Chapter Overview.
Now that you have some idea of program planning and object oriented programming, you’re ready for the fundamentals of Java programming. We give an introduction to Java applications, starting with writing results to the monitor screen, going on, among other things, to using for loops, performing calculations with primitive types, using wrapper classes, writing static methods and using prewritten ones from the Java library.

Contents

1  THE SHORTEST MEANINGFUL JAVA PROGRAM 3
2  RUNNING THE PROGRAM IN JAVA 5
3  THE int PRIMITIVE TYPE 7
4  VARIABLE NAMES 9
5  AN INTRODUCTION TO THE for LOOP 11
6  THE String CLASS 15
7  THE char PRIMITIVE TYPE 15
8  USING char VARIABLES AS LOOP VARIABLES 18
9  STATIC METHODS 18
10 THE NUMERIC PRIMITIVE TYPES 22
11 THE ORDER OF EVALUATION 24
12 THE final STATEMENT 26
13 THE WRAPPER CLASSES 28
14 THE toString METHOD 29
15 THE Math CLASS 32
16 JOptionPane CLASS 32
Java Chapter 2

17 THE boolean PRIMITIVE TYPE 34

18 SOLVING A PROBLEM 34
1 THE SHORTEST MEANINGFUL JAVA PROGRAM

Figure 2.1a displays the smallest Java program that produces output. Recall that a program consists of a class that can contain one or more methods. Here the class is called Prog1 and it contains only one method main. We save the program using the class name and java extension, here Prog1.java. Remember that according to convention, all class names should be capitalized. Thus anytime a word is capitalized in a statement, it will be a class name.

As indicated in \texttt{public static void main( String[]} \texttt{arg)} the name of the method is \texttt{main}. The term \texttt{public} is the \textit{access specifier}. In general, it indicates the accessibility of a method to other classes. The fact that the \texttt{main} method is \texttt{static} prohibits it from calling a non-static method in the same class by simply using the method’s name. The next term \texttt{void} is the \textit{return type} and it indicates that the method returns no information. The terms \texttt{String[] arg} will remain a mystery for the first few chapters except that we’ll say here that \texttt{arg} is an arbitrary choice of characters that you are allowed to use to represent a memory location and it is used to input information when a program is run from a system, like Unix, that uses a command line.

\textbf{MOTIVATION}

At this point you may wonder how the program communicates any results to the outside world. The \texttt{System.out.println} statement is one way it does this.

\texttt{System.out.println(" Hello John and Mary!");} enables the program to communicate with the outside world, by printing \texttt{Hello John and Mary!} on the monitor screen. What appears between the double quotes, here, \texttt{Hello John and Mary!} is called a "string" and with just the exception of when a $\backslash$ is used, is printed exactly on the screen just as it appears between the quotes. See Figure 2.1b. This is the first example of a statement. A statement is an instruction to the computer to perform some task. All statements end with a semicolon. Since \texttt{System} is capitalized, it is a class. In fact it is a class that is found in the API (Application Programming Interface) that comes with the SDK (Software Development Kit). Java is case-sensitive, so writing any part of a statement or method heading in the wrong case produces an error. Thus \texttt{system.out.println(" Hello John and Mary!")}, or for that matter, \texttt{Public static void Main( String[]} \texttt{asd)} produce errors since \texttt{system} should be capitalized and \texttt{Main} shouldn’t be. Using \texttt{out} directs the output to the monitor screen. Ordinarily you’d expect \texttt{out} to be a method of the \texttt{System} class. Actually, \texttt{out} is an instance of another class called \texttt{PrintStream} and \texttt{println} is a method of that class. This seems pretty complicated now; but since you will be using it in almost every program, writing it will become second nature to you. Note that the information (for now just one method) in a class is sandwiched between the brackets "{" and "}". Similarly, the information in a method (for now, just one statement) is sandwiched between another pair of these brackets.

We’ve written the statements and the closing ")" each on a separate line; and have indented the statements so that the program is easier to read. Some integrated development environments (IDEs) do this automatically for you. To save space, you could compress the program into two lines, as is shown in Figure 2.1c; However, the program would be much more difficult to read and understand.
A SIMPLE JAVA PROGRAM

```java
public class Prog1
{
    public static void main( String[] arg)
    {
        System.out.println(" Hello John and Mary!");
    }
}
```

Figure 2.1a. The class Prog1 contains one method, main which does not return any information, hence its void. The static means that in main you could not call any non-static method in the same class simply by using the method’s name. The method contains one statement. If "[]" is omitted after String or main is capitalized, the virtual machine will issue the seemingly cryptic execution message: Exception in thread "main" java.lang.NoSuchMethodError: main meaning that it cannot find the main method in the prescribed form. A Java application must have a main method in order to be executed.

Hello John and Mary!

Figure 2.1b. Running the program of Figure 2.1a. The println prints the string Hello John and Mary! on the monitor screen.

A TERRIBLE WAY OF WRITING A PROGRAM

```java
public class Prog1{ public static void main( String[] arg)
{ System.out.println(" Hello John and Mary!");} }
```

Figure 2.1c. If the program were written on two lines, it would compile. It would, however be difficult for a human to understand.
After the computer executes a `println` statement, it prints what is indicated and then the cursor goes to the beginning of the next line. A more technical explanation is that after the computer executes a `println` statement, it performs a carriage return (moves cursor to the beginning of the line) and a line feed (moves up the information on the screen, one line vertically). Thus `println("one"); println("two");` in Figure 2.2a produce output on two successive lines as shown in Figure 2.2b.

The `print` method is another method of the `PrintStream` class. When it is executed, the cursor remains where it was after the last character is printed. Thus after the computer executes `System.out.print("My name")`, the cursor is one column to the right of the "e". When the computer executes the next output statement, for instance, `System.out.print("is Ivan")`, it prints the string `is Ivan` on the same line on which `My name` was printed. The result is `My name is Ivan` (see Figure 2.2b).

### MOTIVATION

At this point, you may want to type your own program and run it. Unless we show you how to run it in Java, your program will just sit there and do nothing.

---

## 2 RUNNING THE PROGRAM IN JAVA

Execute the following steps in order to run the program. We assume that you have downloaded from the Sun web page the Software Development Kit (SDK). It contains the java compiler (it's called `javac`), the interpreter (it's called `java`) that executes your compiled program and the Java library.

1. If you are working on a pc, use an editor like `Notepad` to write your program or if you are using the Unix system use `emacs` or `vi`. Remember, in order to compile and run `class Prog1` we must save our program as `Prog1.java`. If we save it as anything else, it will not compile.

2. The next step is to compile your java program into java bytecode so it can be interpreted by the virtual machine. If you are using a pc, click the MS-DOS Command Prompt icon to get a screen that has the command line prompt `c:\>` and a blinking cursor. On either a pc or Unix system, change the directory to the subdirectory you want your program to be in, for instance, `cd programs`.

   In Appendix A, we describe how to include the subdirectory containing `javac` and `java` in the `PATH` variable. This will enable you to process your programs in any directory. Next type `javac Prog1.java` at the command line. If there are any grammatical errors, they are called `syntax` errors, the compiler will indicate them. Correct these errors and recompile your program. A successful compilation produces java bytecode that will, in our case, be stored in `Prog1.class`.

2. When you type `java Prog1`, the java virtual machine is launched and the `println` statement will display the results on the screen. If you include the `class` extension by typing `java Prog1.class`, an error will occur. If the `CLASSPATH` variable is not blank, you must alter it so the java interpreter knows where the `class` file is. Appendix A describes how to do this.
print vs. println

public class Printing {
    public static void main(String[] arg) {
        System.out.println("one");
        System.out.println("two");
        System.out.print("My name");
        System.out.println(" is Ivan");
    }
}

Figure 2.2a After a println is executed, a carriage return and line feed is executed. After a print1, however, the cursor remains on the same line.

one
two
My name is Ivan

Figure 2.2b. Running the program of Figure 2.2a

EXECUTING A JAVA PROGRAM

student@dept % javac Prog1.java
student@dept % java Prog1

Hello John and Mary!

Figure 2.3. The javac compiler produces java bytecode stored in Prog1.class. The java command interprets the java bytecode. The word java is followed by the class name without the extension. The session shown here is on a Unix system (where the prompt is student@dept %). On a pc, it's the same except that the prompt is c:\>.
3 THE int PRIMITIVE TYPE

In order to write more complicated programs, you must store data in the computer’s memory locations, called variables and later retrieve and use them. In Figure 2.2 we display a program that calculates the area of a rug. Before you store data you must indicate how the data should be stored. The first type of data we discuss is the int (it stands for integer, a number without a decimal point, e.g., 15 or -9). It belongs to the primitive type family that consists of the numeric types (numbers and characters); and the boolean type (true and false values).

To store int values in memory locations, you must list the names of these locations in a variable declaration beginning with int. It is best to assign the initial values to the variables here, as in int length = 20; The int declaration of these variables must precede their use in other statements. One or more blanks must separate int from the first variable. The assignment begins with a variable – here the int variable length – followed by the symbol for assignment, "=", and then by what you want to place in the location named by the variable – here 20, as is shown in Figure 2.4a. The other assignments in the program follow the same form. Thus in the next int declaration int width=15, area; we make an assignment to width and list another variable area that will be assigned a value later. When more than one assignment and or variable appear here they form what is called a "list". The items in it are separated by commas and it is terminated with a semicolon.

Because you must specify the type of each variable used in your program, Java is called a "strongly typed" language. How the assignments are made during execution is shown in the Table for Figure 2.4a. The first line

<table>
<thead>
<tr>
<th>Statement</th>
<th>length</th>
<th>width</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td>length = 20</td>
<td>20</td>
<td>Uninit</td>
<td>Uninit</td>
</tr>
</tbody>
</table>

shows that although length has been assigned a value, width and area have not been assigned values – they are thus uninitialized (we use the abbreviation "Uninit" here). The second line shows that the computer remembers what has been stored in length, and that it stores 15 in width.

<table>
<thead>
<tr>
<th>Statement</th>
<th>length</th>
<th>width</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td>width = 50</td>
<td>20</td>
<td>15</td>
<td>Uninit</td>
</tr>
</tbody>
</table>

The statement, area = length*width; following the int definitions assigns a value to area. It is called an "assignment" statement; it uses the Java symbol for the multiplication operator, * and instructs the computer to take the value stored in length, multiply it by the value stored in width and assign the result to the location, area, as is shown in the third line of the table. See Figure 2.4b for the execution results.

<table>
<thead>
<tr>
<th>Statement</th>
<th>length</th>
<th>width</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td>area = length*width</td>
<td>20</td>
<td>15</td>
<td>300</td>
</tr>
</tbody>
</table>

The value placed in a variable can change, as can be seen when width is assigned a new value in width = 10. A memory location can store one value at a time. When we assign it a new value, the bits constituting the binary number stored in the location are reconfigured and the original value is lost. We see from the table that the value of area is not changed until area = length*width is executed again.

The program begins with a few lines starting with //. The "/" indicates that what follows on the line is a comment. Comments are not processed but are used to inform the reader what a particular program or method does. In order to maintain the facing-page format of the book, in the future we won’t use comments to indicate the author and date the program was written.
USING THE int PRIMITIVE TYPE

//program written by S. Marateck 11/17/02

//Calculates area of rugs and introduces the
//use of int variables.

public class RugArea

// introduction to primitive type int

{ public static void main( String[] arg)
{
    int length =20;
    int width=15, area;
    area = length*width;
    System.out.println( "L = " + length + ", W = " + width + ", area = " + area);
    width = 10;
    area = length*width;
    System.out.println( "L = " + length + ", W = " + width + ", area = " + area);
}
}

Figure 2.4a. The value 20 is assigned to the int variable length. In a declaration, the type of
the variable is given and a value can also be assigned to it. In the println, because the "+" is
sandwiched between the string and the int variable, the "+", causes the value the of int area to
be converted to a string and then the string is appended to the end of the string area. This is
called concatenation.

<table>
<thead>
<tr>
<th>Statement</th>
<th>length</th>
<th>width</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td>length = 20</td>
<td>20</td>
<td>Uninit</td>
<td>Uninit</td>
</tr>
<tr>
<td>width = 15</td>
<td>20</td>
<td>15</td>
<td>Uninit</td>
</tr>
<tr>
<td>area = length*width</td>
<td>20</td>
<td>15</td>
<td>300</td>
</tr>
<tr>
<td>width = 10</td>
<td>20</td>
<td>10</td>
<td>300</td>
</tr>
<tr>
<td>area = length*width</td>
<td>20</td>
<td>10</td>
<td>200</td>
</tr>
</tbody>
</table>

The Table for Figure 2.4a. Shows how the values are assigned to the variables as the execution
progresses. When width is given a new value, the value of area doesn’t change until area =
length*width is executed again.

L = 20, W = 15, area = 300
L = 20, W = 10, area = 200

Figure 2.4b. Running the program of Figure 2.4a
When the program is compiled, an uninitialized variable on the right-hand side of an assignment statement produces a compiler error. So if width had not been initialized, a compilation error occurs at area = length * width. No java byte output is generated until your program is free of compilation errors.

In System.out.println( "area = " + area); because the "+" is sandwiched between a string and a numerical value, the computer first converts the value of the int area to a string, "300" and then combines it with the string "area = " giving "area = 300". Combining two strings is called concatenation. When, however, a "+" is sandwiched between two int variables or values, the computer adds the two values 1. For instance in int sum = 2 + 3, the value 5 is assigned to sum. The "+" and is an example of operator "overload", that is, an operator can have more than one function and its function depends on the context in which it is used. If you wanted to add two numbers and concatenate the sum with a string that precedes it, you would have to indicate that you wanted the numbers added first. This is done by placing the addition in parentheses and then concatenate the sum with a string. Operations in parentheses are done first. For example, System.out.println( "The sum is " +(3+5) ); produces The sum is 8. If the parentheses are omitted as in System.out.println( "The sum is " +3+5 ); the result is The sum is 35 because the "+" operates from left to right. First the string is concatenated with 3 and then the resulting string is concatenated with the 5. Thus it should be obvious that System.out.println( 3+5+" is the sum"); produces 35 as the sum, since the addition is done first. We can summarize all of this by noting that in a + b + c, if a is a string and b and c are numerical values, or if c is the string and a and b are the numerical values, then the "+" is not associative, i.e., (a + b ) + c does not equal a + (b + c). If you wanted to add one to the value of j and store the result back into j, you would write j = j + 1; This can be abbreviated j++.

If an int value contains one or more of the following characters, it is an illegal int value and will cause a syntax error: a decimal point; a comma; any other non-numeric character (e.g., the $ in $56); or a blank that follows one of the digits and precedes the next digit in the number, e.g., 45 67. A blank, such as the one appearing here, is called an embedded blank.

4 VARIABLE NAMES

There are rules that must be followed in forming variable names. The same rules apply to forming names for methods and classes and other programming items. All of these names are called identifiers. It’s important to distinguish at this point what is allowed in an identifier and the subset of this that consists of the convention used is in forming them. Here is what is allowable: Identifiers may begin with a letter or an underscore (_), and can be followed by these characters as well as by digits. The convention, however, is that identifiers should begin with a lowercase letter and if the identifier consists of two words, the second one should be capitalized. For example, if the two words are old length, the identifier would be oldLength. Using the rules for forming an identifier, we see that length1 is legal; whereas 2length is not. Identifiers should describe what they represent. Thus oldLength describes a previously assigned length. Table 2.1 shows examples of legal identifiers and in Table 2.2, examples of illegal ones. Table 2.2 is a list of words that are reserved for a specific use in Java. Using them as identifiers will cause a compiler error.

1If there is a string value in the expression, the integer values may be concatenated, as we will see later in the paragraph.
IDENTIFIER NAMES

<table>
<thead>
<tr>
<th>Valid identifier names</th>
</tr>
</thead>
<tbody>
<tr>
<td>newLength</td>
</tr>
<tr>
<td>oldLength</td>
</tr>
<tr>
<td>obj</td>
</tr>
<tr>
<td>obj1</td>
</tr>
</tbody>
</table>

Table 2.1 Identifier names should describe what they represent. They must begin with a letter or underscore. They can also contain digits but no other characters. If an identifier consists of two words, e.g., old and length, convention dictates that you capitalize the second word, writing oldLength as opposed to using an underscore, writing old_length.

ILLEGAL IDENTIFIER NAMES

<table>
<thead>
<tr>
<th>bad variable</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>first obj</td>
<td>embedded blank</td>
</tr>
<tr>
<td>1obj</td>
<td>starts with digit</td>
</tr>
<tr>
<td>one$</td>
<td>illegal character, $</td>
</tr>
</tbody>
</table>

Table 2.2 Illegal identifier names and why they are illegal.

RESERVED WORDS IN JAVA

<table>
<thead>
<tr>
<th>abstract</th>
<th>else</th>
<th>interface</th>
<th>strictfp</th>
<th>while</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>extends</td>
<td>long</td>
<td>super</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>false</td>
<td>native</td>
<td>switch</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>final</td>
<td>new</td>
<td>synchronized</td>
<td></td>
</tr>
<tr>
<td>case</td>
<td>finally</td>
<td>null</td>
<td>this</td>
<td></td>
</tr>
<tr>
<td>catch</td>
<td>float</td>
<td>package</td>
<td>throw</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>for</td>
<td>private</td>
<td>throws</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>if</td>
<td>protected</td>
<td>transient</td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td>implements</td>
<td>public</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>import</td>
<td>return</td>
<td>try</td>
<td></td>
</tr>
<tr>
<td>do</td>
<td>instanceof</td>
<td>short</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>int</td>
<td>static</td>
<td>volatile</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3 These are called reserved words. They cannot be used as identifier names.
AN INTRODUCTION TO boolean VALUES

A non-numeric value is called a "boolean" value. It can be assigned to a boolean variable and can be either true or false, both written in lowercase. The relational operators ==, >=, <=, >, <, !=, shown in Table 2.4, are used to create boolean expressions. The simplest expression consists of a variable identifier or a value, such as j or 5. When involved in calculations, the identifiers or values are called operands. In general an expression consists of operands and operators. Thus j <= 5 is an expression consisting of the operands j and 5, and the operator <=. It is true if the value of j is less than or equal to 5. Typing operators consisting of two characters, e.g., <=, with an embedded blank (< =) causes an error. We'll see that boolean expressions enable us to write programming loops.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
</tbody>
</table>

Table 2.4

5 AN INTRODUCTION TO THE for LOOP

The simplest way to repeat a statement's execution is to place it in a for loop. The structure of the loop is that it begins with a for statement and is followed by a statement or a statement block. We remind you that a statement block begins with a "{" and ends with a "}". Let's say that we want to print "Ivan Smith" 3 times, we would write it as shown in Figure 2.5a

```java
for(int j = 0; j < 3; j++)
    System.out.println("Ivan Smith");
```

The computer interprets this as letting the value of the int variable j, it is called the loop index, vary from 0 to 2. In computer science it is a convention to let the initial value of the loop index, here j, be zero. A loop is executed while the value of the boolean expression, here j < 3, is true. Each time the loop is executed, the value of j is incremented by one due to j++. The variable j becomes undefined in the program after the println statement is executed. Thus

```java
for(int j = 0; j < 3; j++)
    System.out.println("Ivan Smith");
System.out.println("j = " + j);
```

would not compile. The compiler issues the message cannot resolve symbol and points to the right-most j in the second println statement. We say that the scope of j does not extend beyond the first println statement.

If you wanted to print an address label three times, you would write it, as shown in Figure 2.5c. Now the scope of j is defined between "{" and "}".
THE for LOOP

public class Looping
{
    static void main(String[] arg)
    {
        for(int j = 0; j < 3; j++)
            System.out.println("Ivan Smith");
    }
}

Figure 2.5a. The statement or statement block following the for is executed 3 times. The for consists of initialization (int j = 0); continuance condition (j<3); increment (j++).

Ivan Smith
Ivan Smith
Ivan Smith

Figure 2.5b. Running the program of Figure 2.5a.

public class AddressPrinter
{
    static void main(String[] a)
    {
        for(int j = 0; j < 3; j++)
        {
            System.out.println("Ivan Smith");
            System.out.println("1059 Nelson Avenue");
            System.out.println("New York, NY 10052");
            System.out.println();
        }
    }
}

Figure 2.5c. Now the statement block is executed thee times.

Ivan Smith
1059 Nelson Avenue
New York, NY 10052

Ivan Smith
1059 Nelson Avenue
New York, NY 10052

Ivan Smith
1059 Nelson Avenue
New York, NY 10052

Figure 2.5d. Running the program of Figure 2.5c.
If the initial value of the loop index is greater than its upper limit, the loop is not executed. For example in `for(int j = 4; j < 4; j++)` the initial value of `j` is 4 and its upper limit is 3, so `j < 4` is false and the loop is not executed.

The form of the for1 statement is for (initialization; continuance – condition; increment)

NESTED LOOPS

The next program, Figure 2.6a, draws a right triangle. To do this we place a loop within another loop. This is called nesting loops. The `k`-loop which is called the inner loop, is nested within the `j`-loop, the outer loop. For each value of `j` the value of `k` goes from 0 to that value of `j`. So when `j` is 0, `k` goes from 0 to 0, so one `x` is printed. Then the `println` is executed and the program performs a carriage return. Next when `j` is 1, `k` goes from 0 to 1, so two `x`'s are printed on the same line, and then the `println` is executed, all of which is shown in Figure 2.6b. Note that the `println` is outside the `k` loop, so it’s executed only once for each value of `j`. If we used `println('x')` instead of `print('x')` the output would have been a vertical line of `x`'s.
NESTED LOOPS

```java
public class Triangle
    // draws a triangle using print
{
    public static void main(String[] arg)
    {
        for(int j = 0; j < 5; j++)
        {
            for(int k = 0; k <= j; k++)
            {
                System.out.print("x");
            }
            System.out.println();
        }
    }
}
```

Figure 2.6a. The inner loop is executed for each value of \( j \), the outer loop index. These two indices must have different variable names. The `println` is executed once after each complete execution of the inner loop.

```
x
xx
xxx
xxxx
xxxxx
```

Figure 2.6b Running the program of Figure 2.6a
6 THE String CLASS

Another class found in the Java API is the String class. To assign the string, "thinking", to the String reference, one just write String one = "thinking". The reference one points to where the string is stored. Now all the methods of the String class can be applied to the object "thinking". For instance, one.length() gives the number of characters in the string, here eight. The indices or subscripts of the characters starts with 0 and goes to the value of length -1.

<table>
<thead>
<tr>
<th>t</th>
<th>h</th>
<th>i</th>
<th>n</th>
<th>k</th>
<th>i</th>
<th>n</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 2.5. Shows the elements of the string stored in the object referenced by one.

Thus the "g", the eighth character, is stored in position seven of the string. Another method of this class is charAt(n). This returns the character in the nth position. Thus one.charAt(7) returns "g". String objects once created cannot be changed. They are therefore called immutable. So you cannot for instance change any character in the string. Thus one.charAt(5) = 'a'; is not allowed. Another methods is the substring method. It has been written so that one.substring(0, one.length() ) gives all the string. Thus in our case, one.substring(0, 8) gives the whole string. In one.substring(2, 5) which yields "ink", the second parameter indicates the index of the last element in the substring (here 4) will have a value of one less than the parameter (here 5). If the second parameter is omitted, it is assumed to be the string length. So one.substring(0) prints the entire string. To construct a string two with all the letters in one in uppercase use two = one.toUpperCase() or all in lowercase use one.toLowerCase(). The expression one.equals(two) tests whether one and two contain the same string; whereas one == two only tests if they have the same address on the heap. A string can also be instantiated like other objects. So String one = new String("thinking") is equivalent to String one = "thinking". In order to convert an integer to a string, concatenate the integer with the empty string, e.g., String st = "" + 123, stores "123" in st. This works with any numerical type.

<table>
<thead>
<tr>
<th>method</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>returns number of elements in string</td>
</tr>
<tr>
<td>charAt(n)</td>
<td>returns character in element n</td>
</tr>
<tr>
<td>substring(n,m)</td>
<td>substring from n to m-1</td>
</tr>
<tr>
<td>toUpperCase()</td>
<td>changes each lowercase letter to uppercase</td>
</tr>
<tr>
<td>toLowerCase()</td>
<td>changes each uppercase letter to lowercase</td>
</tr>
</tbody>
</table>

Table 2.4. The URL at: http://java.sun.com/j2se/1.4/docs/api/index.html shows all the methods in the API classes. Here are some of the String methods.

7 THE char PRIMITIVE TYPE

Character values consist of one character or a control character sequence or something called the Unicode value, each sandwiched between two single quotes when they are assigned to a variable. A control character sequence issues commands to the printer head or loudspeaker. Character values are stored in variables of the char primitive type. Any character including a blank can be used as
a character value. Examples of these are a, A, 2, and $.

In older languages, a character was represented by a byte (8 bits). So you can have 2^8 or 256 bit combinations and thus can represent a total of 256 characters and control characters. Since the code starts at zero, Unicode values range from 0 to 65536. Associated with each of the first 128 characters or control characters is a numerical code called the ASCII (American Standard Code for Information Interchange) code. In Java, however, a character is represented by two bytes. This increases the number of characters that can be represented to 256^2 or 65536. Associated with each character or control character is a numerical code called Unicode. The first 128 characters of the Unicode form what is called the basic Latin subset of the Unicode and are identical to the ASCII code. The term given to the order of the characters in the code is the collating sequence. In any collating sequence, each successive digit is assigned a higher numerical code than the previous one, and these codes are contiguous.

Character values are integers with their value given by their ASCII (Unicode) code. We know that character values occupy two bytes of memory; whereas an int value occupies four bytes. Since the range of char values is narrower than that of ints, char values can be assigned to ints simply by using the assignment operator. So if char c; int i, you may write i = c. Writing c = i, however, produces a compilation error ”possible loss of precision” since the range of a char is narrower than the range of an int. To overcome this, precede the i with (char), i.e., c = (char)i. This is called casting the int as a char. What has been done is to convert an int into a character with the int’s value as its ASCII code. The Java compiler is not as restrictive, however, when parsing an assignment of an integer value to a char variable: If the integer value is less than or equal to 65535 (the Unicode’s highest value), casting is not necessary. So c = 97 will store the character a – its ASCII code is 97, in the variable c.

The decimal equivalent of the ASCII characters can be used in a program. There are two ways of converting a char value to its ASCII code. The first is by assigning the char to an int. The int value will be the char’s ASCII code. Thus i = ’2’ stores 50, the ASCII code for the character 2, in i. The second way is to use a character value as an operand with the numerical operations ”+” or ”-“. So println(’2’ + 2) prints 52 and println(’2’ + ’2’) produces 100. Since the ASCII code for ’0’ is 48, the assignment i = ’2’ - ’0’ stores 50 - 48 or 2, the actual value of ’2’ in i. Similarly, given char ch, then ch = (char)(ch + 1) adds 1 to the ASCII code stored in ch and reassigns it to ch. This can also be written ch++. Note that in ch + 1, since the char variable is included in a calculation, it’s converted into an int and thus the resulting sum must be cast in order to assign it to a char variable. To print the ASCII value of a character value, simply cast it. Thus println( (int)’2’ ) prints 50.

When a ”+” is sandwiched between a string and a char value or identifier, the char is converted to a string and is concatenated with the original string. Thus ”dogg” + ’y’ becomes doggy.

Examples of control character sequences that are formed by a slash followed by a single letter are shown in the Table 2.6. Since these sequences begin with a slash they are also called escape sequences because they were historically placed in the code by using the Esc (escape) key.

<table>
<thead>
<tr>
<th>Control character sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\n</td>
<td>new line</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
</tbody>
</table>
Table 2.6 The control characters must be formed by preceding the letters `b`, `n`, `r` and `t` with a backslash (`\`).

An example of a control character assignment to produce a new line is `char c = '\n'`. Control characters are mostly used in strings in print statements and when they are, they are not sandwiched between single quotes. For example, `System.out.println("char \tcode")` because of the control character `\t` prints `code` starting in column 9. Subsequent tabs skip to columns 17, 26, 35 etc, producing columns eight spaces wide. So if you wanted to print `a` under `char` produced by the previous `println` and 97 directly under `code`, you would use `System.out.println("a\t97")`. The slash can also be used in a string to indicate that the character following it be interpreted as the character itself and not part of an escape sequence or string delimiter. Thus to indicate a slash in a string, write `\`; and to indicate a double quote, write `"`. Thus `System.out.println("\"numerator\denominator\"")` prints "numerator\denominator". A Unicode escape sequence is formed by following a `\` with the letter "u" and following that by a four-digit hexadecimal number. For instance, the escape sequence `\u0007` is the Unicode representation for the ASCII code 7, the loudspeaker beep.
8 USING char VARIABLES AS LOOP VARIABLES

In Figure 2.19a we use the char variable letter as the loop index. The effect of letter++ is to increment the loop index’s ASCII code by one after each time the print is executed. In the second for loop, letter-- decreases the index’s ASCII code by one. The results of running the program are shown in Figure 2.19b.

9 STATIC METHODS

At times we may want access to a class that has a variety of utility methods that we can use without creating an object. In the next few programs we develop such a class. We can access information by dotting the class name with one of its static methods. The first method, public static void reverse(String s) shown in Figure 2.20a reverses the string parameter, so if s is "abcd", the method prints "dcba". To do this we concatenate each character in the string with a String variable t. We run the loop backwards starting with the last character in the original string. If the length of the string is len, the index of the last character is len - 1, in our case, 3. We thus write for(int j = len - 1; j >= 0; j--). The j-- decrements the loop index, as is shown in the Table for Figure 2.20a. The results are shown in Figure 2.20b.

The method reverse is called in the driver by Int.reverse(sample) where Int is the class name. If reverse were non-static this could not be done. If main is in the same class as reverse, since they are both static methods you could invoke reverse simply by writing reverse(sample) in main.

Since the String class is immutable, how does the concatenation work? Each time t = t + s.charAt(j) is executed, a new object is created pointed to by t. The garbage collector releases the portion of the heap pointed to by the previous t. We will show later in the text that this could be done in a more efficient way by using the StringBuffer class.

USING char VARIABLES

```java
public class Alpha
{
    public static void main(String[] arg)
    {
        //print alphabet forwards
        for( char letter = 'a'; letter <= 'z'; letter++)
            System.out.print(letter);
        System.out.println();
        //print alphabet backwards
        for( char letter = 'z'; letter >= 'a'; letter--)
            System.out.print(letter);
    }
}
```

Figure 2.19a In the compiler the letters are represented by their ASCII codes, so the loop is executed just like one in which the loop index is an int.
Figure 2.19b Running the program of Figure 2.19a. The second loop runs backwards because \texttt{letter--} decrements the loop index.

REVERSING A STRING–STATIC METHODS

```java
public class Int // consists of a static utility class
{
    public static void reverse(String s) // prints a string in reverse
    {
        int len = s.length();
        String t = ""; // initialize to empty string
        for(int j = len - 1; j >= 0; j--)
        {
            t = t + s.charAt(j);
            System.out.println(t);
        }
    }
}

class Driver
{
    public static void main(String[] arg)
    {
        String sample = "abcd";
        Int.reverse(sample);
        System.out.println();
    }
}
```

Figure 2.20a Each character is concatenated with \texttt{t1}. Since \texttt{reverse} is static we access it by dotting the class name with the method, \texttt{Int.reverse(sample)}.

<table>
<thead>
<tr>
<th>j</th>
<th>\texttt{s.charAt(j)}</th>
<th>original t</th>
<th>final t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>d</td>
<td>empty</td>
<td>d</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>d</td>
<td>dc</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>dc</td>
<td>dcb</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>dc</td>
<td>dcb</td>
</tr>
</tbody>
</table>

Table for Figure 2.20a Shows how the program builds the string \texttt{t} using \texttt{t = t + s.charAt(j)}.

d
dc
dcb
dcba

Figure 2.20b Shows how the final string evolves.
The second method, shown in Figure 2.21, also reverses the string but does so by running the loop forwards. The difference is that in \( t = s.\text{charAt}(j) + t \) the character precedes the string in the concatenation. How this evolves is shown in the Table for Figure 2.21.

<table>
<thead>
<tr>
<th>( j )</th>
<th>( s.\text{charAt}(j) )</th>
<th>original ( t )</th>
<th>final ( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>empty</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>a</td>
<td>ba</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>ba</td>
<td>cba</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>cba</td>
<td>dcba</td>
</tr>
</tbody>
</table>

The Table for Figure 2.21. Shows how the final string evolves from \( t = s.\text{charAt}(j) + t \)

The method will be invoked by `Int.reverse1(sample)` where `sample` is assigned "abcd".

The next method (Figure 2.22) sums the digits in a string assuming all the characters are digits. The first task is to convert the digit in character form to its int numerical value. We’ve seen that if the operands in an addition or subtraction are characters, the calculation is done using the ASCII code values of the characters. So ‘3’-’0’ is 51 - 48 or 3. We find the length of the string, then analyse each character. If we just converted the digit in character form to an int, we’d get a number 48 too large, so we subtract the character ’0’—its value is 48 as shown in the Table for Figure 2.22.

<table>
<thead>
<tr>
<th>( s.\text{charAt}(j) )</th>
<th>code for ( s.\text{charAt}(j) )</th>
<th>code for ’0’</th>
<th>digit</th>
<th>original sum</th>
<th>final sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>49</td>
<td>48</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>48</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>48</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>48</td>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>48</td>
<td>5</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

Table for Figure 2.22. Shows how the addition of the digits work using \( \text{digit} = s.\text{charAt}(j) - \text{’0’}; \text{sum} = \text{sum} + \text{digit} \).

After each character is converted to a digit, it’s added to `sum` which is initialized to zero before the loop. The fact that the return type is no longer `void` but is now `int` means that we must end the method with `return` followed by an `int` value, here `sum`. When it’s called in the `main` method in `int sum = Int.sumDigits(st)` the result is assigned it an `int`. Since the method returns a value, it may be placed in an output statement, e.g., `System.out.println(Int.sumDigits)`, whereas a `void` method cannot. Although there may be more than one `return` in a non-`void` method, the last statement must be a `return`.

Note that a `static` method can only call other `static` methods or use `static` fields. There is no restriction, however, on instance methods. They can call both other instance methods (see Figure 2.17) and `static` methods too. The latter happens when you need a method that applies to all the objects of a class. An example of static method used with objects would be in a program simulating a coin throwing game played by two players represented by two objects (see Chapter 5). The total number of throws could not be an instance variable. It would be recorded by a static variable, `total` which would be used by both objects. This value would be retrieved by a `static` method `getTotal()`. Since this method is used for the entire class, `static` methods are also called `class` methods.
REVERSING A STRING II

```java
public static void reverse1(String s) {
    // 2nd version for printing a string in reverse
    int len = s.length();
    String t = ""; // initialize to empty string
    for(int j = 0; j < len; j++)
    {
        t = s.charAt(j) + t;
        System.out.println(t);
    }
}
```

Figure 2.21. Reverse a string by having the character precede the intermediate string.

SUMMING THE DIGITS IN A STRING

```java
public static int sumDigits(String s) {
    // sums the digits in the string
    int len = s.length();
    int sum = 0;
    int digit;
    for(int j = 0; j < len; j++)
    {
        digit = s.charAt(j) - '0';
        sum = sum + digit;
    }
    return sum;
}
```

Figure 2.22. Converts the digit in character form to an int. Since the return type is an int, we end the method with return followed by an int value. Although there may be more than one return in a non-void method, the last statement must be a return.
The final method (Figure 2.23) converts an entire string consisting of digits to its int equivalent. It differs from the previous method in that the sum is calculated in \( \text{sum} = 10 \times \text{sum} + \text{digit} \). How this works is shown in the Table for Figure 2.23. If all the methods including main are in the same class and are all static, as they are in Figure 2.23, all that is required to call them is to write their name with the proper parameter.

<table>
<thead>
<tr>
<th>digit</th>
<th>original sum</th>
<th>10*sum</th>
<th>final sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>120</td>
<td>123</td>
</tr>
<tr>
<td>4</td>
<td>123</td>
<td>1230</td>
<td>1234</td>
</tr>
<tr>
<td>5</td>
<td>1234</td>
<td>12340</td>
<td>12345</td>
</tr>
</tbody>
</table>

Table for Figure 2.23. Shows how \( \text{digit} = \text{s.charAt(j)} - '0'; \text{sum} = 10 \times \text{sum} + \text{digit} \) works.

10 THE NUMERIC PRIMITIVE TYPES

THE INTEGRAL PRIMITIVE TYPES

The int type, is one member of a family of integer primitive types consisting of byte, short, int and long, enumerated here in ascending order according to range (maximum minus minimum value). The char type is also a member of this family. All of the ranges and the number of bits allocated to numbers of each type are listed in Table 2.7. We see that the maximum value of an int is a ten-digit number, 2147483647. Assigning a larger number to a long variable, as in \( \text{long lo} = 123456789012 \), causes a compilation error unless the number is immediately followed by an uppercase or lowercase "L". So \( \text{long lo} = 123456789012L \) will compile.

THE FLOATING POINT PRIMITIVE TYPES

Numbers that contain a decimal point are called floating point numbers. There are two types, float and double. Their ranges are also shown in the Table 2.7. Since float’s have a narrower range than double’s, if \( d \) is a double then \( \text{float fl} = d \) causes the "possible loss of precision" error. To correct it, cast \( d \) as a float by writing \( \text{float fl} = (\text{float})d \). This doesn’t increase the precision, it just warns you of the loss of it. All floating pointing values unless indicated otherwise are considered double precision. Thus \( \text{float fl} = 12.3 \) causes an error because a double is assigned to a float. To correct this, indicate that 12.3 is a float, by placing a lowercase or uppercase "F" after it, thus typing this statement as \( \text{float fl} = 12.3f \). To assign the floating points types to any of the integer types you must use casting, as in \( \text{long lo} = (\text{long})\text{fl} \). When a floating point number is cast as an integer, it is truncated, that is, all digits to the right of the decimal point are lopped off. So 12.8 becomes the int 12. Whenever an integer type, even a byte (which we know has a narrower range then a char) is assigned to a char variable, it must be cast, e.g., if byte \( b = 10 \); you must write char c = (char)b.

When a float has more than seven digits, it’s printed in exponential form. So 123456789.0 is printed as 1.23456789E8; however, only the first seven digits of a float are significant. You can see this by observing the loss of precision when an int constant having more than seven digits is assigned to a float as is the case when \( \text{float fl} = 1234567890 \) is executed. When the value of fl is printed, it is seen to be 1.23456794E9. The last two digits, 94, are not significant. A double constant can have at most seventeen significant digits. So when a long constant having more than
public static int parseInt(String s) {
    //converts the digits in the string to an integer
    int len = s.length();
    int sum = 0;
    int digit;
    for (int j = 0; j < len; j++) {
        digit = s.charAt(j) - '0'; //ascii codes are subtracted
        sum = 10 * sum + digit;
    }
    return sum;
}

public static void main(String[] arg) {
    String sample = "abcd";
    String st = "12345";
    reverse(sample);
    reverse1(sample);
    int sum = sumDigits(st);
    System.out.println("sum of digits in " + st + " is "+sum);
    int num = parseInt(st) + 1;
    System.out.println(st + " + 1 is " + num);
}

Figure 2.23. Shows how a string of digits is converted to an int. Since all the methods are in the same class and are static, they are invoked simply by writing the method name and the appropriate parameter. Our method parseInt produces the same results as the parseInt that is part of the API (section 2.20)

THE RANGE OF THE NUMERICAL PRIMITIVE TYPES

<table>
<thead>
<tr>
<th>Type</th>
<th>range</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>-128 to 127</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>-32768 to 32767</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>-2147483648 to 2147483647</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>-9223372036854775808 to 9223372036854775807</td>
<td>64</td>
</tr>
<tr>
<td>char</td>
<td>0 to 65535</td>
<td>16</td>
</tr>
<tr>
<td>float</td>
<td>1.4E-45 to 2.4028235E38</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>4.9E-324 to 1.7976931348623157E308</td>
<td>64</td>
</tr>
</tbody>
</table>
seventeen digits is assigned to a `double` variable, precision is again lost. When a `double` or `float` is divided by 0, the result is printed or stored as plus or minus "infinity" depending upon the sign of the numerator. Execution continues after that. When an integer type is divided by zero, however, the program terminates and issues what is called an exception. When a program prints a `double` or `float` that is assigned the quotient of zero divided by zero, the letters `NaN` are printed where `NaN` means "not a number". Floating point numbers are just approximations to the actual value and therefore should never be be used as a for loop index nor tested for equality. For instance `x*(1.0/x) == 1.0` may not be true for some values of `x`. The casting rules that apply to assignment statements apply also to passing parameters. In Figure 2.24a three actual parameters are needed to correspond to the three formal parameters in the method heading `testParam(int a, float b, char c)`.

### 11 THE ORDER OF EVALUATION

The operators for floating point values are "*" (multiplication), "/" (division), "+" (addition), and "-" (subtraction). The integer operators for addition, subtraction and multiplication are the same as for floating point values. Remember that in a division, the "dividend" is divided by the "divisor". The number of times the divisor goes into the dividend is called the "quotient". What is left over after the division is called the "remainder". The "/" operator produces the quotient and the "%" produces the remainder. The quotient operator used in integer division has a different function than when used with floating point values. For instance, the value of 7/3 is 2 not 2.33. The remainder, 7%3 is 1. This last operation, 7%3, is also read "seven mod three" where "mod" stands for modulo. As another example, 3/4 is 0 because 4 goes into 3 zero times. The remainder 3%4 is three. The "%` can also be used with floating point numbers, e.g., 12.5%4.0 is 0.5.

If two non-assignment operators have the same precedence, (see Table 2.8) they are evaluated from left to right. Thus `y = 3+4+5;` is evaluated as `y = 7+5;` or `y = 12`. Similarly, `y = w*x/z` is evaluated as `y = (w*x)/z`. If operators of a mixed precedence appear in an expression, the ones with the highest precedence are performed from left to right, then the one with the second highest precedence is performed from left to right, and so forth. Thus in `z = 4/3 + 8%3 - 3` since the "/" and "%" have higher precedence than addition and subtraction, the result is evaluated as `z = 1 + 2 -3`; which is evaluated as `z = 3-3`. If an expression is enclosed in parentheses, it is done first. So in `(x+y)*z`, `x+y` is done first, and is then multiplied by `z`. If more than one pair of parentheses appears, the pairs are evaluated from left to right. Inner parentheses are evaluated first. So the order of evaluation in `(a+b)*(c-d)/(e+f)*g`, is `a+b, c-d, e+f, (e+f)*g, (a+b)*c-d`, and that divided by `(e+f)*g`. If an integer and a floating point value are used with a binary operator (one that has two operands), the computer treats it as if both operands are floating point, so 3/4.0 is 0.75. Table 2.8 indicates the order of evaluation of various operators we have encountered so far. This order is also called the order of precedence. The operators at the top are said to "bind more tightly" than the ones lower than them in the table. `x` represents an expression. The operator with the highest precedence (the parentheses) is listed first; the one with the lowest (=), is listed last. The `+x` and `-x` are unary operations – only one operand is involved.
PASSING PARAMETERS

```java
public class Passing {
    public static void testParam(int a, float b, char c) {
        System.out.println("int: \"+ a + \" float: \"+ b +\" char: \"+c);}
    
    public static void main(String[] asd) {
        testParam('a', 123, (char)97);
    }
}
```

Figure 2.24a. The type of the actual parameters must be compatible with that of the formal parameters. So 97 must be cast as a `char`; 'a' is not cast because a `char` value is narrower than an `int`. Each formal parameter is paired with a type specifier that precedes it; these pairs are separated by commas. Another method with the heading `testParam(char c, float b, int a)` would have a different signature than the method shown here because of the order of the type specifiers.

int: 97
float: 122.0
char: a

Figure 2.24b. Running the program of Figure 2.24a.

THE ORDER OF EVALUATION

<table>
<thead>
<tr>
<th>Definition</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>parentheses</td>
<td>( )</td>
</tr>
<tr>
<td>postfix operators</td>
<td>[] ++ -x -</td>
</tr>
<tr>
<td>unary operators</td>
<td>+x -x</td>
</tr>
<tr>
<td>creation or cast</td>
<td>new (type)x</td>
</tr>
<tr>
<td>multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>additive</td>
<td>+ -</td>
</tr>
<tr>
<td>relational</td>
<td>&lt; &gt; &gt;= &lt;=</td>
</tr>
<tr>
<td>equality (boolean)</td>
<td>= !=</td>
</tr>
<tr>
<td>assignment</td>
<td>=</td>
</tr>
</tbody>
</table>

Table 2.8 The operators we have discussed so far. The operator with the highest precedence, (), is listed first; the one with the lowest ( = ), is listed last.
In Figure 2.25a we use the "/" and "%" in integer arithmetic to reverse the order of digits in an integer. Note that \(341 \div 10\) is 1, so we have obtained the right-most digit. Also \(341/10\) is 34, so the original number is modified by the removal of the right-most digit. The Table for Figure 2.25a shows how this is done.

<table>
<thead>
<tr>
<th>number</th>
<th>digit=number%10</th>
<th>reverse</th>
<th>10×reverse + digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>341</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>4</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>14</td>
<td>143</td>
</tr>
</tbody>
</table>

The Table for Figure 2.25a.

In \(len = ("" + number).length()\), the int parameter is converted to a string and then its length is determined. This indicates the number of digits in the integer and thus the number of iterations in the loop. If there are too many iterations, the value of reverse will have trailing zeros, e.g., 14300

12 THE final STATEMENT

It is sometimes advantageous to use identifiers whose values should not change during the program’s execution. These identifiers are called "final constants". If you try to change the value of a final constant later in the program, an error occurs. The convention is that final constants are written in uppercase. We will be using final static int HEIGHT = 10, WIDTH = 50. If there are two parts or words in a constant identifier, as in "max value", use an underscore to separate them, i.e., MAX_VALUE. A static method cannot call instance variables or instance (non-static) methods. A static method can only access static declarations (and other static methods). That’s why we labeled the final declaration here as static. It is accessed by the static methods printValue and drawLines. A final declaration, however, need not be static.

The program in Figure 2.26a produces a pattern in a region indicated by the HEIGHT and WIDTH final constants. HEIGHT indicates the number of lines in the region and WIDTH, the number of columns in each line. The two innermost loops print a line of characters indicated by the parameters c1 and c2. The first of these loops prints \(m\) characters and the second prints \(n\) characters. So there are \(m + n\) characters printed when those two loops are finished. This process should be repeated \(\text{WIDTH}/(\text{m+n})\) times, an integer, so we write int width = \(\text{WIDTH}/(\text{m+n})\) and width is used to repeat the two innermost loops in for(int in1 = 0; in1 < width; in1++). The driver for the program is shown in Figure 2.26b. We want the pattern in successive lines to alternate the order of c1 and c2, so we swap them at the end of each line.

The process of swapping is described in the Table for Figure 2.26a

<table>
<thead>
<tr>
<th>temp</th>
<th>1 ←</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 _</td>
<td>2 ↑</td>
</tr>
</tbody>
</table>

Table for Figure 2.26a.

If we just write \(c1 = c2;\ c2 = c1\), both values would be the original value of c2.
REVERSING THE ORDER OF THE DIGITS IN AN INTEGER

```java
public class ReverseInt //reverses the digits in an integer
{
    public static int intReverse(int number)
    {
        int digit, reverse = 0;
        int len = ("" + number).length();//# of digits in number
        for(int j = 0; j < len; j++)//
        {
            digit = number % 10;//get right-most digit
            number = number / 10;//remove right-most digit
            reverse = 10*reverse + digit;//form new integer
        }
        return reverse;
    }
    public static void main(String [] asd)
    {
        System.out.println( intReverse(341) );
    }
}
```

Figure 2.25a. Dividing by ten removes the right-most digit. The result is then reassigned to number. The reversed number is stored in reverse.

143

Figure 2.25b. Running the program of Figure 2.25a.

```java
public class Pattern //makes a pattern using several parameters
{
    final static int HEIGHT = 4, WIDTH = 50;
    public static void drawLines(int m, int n, char c1, char c2)
    {
        int width = WIDTH/(m+n);
        for(int out = 0; out < HEIGHT; out++)
        {
            for(int in1 = 0; in1 < width; in1++)
            {
                for(int in2 = 0; in2 < m; in2++)
                    System.out.print(c1);
                for(int in3 = 0; in3 < n; in3++)
                    System.out.print(c2);
            }
        }
    System.out.println();
    char temp = c1;//swap characters
    c1 = c2;
    c2 = temp;
    }
    public static void printSize()
    {
        System.out.println("The height is "+ HEIGHT + " The width is "+ WIDTH);
    }
}
```

Figure 2.26a
13 THE WRAPPER CLASSES

Each of the primitive types have associated with them a class called the *wrapper* class containing among others, many static utility methods and a method that converts an object of the class to its primitive type. The names of classes corresponding to the non-character numeric primitive types are *Byte*, *Short*, *Integer*, *Long*, *Float*, and *Double*. These all have a static method that converts a string into the appropriate numerical primitive type. The respective names of these methods are *parseByte*, *parseShort*, *parseInt*, *parseLong*, *parseFloat* and *parseDouble*. For instance, *Integer.parseInt("1234")* generates the int 1234. Similarly *Double.parseDouble("1234.5")* produces the double 1234.5. The numeric wrapper class corresponding to the *char* primitive type is the *Character* class (it doesn’t have a parse method); and the non-numeric wrapper class, the one corresponding to the *boolean* type, is the *Boolean* class.

The numeric wrapper classes have final static fields *MAX_VALUE* and *MIN_VALUE* that contain the maximum and minimum values of these primitive types. For instance, *Integer.MAX_VALUE* is 2147483647. As we will now see, one can create wrapper class objects. The static fields apply to all objects of the class; hence static fields are also called *class* fields.
public class Driver4
{
    public static void main(String[] asd)
    {
        printSize();
        drawLines(3, 2, 'X', '*');
    }
}

Figure 2.26b. The driver for the pattern program.

The height is 4 The width is 50
XXX***XXX**XXX***XXX**XXX***XXX**XXX***XXX**XXX***XXX**
***XXX***XXX**XXX***XXX**XXX***XXX**XXX***XXX**XXX***XXX**
XXX***XXX**XXX***XXX**XXX***XXX**XXX***XXX**XXX***XXX**
***XXX***XXX**XXX***XXX**XXX***XXX**XXX***XXX**XXX***XXX**

Figure 2.26c. Running the program of Figure 2.26a.

14 THE toString METHOD

The Object class contains a method, toString, that converts objects to strings. Each of the wrapper classes has a toString method that supercedes the Object’s one. These methods convert a primitive type to a string. A version of the method can be written for any class. It can have no parameters. One way of implementing toString is shown in Figure 2.28a for class StringTest. Whenever an object of the class is used in a println, as in System.out.println(st), or concatenated with a string, as in s = "w/concatenation: " + st, the toString method is automatically invoked. Here it is used to print the instance variable ch (see Figure 2.28b) and st, the result of a concatenation. If a class does not have a toString method, the println would use the Object’s class version of the method. This prints the class’s name, an @, and then the hexadecimal version of object’s hash code. So if the class’s name is AnyClass, and anyObj is an instance of the class, then println(anyObj), would print for instance, AnyClass@256a7c.

To understand how the toString method works for the wrapper classes, let’s write a simplified version of the Character class (see Figure 2.28c.) In the constructor, we set the instance variable equal to the parameter used in the instantiation, value = ch. The toString method simply returns ""+value, the value of the instance variable value concatenated with the empty string. The method charValue, as you expect, only returns value.

THE toString METHOD

public class StringTest //shows how toString is implemented
{
    private char ch;
    public StringTest(char c)
    {
        ch = c;
    }
    public String toString()//allows using object in print
    {
        return "character used is " + ch;
    }
}
Figure 2.28a. If a class has a `toString()` method, whenever an object of that class appears in a `println` or in a concatenation, the string returned by `toString()` is used.

Figure 2.28b. Running Figure 2.28a.
THE toString METHOD IN THE Character CLASS

```java
public class Character
//simulates the Character wrapper class
{
    char value;

    public Character(char ch)
    {
        value = ch;
    }

    public char charValue()
    {
        return value;
    }

    public String toString()
    {
        return ""+ value;
    }
}

class Tester
{
    public static void main(String[] asd)
    {
        char ch = 'a';
        Character c = new Character(ch);
        System.out.println("value is " + c.charValue());
        System.out.println( c);
    }
}
```

Figure 2.28c. A simplified version of the `Character` class. The `toString` method returns the value of the instance variable `value` concatenated with the empty string.

value is a
a

Figure 2.28d. Running the program of Figure 2.28c.
15 THE Math CLASS

The `Math` class has static methods that perform mathematical functions. Some of them are listed in the Table. For instance, to get a random number between zero and one, use `double num = Math.random();`. To generate random 0’s and 1’s to simulate a coin toss, use `int coin = (int)(Math.random() + 0.5)`. Here a random number that is greater than 0.5 after the addition becomes a number greater than 1.0 and is truncated to 1. One that is less than 0.5 is truncated to 0.

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>determines the absolute value</td>
<td>same as argument</td>
</tr>
<tr>
<td>cos(x)</td>
<td>x is a double &amp; in radians</td>
<td>double</td>
</tr>
<tr>
<td>exp(x)</td>
<td>raises x to the base e</td>
<td>double</td>
</tr>
<tr>
<td>log(x)</td>
<td>x is a double</td>
<td>double</td>
</tr>
<tr>
<td>max(x, y)</td>
<td>maximum of two values</td>
<td>same as argument</td>
</tr>
<tr>
<td>min(x, y)</td>
<td>minimum of two values</td>
<td>same as argument</td>
</tr>
<tr>
<td>random()</td>
<td>0 ≤ result &lt; 1</td>
<td>double</td>
</tr>
<tr>
<td>sin(x)</td>
<td>x is a double &amp; in radians</td>
<td>double</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>x is a double</td>
<td>double</td>
</tr>
<tr>
<td>tan(x)</td>
<td>x is a double &amp; in radians</td>
<td>double</td>
</tr>
<tr>
<td>toRadians(x)</td>
<td>x in degrees</td>
<td>double</td>
</tr>
</tbody>
</table>

16 JOptionPane CLASS

A package is a group of related classes. In order to make one available for easy use in a program, the package must be imported from the API. The program in Figure 2.29 imports the package containing the `JOptionPane` class which has a static method that displays a window on the screen. In it we can type data to be entered into the program as a string. In `Reverse.intReverse(number)` we invoke the static method `intReverse` of the class `ReverseInt` (Figure 2.25a) that reverses the order of digits in an integer. The `String`, `Math`, `System` and wrapper classes that we’ve used so far are part of a package, `java.lang`, that’s automatically imported into programs. The term `import` is somewhat of a misnomer. If you omit `import javax.swing.JOptionPane`, you can still use `showInputDialog` by writing its fully-qualified name in the program, i.e., `String input = javax.swing.JOptionPane.showInputDialog("Type string")`. It is, however, inconvenient. The statement `System.exit(0)` is required to exit the program. A non-zero argument would indicate an abnormal exit.
USING JOptionPane

```java
import javax.swing.JOptionPane;
public class Test
{
    public static void main(String[] args)//reads input from terminal using GUI
    {
        String input = JOptionPane.showInputDialog("Type string");
        int number = Integer.parseInt(input);
        System.out.println(ReverseInt.intReverse(number)); // see Figure 2.25a
        System.exit(0); // needed to gracefully exit program
    }
}
```

Figure 2.29. We import a package from the API into our program. The import statement must precede the program. The method showInputDialog produces a window in which you can type input. Since int data is required from the input, Integer.parseInt is used.
17 THE boolean PRIMITIVE TYPE

One can either assign a boolean value (true or false) or a boolean expression to a boolean variable. An example of the latter is shown in method isMatch in Figure 2.30a. Here we first generate the integers 0 and 1 randomly using \texttt{int first = (int)(2*\texttt{Math.random()})}. We then send first to method \texttt{isMatch} where a second integer, either 0 or 1, is generated. In \texttt{boolean match = second \texttt{== first}} since the test for equality (\texttt{==}) has a higher precedence than the assignment (=), the boolean expression \texttt{second == first} is evaluated as true or false, depending on whether the two generated digits are equal or not. The result is assigned to \texttt{match} and then returned by the method. Since a primitive type is returned, we can place \texttt{isMatch} in a print statement in the \texttt{main} method and either true or false is printed. For every four pairs of digits generated, we would expect two pairs to be comprised of the same digits. Figure 2.30b shows the results. In future programs, when the value of a boolean expression is returned via a boolean variable, we will skip the assignment to the boolean variable and just return the expression. If we did this in this program we would write \texttt{return second == first} and eliminate the \texttt{match} variable.

Why does the algorithm we use to generate a random number produce a zero or one? Since \texttt{random()} produces a random positive \texttt{double} less than one, \texttt{2*\texttt{Math.random()}} yields a positive \texttt{double} less than two. Casting truncates the result, producing 0 or 1.

18 SOLVING A PROBLEM

THE PROBLEM STATED

Let’s investigate a problem from the branch of mathematics called number theory that envolves using some of the algorithms we learned so far. Take a three-digit integer, let’s say 132, and form another integer by placing the digits in descending order, in our case 321. The digits are now said to be sorted in descending order. Next, form the number whose digits are in the reverse order, here 123. Subtract it from the original sorted-digit-number: 321 minus 123 equals 198. Repeat this process using 198 as the three-digit integer and so on, for five times, and observe the results. It is known that unless all three digits are the same, the results should converge to 495 after a few iterations. Let’s write a program to verify this.
THE boolean PRIMITIVE TYPE

public class RandomBoolean

//Assigning boolean expressions
{ public static boolean isMatch(int first)
    //return true if the two digits generated are equal
    {
        int second = (int)(2*Math.random());
        System.out.print(second);
        boolean match = second == first;
        return match;
    }

    public static void main(String[] asd)
    {
        for(int j = 0; j < 8; j++)
        {
            int first = (int)(2*Math.random());
            System.out.print("" + first + isMatch(first) + "{" );
        }
        System.out.println();
    }
}

Figure 2.30a. A boolean variable is assigned the value of a boolean expression. The variable match can be eliminated and the return statement written as return second == first.

00true|11true|10false|01false|10false|00true|10false|00true|

Figure 2.30b. Running the program of Figure 2.30a
Common Errors

- Placing a semicolon after a method heading, e.g., `void one();` causes the message "missing method body, or declare abstract"

- Placing a semicolon directly after a `for` statement causes the `for` to operate on an empty statement.

- Using a variable as the increment in a `for` loop that is independent of the initial and continuance condition may cause an infinite loop, e.g., in `for(int j = 0; j < MAX; size++)` will cause an infinite loop if `size` is not altered in the loop.

- Forgetting to place the `main` header in a class that requires one will cause a compilation error.