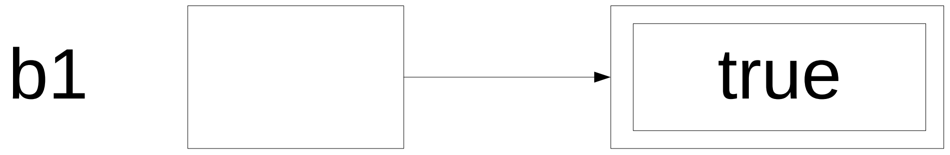
One disadvantage of new is that it cannot be used to allocate a single variable of a primitive type. Also, numerous library container classes (such as `Stack` and `Vector`) can only operate on objects, not on primitive variables. To get around these shortcomings, the standard library comes with a family of classes that parallels the primitive types. These classes are sometimes called *primitive wrappers* because an object of a wrapper class type simply contains a variable of the corresponding primitive type. Here is a list of the wrapper types:

Table 4-1: Primitive Wrappers

Class	Primitive Type
`Boolean`	`boolean`
`Character`	`char`
`Byte`	`byte`
`Short`	`short`
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`

The following example (see directory Rf11) demonstrates some of the methods these classes provide:

```
Boolean b1 = new Boolean(true);
```



```
Boolean b2 = new Boolean("no way");
Boolean b3 = new Boolean("True");

System.out.println("b1.booleanValue = " + b1.booleanValue());
System.out.println("b2.booleanValue = " + b2.toString());
System.out.println("b3.booleanValue = " + b3);
```

The output produced is:

```
b1.booleanValue = true
b2.booleanValue = false
b3.booleanValue = true
```

Each wrapper class provides at least two constructors: One takes a primitive-typed argument of the corresponding type, as in `Boolean(boolean b)`, `Character(char c)`, `Integer(int i)`, and `Double(double d)`; the other takes a string. In the case of `Boolean(String s)`, `s` is interpreted as being true only if it contains the string `true' spelled without concern for case. As shown by the output produced, `"no way"` indicates false while `"True"` is seen as true.

Each wrapper class also provides a method of the form *type*`Value()`, as in `booleanValue()`, `charValue()`, `intValue()`, and `floatValue()`. These methods simply return the value of the primitive variable contained in the wrapper object.

Let's consider each of the three expressions used in the `println` calls above: `b1.booleanValue()`produces a `boolean` value and when the compiler sees that that value needs to be concatenated with a string, it implicitly generates a call to the `toString` method (which all wrapper classes also define) concatenating the resulting string. In the second case, `b2.toString()` returns a string containing the internal value directly and concatenates that. Finally, `b3` results in an implicit call to `toString`.

Each wrapper class also defines an `equals` method that does the obvious thing:

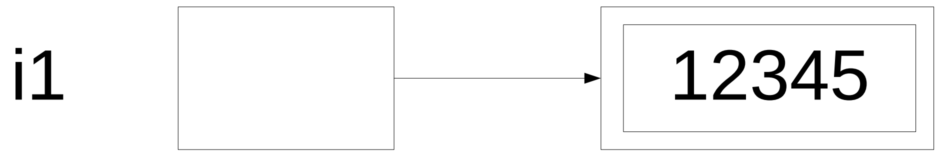
```
System.out.println("b3.equals(b2) = " + b3.equals(b2));
System.out.println("b3.equals(b1) = " + b3.equals(b1));
```

The output produced is:

```
b3.equals(b2) = false
b3.equals(b1) = true
```

In the numeric wrapper classes, the constructor taking a string expects that string to contain the text representation of a value of the given numeric type, however, no leading or trailing white space is permitted:

```
int i = 12345;
Integer i1 = new Integer(i);
```



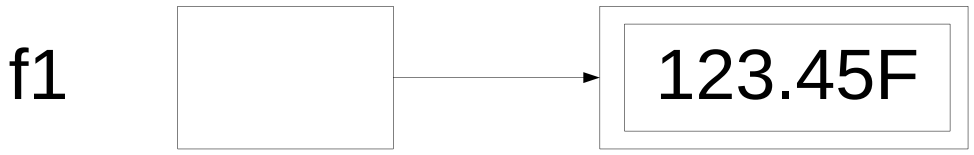
```
System.out.println("i1 = " + i1);
System.out.println("i1 = " + i1.intValue());

Integer i2 = new Integer("4567");
System.out.println("i2 = " + i2);
```

The output produced is:

```
i1 = 12345
i1 = 12345
i2 = 4567
```

The code continues:

```
Float f1 = new Float(123.45);
```



```
System.out.println("f1 = " + f1);
System.out.println("f1 = " + f1.floatValue());

Float f2 = new Float("987.654e13");
System.out.println("f2 = " + f2);
```

The output produced is:

```
f1 = 123.45
f1 = 123.45
f2 = 9.87654e+015
```

Earlier, it was stated that new cannot allocate memory for a single primitive variable. Strictly speaking, that's true; however, we can achieve the same effect by allocating memory for an array of 1 element. Since an array of any size (including 1) is handled using references, we can allocate a single variable. However, doing so requires us to use subscript notation from that point on, and that makes the code far less intuitive.

Starting with JDK1.5, Java permits the value of any primitive-type expression to be converted implicitly to its corresponding wrapper type. This process is called *boxing*. The resulting reference-type expression can be converted back again implicitly in a process known as *unboxing*. This makes it easier to deal with primitive-type values in a reference type context. For more information, see §6.13.

Exercise 4-7*: Write a program that accepts an arbitrary number of command-line arguments, each of which is an integer or floating-point value, and displays the sum of all these values. (See lab directory Lbrf06.)

Exercise 4-8: Look at the documentation for the wrapper classes to see the complete list of methods they provide. (Note that new methods were added with both JDK1.1 and JDK1.2/Java 2.)

4.10 Modifiable Strings

For many situations, Strings work just fine, however, sometimes it would be nice to be able to change the length and/or contents of a String without allocating a new one. (In fact, we cannot pass a String to a method and expect that method to change that string. While that method can change its copy of the reference to refer to a new String, it has no way of getting the original reference to refer to that new string.) While these operations are not permitted on objects of class `String`, they are permitted on objects of class `StringBuffer`. A StringBuffer can have its contents changed and it can shrink and grow on demand.

The following examples (see directory Rf12) introduce the operations permitted on StringBuffers:

```
StringBuffer sb1 = new StringBuffer();
System.out.println("sb1.capacity = " + sb1.capacity());
System.out.println("sb1.length   = " + sb1.length());
```

The output produced is:

```
sb1.capacity = 16
sb1.length   = 0
```

The constructor with no arguments causes a StringBuffer to be allocated that contains an empty character sequence. However, space is allocated for 16 characters by default. Each StringBuffer has a *capacity* and a current used length, which is always less than or equal to its capacity. As a StringBuffer grows, its capacity is expanded automatically as necessary.

```
StringBuffer sb2 = new StringBuffer(22);
System.out.println("sb2.capacity = " + sb2.capacity());
System.out.println("sb2.length   = " + sb2.length());
```

The output produced is:

```
sb2.capacity = 22
sb2.length   = 0
```

In this case, `sb2` is still empty but its allocated size is 22 characters.

```
StringBuffer sb3 = new StringBuffer("Hello");
System.out.println("sb3.capacity = " + sb3.capacity());
System.out.println("sb3.length   = " + sb3.length());
```

The output produced is:

```
sb3.capacity = 21
sb3.length   = 5
```

When a StringBuffer is constructed from a String, the allocation size is 16 more than the length of the String.

```
sb3.setCharAt(2, 'X');
sb3.setLength(10);
System.out.println((int)sb3.charAt(8));
sb3.setLength(3);
System.out.println("sb3 = " + sb3);
```

The output produced is:

```
0
sb3 = HeX
```

The `setCharAt` method allows the character with the given index to be set to the value specified. `setLength` can be used to extend the StringBuffer's length—in this case from 5 to 10—and the newly added characters each contain the value \u0000. This method can also be used to shorten the length, as shown, by truncating the StringBuffer. (Method `ensureCapacity` is used to ensure that the capacity of the StringBuffer is at least the size requested.)

```
sb1.append("Begin");
String s1 = sb1.toString();
System.out.println("s1 = " + s1);
sb1.append(true);
String s2 = sb1.toString();
System.out.println("s1 = " + s1);
System.out.println("s2 = " + s2);
```

The output produced is:

```
s1 = Begin
s1 = Begin
s2 = Begintrue
```

We can add extra characters to the end of a StringBuffer using `append`, whose argument can be any primitive type or String. At any time, we can take a snapshot copy and store it in a String, by calling `toString`.

```
sb1.append(1234);
sb1.append(4.56);
System.out.println("sb1          = " + sb1);
System.out.println("sb1.capacity = " + sb1.capacity());
System.out.println("sb1.length   = " + sb1.length());
```

The output produced is:

```
sb1          = Begintrue12344.56
sb1.capacity = 34
sb1.length   = 17
```

We can add extra characters at the front or middle of the StringBuffer using `insert`:

```
sb1.insert(0, 'X');
sb1.insert(6, false);
System.out.println("sb1          = " + sb1);
System.out.println("sb1.capacity = " + sb1.capacity());
System.out.println("sb1.length   = " + sb1.length());
```

The output produced is:

```
sb1          = XBeginfalsetrue12344.56
sb1.capacity = 34
sb1.length   = 23
```

The `reverse` method reverses the StringBuffer, in place:

```
sb1.reverse();
System.out.println("sb1 = " + sb1);
```

The output produced is:

```
sb1 = 65.44321eurteslafnigeBX
```

The String class has a constructor that takes a StringBuffer. That along with the `toString` method in class `StringBuffer`, allows us to convert either form of string object to the other:

```
String s3 = new String(sb1);
System.out.println("s3 = " + s3);
```

The output produced is:

```
s3 = 65.44321eurteslafnigeBX
```

Exercise 4-9: Look at the documentation for the classes `String` and `StringBuffer` to see the complete list of methods they provide.

Exercise 4-10*: Without changing its signature, re-implement `compressString` such that it preallocates a `StringBuffer` rather than using string concatenation. (See lab directory Lbrf07.)

4.11 Static Initialization Blocks

We have seen that static variables can be initialized using non-constant expressions; however, sometimes they need to be initialized in such a way that cannot be expressed in a single expression. For example, how can we initialize a static array of primitives or objects? Here's an example (see directory Rf14):

```
public class InventoryData
{
/*1*/   private static final int[] monthlyTotal = new int[12];
/*2*/   private static final int AVERAGE_MONTHLY_TOTAL;

/*3*/   static {
                int total = 0;

                for (int i = 0; i < monthlyTotal.length; ++i)
                {

                        // initialize monthlyTotal[i] some how

                        total += monthlyTotal[i];
                }
/*4*/           AVERAGE_MONTHLY_TOTAL = total/monthlyTotal.length;
        }

        public static void display()
        {
                // …
        }
}
```

The initializer in case 1 allocates the memory for the array, each of whose elements takes on the default value, zero. Since the actual initial values must be obtained at runtime (by reading a file, for example), we need some way to achieve that without our having to remember to call some initialization method once and only once. We do this in case 3 by using what is called a *static initialization block*. Such a block can contain local variable definitions and executable statements just like a method. Multiple static initialization blocks are permitted and along with the static field member initializers, they are executed in their order of appearance in the source file.

In JDK1.0, a class variable could only be declared `final` if it contained an initializer; however, that restriction was lifted in JDK1.1, allowing such a variable to be initialized by some other means at runtime.

In case 1, if the number of monthly totals were not known in advance, we could omit the initializer expression altogether and allocate the memory for the array inside the static initializer block once that number was known.

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 §1.10, 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, given the way Point has been defined, 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.

We have already seen examples of overloaded methods within a class, so it should come as no surprise that we can overload constructors. In this example, we provide a second constructor, which takes no arguments and creates the new Point at the origin.

Like variables, methods not having the static modifier are instance methods, so they can only be called to operate on an instance of their parent class:

```
// instance methods
        public void move(int xor, int yor)      // called using p.move(1, -4)
        {
                x = xor;
                y = yor;
        }

        public void translate(int xor, int yor) // called using p.translate(2, 3)
        {
                x += xor;
                y += yor;
        }

        public boolean equals(Point p)          // called using p1.equals(p2)
        {
                if (p == null)
                {
                        return false;
                }
                return (x == p.x) && (y == p.y);
        }

        public String toString()
        {
                return "(" + x + "," + y + ")";
        }
}
```

The methods move and translate are self-explanatory. Note the test for null in equals; without this, attempting to select the x or y member in the Point referred to by p when no Point exists, will cause an exception to be thrown. As we can see, the method toString creates a string containing a printable representation of a Point. We have already seen that numerous library classes contain a method by this name and they perform the same function for their parent class. According to the formal description of toString, this method "creates a string representation of the object. In general, the toString method returns a string that 'textually represents' this object. The result should be a concise but informative representation that is easy for a person to read".

Note that the ordering of the definitions of the constructors and methods is arbitrary since the scope of a class member is the whole class, allowing forward references to constructors and methods.

Also, note that none of the constructors or methods is static, since each will be called to operate on a specific Point rather than on the class as a whole.

Now let's use these new constructors and methods:

```
public class Cl04
{
	public static void main(String[] args)
	{
/*1*/		Point p1 = new Point();
/*2*/		System.out.println("p1 = " + p1);

/*3*/		p1.move(5, 7);
		System.out.println("p1 = " + p1);

/*4*/		Point p2 = new Point(9, 1);
/*5*/		p2.translate(-4, 12);
		System.out.println("p2 = " + p2);

/*6*/		System.out.println("p1 equals p2 = " + p1.equals(p2));
/*7*/		System.out.println("p1 equals Point(5,7) = "
				+ p1.equals(new Point(5, 7)));
	}
}
```

In case 1, we create a Point using the constructor that takes no arguments. This uses the same syntax and has the same effect as the previous example in which we had no constructor defined. If no constructor is defined for a class, the compiler defines a *default constructor*, which initializes all of the instance variables to zero or null, depending on their type. However, once we provide a constructor with one or more arguments, the compiler will no longer automatically define one with no arguments, so we have defined one explicitly.

In case 2, we display the value of a Point using the same notation as we have for displaying the value of primitive variables. And, no surprise, the same process is being performed; the compiler creates a string from the value of the argument. In the case of an object, it calls the `toString` method for that class. So, we see then that a method by this name has a predefined meaning and it's called as necessary.

We call the `move` method for Point `p1` in case 3, then we create a Point using the constructor that takes two arguments, in case 4, and then translate that Point in case 5. Each time an instance method such as `move` and `translate` is called, it operates on the object for which it was called. For example, `p1.move(5, 7)` moves the location of Point p1 and `p2.translate(-4, 12)` translates the location of Point p2.

As we have seen in earlier examples, a number of library classes contain a method called `equals`, which compares two objects for equality, so we provide a version for class `Point`.[1] By definition, two Points are equal if they represent the same location. Case 7 is much like case 6 except that one of the Points being used is created "on the fly", as an unnamed temporary.

[1] While this version suffices for now, we'll show an alternate approach in §6.11.

The output produced by this program is:

```
p1 = (0,0)
p1 = (5,7)
p2 = (5,13)
p1 equals p2 = false
p1 equals Point(5,7) = true
```

When a class has multiple constructors, it is not uncommon for one of them to be defined in terms of another. For example, the constructor of the Point class taking no arguments can easily be turned into a call to the constructor taking two, as follows:

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

public Point()
{
        this(0, 0);     // call sibling constructor
}
```

We can invoke another constructor simply by using the notation `this(`*arg-list*`)`, where *arg-list* is an argument list whose number and argument types correspond to the signature for some other constructor for the same class. **If present, such a call must be the first statement of the constructor.** When the called constructor terminates, it returns to its caller, so we can have statements following such a call.

Exercise 5-1*: In §3.5, we implemented a crude model of a Circle along with a pair of methods called `initCircle` and `printCircle` (see labs directories Lbme02a and Lbme02b). Take one of these solutions and turn it into two separate files, one of which defines and implements a class called `Circle`, the other containing an application program that defines and initializes four Circle objects and displays their contents. Specifically, based on the four `initCircle` methods, create a set of constructors, and based on `printCircle`, create a version of `toString`. (See lab directory Lbcl04.)

5.3 Data Hiding within a Class

In all the methods we added to class `Point` above, we access the private fields directly by name. As such, any changes to the way in which they are represented can affect some, if not all, methods in that class. And there is no limit on the number of methods a class can have. If we agree that it makes sense to hide implementation details from methods outside a class, why not also protect them from most of the methods inside the class as well? Then, not only can changing a class's internal representation have no impact on existing programs that use that class, it can also have almost no impact on the methods in that class.

If class methods access private fields only indirectly, we can make the class immune to changes in the way in which the private fields are represented. We can provide indirect access to a private field's value via what is commonly called an *accessor*, a method whose name is of the form `get`*Attribute*. To provide indirect access to change a private field's value we use a *mutator*, a method whose name is of the form `set`*Attribute*.