CHAPTER 2
AN INTRODUCTION TO JAVA

©2007 by Samuel L. Marateck

Available on line at: http://cs.nyu.edu/courses/fall07/V22.0101-002/try3.pdf

Chapter Overview.

In this one we give an introduction to Java applications, starting with writing results to the monitor screen, going on, among other things, to using `for` loops, instantiating objects, using parameters, using the constructor and then overloading it, performing calculations with primitive types, using wrapper classes, polymorphism, writing static methods and using prewritten ones from the Java library, implementing an algorithm and testing the program for accuracy, and finally explaining the use of the Comparable interface.

Contents

1 THE SHORTEST MEANINGFUL JAVA PROGRAM 4

2 RUNNING THE PROGRAM IN JAVA 6

3 THE int PRIMITIVE TYPE 8

4 VARIABLE NAMES 10

5 AN INTRODUCTION TO THE for LOOP 12

6 USING OBJECTS 14

7 USING PARAMETERS 16

8 THE String CLASS 18

9 THE CONSTRUCTOR 18

10 OBJECT ALIAS 22

11 PITFALLS 24

12 USING this TO REFER TO THE IMPLICIT PARAMETER 30

13 OVERLOADING THE CONSTRUCTOR 30

14 THE char PRIMITIVE TYPE 32

15 USING char VARIABLES AS LOOP VARIABLES 34

16 STATIC METHODS **34**

17 THE NUMERIC PRIMITIVE TYPES **38**

18 THE ORDER OF EVALUATION **40**

19 THE final STATEMENT **42**

20 THE WRAPPER CLASSES **44**

21 CREATING WRAPPER OBJECTS, POLYMORPHISM **44**

22 THE toString METHOD **46**

23 THE Math CLASS **48**

24 JOptionPane CLASS **48**

25 THE boolean PRIMITIVE TYPE **50**

26 SOLVING A PROBLEM **50**

27 POLYMORPHISM REDUX **55**

28 THE COMPARABLE INTERFACE **55**

29 CHAPTER REVIEW **56**

blank page.

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 `public static void main( String[] arg)` the name of the method is `main`. The term `public` is the *access specifier*. In general, it indicates the accessability of a method to other classes. The fact that the `main` method is `static` prohibits it from calling a non-static method in the same class by simply using the method's name. The next term `void` is the *return type* and it indicates that the method returns no information. The terms `String[] arg` will remain a mystery for the first few chapters except that we'll say here that `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.

MOTIVATION

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

`System.out.println(" Hello John and Mary!");` enables the program to communicate with the outside world, by printing `Hello John and Mary!` on the monitor screen. What appears between the double quotes, here, `Hello John and Mary!` is called a "string" and with just the exception of when a \ 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 `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 `system.out.println(" Hello John and Mary!")`, or for that matter, `Public static void Main( String[] asd)` produce errors since `system` should be capitalized and `Main` shouldn't be. Using `out` directs the output to the monitor screen. Ordinarilly you'd expect `out` to be a method of the `System` class. Actually, `System.out` is an instance[1] of another class called `PrintStream` and `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 braces "{" and "}". Similarly, the information in a method (for now, just one statement) is sandwiched between another pair of these braces.

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.

[1]technically an object

A SIMPLE JAVA PROGRAM

```
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

```
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 succesive 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("Myname")`, 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 nameis 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.

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 `print`, 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 shows a program that calculates the area of a rug. Before you storing 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, – called `int` literals). It belongs to the *primitive* type family consisting of the numeric types (numbers and characters); and the `boolean` type (true and false values).

To store `int` literals 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

Statement	`length`	`width`	`area`
`length = 20`	20	Uninit	Uninit

shows that although `length` has been assigned a value, `width` and `area` have not been assigned values – they are thus unitialized (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`

Statement	`length`	`width`	`area`
`width = 50`	20	15	Uninit

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.

Statement	`length`	`width`	`area`
`area = length*width`	20	15	300

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*.

Statement	`length`	`width`	`area`
`length = 20`	20	Uninit	Uninit
`width = 15`	20	15	Uninit
`area = length*width`	20	15	300
`width = 10`	20	10	300
`area = length*width`	20	10	200

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 [2]. 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 8 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*. A negative value begins with a minus sign, e.g., -3.

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.

[2] If there is a string value in the expression, the integer values may be concatenated, as we will see later in the paragraph.

IDENTIFIER NAMES

Valid identifier names
`newLength`
`oldLength`
`obj`
`obj1`

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

bad variable	reason
`first obj`	embedded blank
`1obj`	starts with digit
`one$`	illegal character, `$`

Table 2.2 Illegal identifier names and why they are illegal.

RESERVED WORDS IN JAVA

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

Table 2.3 These are called *reserved words.* They cannot be used as identifier names.

AN INTRODUCTION TO boolean VALUES

A non-numeric value[3] 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 indentifier 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.

Operator	Explanation
$==$	equals
$>=$	greater than or equal
$<=$	less than or equal
$>$	greater than
$<$	less than
$!=$	not equal

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

```
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 underfined in the program after the `println` statement is executed. Thus

```
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 "}".

[3] Remember that we consider the **char** type as part of the integer family.

THE for LOOP

```
public class Looping
{// Introduction to the for loop
    public 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
{// the for operates on a statement block.
    public 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 it's 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 it's upper limit is 3, so `j < 4` is false and the loop is not executed.

The form of the `for` 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.

6 USING OBJECTS

Let's see how to use an object in Java. In Figure 2.7, we instantiate an object in `ObjectIntro obj = new ObjectIntro();` The declaration part, `ObjectIntro obj`, indicates that `obj` is of type `ObjectIntro`, the class name. The `new ObjectIntro()` part tells the computer to construct an instance of `ObjectIntro`. In doing this it reserves space for the object in the type of memory called the *heap.* This process is called *instantiation* and `obj` contains the address on the heap where this object is stored; `obj` is called a *reference variable*[4] or a *pointer.* Since this is done during execution, the heap is called *dynamic memory.* Once you instantiate the object, you can invoke its methods, here `writeName`, by typing `obj.writeName()`, that is, by placing a dot after the reference and following it by the method name, and in this case, simply following that by "()". The Table for Figure 2.7a shows how the heap is configured,

	information pertaining to the object
`obj`	

Table for Figure 2.7 The part of the heap that stores the object referenced by `obj`

where `obj` is the address on the heap that the memory allocated for the object begins. The object is the data stored on the heap. The methods that operate on the object are stored in the part of the memory called the *method area* which is logically part of the heap. The method begins with `public void writeName()`. The `void` part indicates that this method does not return anything. Since we are not passing any information to the method, nothing appears in the parentheses after `writeName`. Any variable that is declared in a method is called a *local* variable. It cannot be accessed ouside the method and it retains its value only during the execution of the method. In `writeName`, `j` is a local variable; but as we know, it's undefined after the loop is executed.

[4] The word *object* is used even for the reference variable.

NESTED LOOPS

```
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

OBJECTS

```
public class ObjectIntro
//prints a string many times
{   public void writeName()
    {   for(int j = 0; j < 5; j++)
          System.out.println("Jane Doe");
    }
    public static void main(String[] arg)
    {   ObjectIntro obj = new ObjectIntro();
        obj.writeName();
    }
}
```

Figure 2.7. `obj` is a reference to an object of `ObjectIntro`. To invoke method `writeName`, write `obj.writeName()`. Memory for objects is allocated during execution-time in an area called the *heap*, hence it's called *dynamic memory*. The object is what is stored on the heap. The methods are stored in the *method area* which is logically part of the heap.[5]

[5] See "The Java Virtual Machine Specification", by Tim Lindholm and Frank Yellin, Addison-Wesley, 2nd ed., 1999. The method area is also called the *permanent generation* space.

Another way of writing the program is to have the original class contain only `writeName`. Then, write a separate class in the same file that only contains the main method as is shown in Figure 2.8. This separate class is called a *driver* since it executes or drives the class containing the method. In order to make the program easier to read, it's useful to separate classes from other classes and methods from other methods with a comment consisting of a dashed line as shown in Figure 2.8 or just a blank line. In order to maintain the book's dual-page format, we have done this only when there is room on a given page. Blank lines and blank spaces are called *white spaces* and should be used to make your program easy to read.

A class is written to be used as a black box, that is, the user of the class should know how to call the class methods but should not have access to the inner workings of the class. Access to the class should be restricted to calling the public methods of the class. If the `main` method driving the class is in the class, everything in the class is accessible to it. If, however, the `main` method is in a separate class, anything that is `private` in the original class is inaccessible. For instance, if we were to mistakingly make `writeName` private by writing `private void writeName()`, the program would not compile because the instruction `obj.writeName()` is not allowed. Please note that if there is more than one class in the file only the class name used to save a file, here `Source`, can be public. Thus we write `class Driver` instead of `public class Driver`. It is common to place the driver in its own file but in the same directory as the other file, then of course it is written as a public class.

7 USING PARAMETERS

One method can pass information to another via the method's heading by using what is called a parameter. In Figure 2.9 we pass information from the main method to `writeName` by placing a variable, `number`, between the parentheses in `obj.writeName(number)`. The variable `number` is called the *actual parameter* and corresponds to the *formal* or *dummy parameter* `num` in the method's heading. The formal parameter is preceded by its type, here, `int`. The computer establishes one memory location for each actual parameter and one for each formal parameter, as shown in the Table for Figure 2.9, and copies the information from the actual to the formal one. This is process is called *passing information by value*, and any change in the value of the formal parameter in the method where it's defined does not effect the value of the actual parameter that was passed to it. We could have used the value 8 as the actual parameter, thus writing `obj.writeName(8)`. The 8 would have been passed to `num`. Now you can understand that in `public static void main(String[] arg)`, `arg` is a parameter.

`main`		
	`number` (actual parameter) ↓	
`writeName`		
	`num` (formal parameter)	

The table for Figure 2.9. Shows that there is one location for the actual parameter, `number` and one for the formal parameter `num`. The value in the first is copied to the second. Any change of the formal parameter does not effect the value of the actual one, i.e., the copying goes one way.

USING A DRIVER

```
public class Source
//Uses a driver to print a string many times
{   public void writeName()
    {   for(int j = 0; j < 5; j++)
          System.out.println("Jane Doe");
    }
}
//---------------------------------------------
class Driver
{   public static void main(String[] arg)
    {   Source obj = new Source();
        obj.writeName();
    }
}
```

Figure 2.8. Another way of writing the program is to have the main method in a different class. If anything in the instantiated class is `private`, it cannot be accessed by the driver. This insures the integrity of the instantiated class – the user cannot accidentally corrupt this class from the driver. Only the class name used to save the program, here `Source`, can be `public`. Hence class `Driver` is not `public`.

USING PARAMETERS

```
public class Parameter
//prints a string many times using a parameter

{   public void writeName(int num)
    {   for(int j = 0; j < num; j++)
            System.out.println("Jane Doe");
    }

    public static void main(String[] arg)
    {   Parameter obj = new Parameter();
        int number = 8;
        obj.writeName(number);
    }
}
```

Figure 2.9. The value of the actual parameter `number` is transferred to the dumy parameter `num`. They both are placed between the parentheses after the method name. The dummy parameter, however, must be preceded by its type, here `int`. If the value of the dummy parameter, here `num`, is changed in the method, the actual parameter, here `number`, remains unchanged.

8 THE String CLASS

Another class found in the Java API is the `String` class. To place the string literal, "thinking" on the heap and have the `String` reference, `one` point to it, write `String one = new ("thinking").` 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.

t	h	i	n	k	i	n	g
0	1	2	3	4	5	6	7

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 method is the `substring` method. In `one.substring(2, 5)` which yields "ink", the first parameter is the index of the first element in the substring, the second one indicates that the index of the last element (here 4) has 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. We see that `one.substring(0, one.length() )`, (here `one.substring(0, 8)`) gives all the 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. When `String` literals are assigned to `String` variable as in `String one =''dog''; String two = ''dog''`, they are placed in the *String literal pool*. Now the "==" can be used to test for equality. So `one == two` is true. 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.

9 THE CONSTRUCTOR

In Figure 2.10a we pass the string `"Jane"` to class `Names` by writing `Names obj = new Names("Jane")`, i.e. use "Jane" as an actual parameter. Next we must write what is called a constructor with a formal parameter corresponding to the actual one. The constructor is placed in the class you wish to instantiate. It must be public, have the same name as the class it is in; but unlike a method, it cannot have a return type. So the constructor's heading for `Names` is `public Names(String s)` where `s` is the formal parameter. Writing `public void Names(String s)` will cause an error because of the presence of `void`.

The variable `st` is declared outside any of the methods. It can be used by all the methods and is thus global. It is a *field* of the class and is called an *instance* variable[6]. The values of the instance variables are stored on the heap. In order to have `s` available to all the class's instance methods, we assign it to `st` in the constructor.

[6]In section 16 `static` variables and methods are briefly discussed; `static` variables are not stored on the heap.

method	function
`length()`	returns number of elements in string
`charAt(n)`	returns character in element `n`
`substring(n,m)`	substring from `n` to `m-1`
`toUpperCase()`	changes each lowercase letter to uppercase
`toLowerCase()`	changes each uppercase letter to lowercase

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.

USING THE CONSTRUCTOR

```
public class Names //Using the constructor; must have the same name as the class
{   private String st;//instance variable

    public Names(String s)// constructor
    {   st = s;
    }
    public void writeName(int num) //prints name num times
    {   for(int j = 0; j < num; j++)
            System.out.println(st);
    }
    public void writeVertical()
    //Prints a string vertically
    {   int len = st.length();
        for(int j = 0; j < len; j++)
            System.out.println(st.charAt(j) );
    }
    public static void main(String[] arg)
    {   Names obj = new Names("Jane");
        obj.writeName(3);
        obj.writeVertical();
    }
}
```

Figure 2.10a. Once "Jane" is passed to the constructor, it is assigned to the instance variable `st` and can then be used by the methods of the object. The constructor cannot have a return type.

```
Jane
Jane
Jane
J
a
n
e
```

Figure 2.10b. Running the program of Figure 2.10a

Instance variables' accessability should always be `private`; hence, `private String st`. In general, the driver should only access the object through it's public methods; it should not be able to change any of the instance variables. For instance, if a user wanted to change `st` in the driver, he could not. In fact, the appearance of `ob.st` in the driver would cause a compilation error. When we instantiate a class without using a parameter, as we did in `Source obj = new Source()`, and do not write a constructor, Java generates one. It is called the *default* constructor because it's the one used when none is specified.

In Figure 2.11a, the class `Names` from the previous program is instantiated for two objects. Now there are two pointers to the heap as depicted in the Table for Figure 2.11a.

st	...	st
obj1		obj2

Table for Figure 2.11a. There are two locations for `st`, one for each instance.

This is why you use objects. Each time an object of that class is instantiated its reference can access all the public methods of the class and through them the object's own instance variables.

Once an object has been instantiated, unless it's reinstantiated it remains in effect for the entire execution of the program, and thus can be used anywhere before the end of the program. There may be many methods in the instantiated class and when each reference is dotted, the indicated method is executed for the proper object. Thus in `obj1.writeVertical()`, the method `writeVertical` is executed for "Jane" and in `obj2.writeVertical()`, it's executed for "Jon".

The driver for `Names` is now in its own class. Java executes the `main` method in Figure 2.11a and ignores the `main` method in Figure 2.10a,.

Once an object has been instantiated, it can be reinstantiated, e.g.,

```
Test one = new Test(3);
one = new Test(5);
```

as shown in Figure 2.12. The reference `one` now points to a new address on the heap.

var	...	var
inaccessible		one

The old location is now inaccessible but still consumes heap memory. This is called *leakage* and under certain conditions can eat up a lot of memory. Part of the virtual machine called the *garbage collector* is automatically invoked and makes the original locations accessible. They can now be used to store information about a new object.

Note that once an object has been declared, as in the left hand side of `Test one = new Test(3)`, when it's reinstantiated, it is an error to redeclare it. Thus writing `Test one = new Test(5)` causes an error, so we write `one = new Test(5)`.

MULTIPLE INSTANCES OF A CLASS

```
public class Driver3 //instantiating a class twice
{ public static void main(String[] arg)
  { Names obj1 = new Names("Jane");
    Names obj2 = new Names("Jon");
    obj1.writeName(2);
    obj1.writeVertical();
    obj2.writeName(1);
    obj2.writeVertical();
  }
}
```

Figure 2.11a. The previous class (see Figure 2.10a) is instantiated for two different objects, That's what object oriented programming is about: You use the methods of an already written class for different values of the instance variable, here "Jane" and "Jon".

```
Jane
Jane
J
a
n
e
Jon
J
o
n
```

Figure 2.11b. Running the class introduced in Figure 2.10a for "jane" and then for "jon".

```
public class Test //shows second creation of an object. Creates leakage.
{ private int var;
  public Test( int num)
  { var = num;
  }
  public void printIt()
  { System.out.println(var);
  }
  static void main(String[] asd)
  { Test one = new Test(3);
    one = new Test(5);
    one.printIt();
  }
}
```

Figure 2.12. An object is instantiated twice. Memory for the 1st becomes inaccessible; but is released by the *garbage collector*.

10 OBJECT ALIAS

When one reference is assigned to another one, as is the case in `Alias obj2 = obj1` in Figure 2.13a, the address of the object is assigned to the second reference variable. Thus they both point to the same location in memory. They are in effect the same variable; `obj2` is said to be an *alias* for `obj1`. Dotting `obj2` has the same effect as dotting `obj1` as is shown in Figure 2.13b.

Using an object alias is quite helpful if you want to keep track of the location to which an object first pointed. Thus after `obj1` is assigned to `obj2`; `obj1` can be reinstantiated, as is done in `obj1 = new Alias(5)`. `object2`, however, still points to the location to which `obj1` originally pointed. This technique is used in generating a data structure called a *linked list*.

OBJECT ALIAS

```
public class Alias
//When an object is assigned to another object, they both point
//to the same location on the heap
{ private int var;
  public Alias( int num)
  { var = num;
  }
  public void printIt()
  { System.out.println(var);
  }
  static void main(String[] asd)
  { Alias obj1 = new Alias(3);
    Alias obj2 =  obj1;

    System.out.print("Using obj1 to invoke printIt() yields ");
    obj1.printIt();

    System.out.print("Using obj2 to invoke printIt() yields ");
    obj2.printIt();

    System.out.println("----------------------------------------");

    obj1 = new Alias(5); //reinstantiate obj1
    System.out.print("Using obj1 to invoke printIt() yields ");
    obj1.printIt();

    System.out.print("Using obj2 to invoke printIt() yields ");
    obj2.printIt();
  }
}
```

Figure 2.13a. If a reference variable is assigned to another one, they both point to the same location on the heap. Thus `obj1` and `obj2` are in effect the same variable. If, however, `obj1` is reinstantiated and thus points to a new location, `obj2` continues pointing to the original location.

```
Using obj1 to invoke printIt() yields 3
Using obj2 to invoke printIt() yields 3
----------------------------------------
Using obj1 to invoke printIt() yields 5
Using obj2 to invoke printIt() yields 3
```

Figure 2.13b. Running the program of Figure 2.13a. Originally both `obj1` invoking `printIt()` and `obj2` invoking it yield the same results.

11 PITFALLS

To see what happens if `st` is not private, let's suppose that it's public in the program of Figure 2.10a and in Figure 2.14a change its value in the driver in `obj.st ="Jan"`: The results, as shown in Figure 2.14b, are now incorrect. The instance variable has been changed; the aim of the program has been subverted! Now you should be able to fully appreciate one of the principles of object oriented programming, *encapsulation*. Encapsulation dictates that all the variables used internally by a class be private. Access to them can only be through the public methods of the class.

A common error is to redeclare the instance variable in the constructor, by writing `String st = s` as is shown in Figure 2.15. Then `st` here would be a local variable and would not be the same variable as the instance variable `st`. This process is called *shadowing* and results in the instance variable being undefined. If the type of an undefined instance variable is one of the numerical primitive types, then it's value is zero. On the other hand, if it's type is a class, as is the case here with a `String` variable, it's value is set to the *null pointer*. The null pointer does not point to any address and therefore cannot be dereferenced. Trying to, as in `string1.charAt(j)`, causes one of the banes of Java programmers, the dreaded *NullPointerException* error.

THE PERILS OF NON-ENCAPSULATION

```
public class Driver4
{   public static void main(String[] arg)
    {  Names obj = new Names("John Doe");
       obj.st ="Jan";//accessing a public instance variable
       obj.writeName(2);
       obj.writeVertical();
    }
}
```

Figure 2.14a. If the instance variable in Figure 2.10a is `public`, it can be accessed from outside of the instantiated class and the goal of the program is subverted! Instead of the results being printed for the intended string *John Doe*, they are printed for *Jan*.

```
Jan
Jan
J
a
n
```

Figure 2.14b. The wrong results are printed.

REDECLARING INSTANCE VARIABLES

```
public class Shadow
{   private String st; //this is the instance variable
    public Shadow(String s)
    {    String st = s;//causes the instance variable to be null
    }
    public void writeName(int num) //prints string num times
    {    for(int j = 0; j < num; j++)
             System.out.println(st);
    }
    public void writeVertical() //prints string vertically
    {   int len = st.length();
        for(int j = 0; j < len; j++)
            System.out.println(st.charAt(j) );
    }
}
class Driver2
{   public static void main(String[] arg)
    {   Shadow obj = new Shadow("John Doe");
        obj.writeName(5);
        obj.writeVertical();
    }
}
```

Figure 2.15. Because `st` is erroneously redeclared in the constructor, the instance variable `st` is undefined and is set to the `null` pointer. Dereferencing the `null` pointer causes a *NullPointerException* error during execution of `st.charAt(j)`.

Blank page.

In Figure 2.16a we further learn how to design a class. Here we want to print horizontally one character at a time with `num` intervening spaces, the string passed to the constructor. Then print it vertically with one character per line. To accomplish the former we nest a loop in `writeHoriz` which prints `num` spaces for each character in the string. We follow the outside loop with a `println`; otherwise new output would be printed on the same line as `writeHoriz`'s output. The driver for this class is shown in Figure 2.16b.

In this class there are two instance variables; `string1` which stores the string passed to it through the constructor, and `len` the length of the string. The length is calculated once in the constructor and is accessible to all methods of the class. Had we not made `len` an instance variable, we would have to determine its value in each of the methods. Figure 2.16c shows the running of the program,

DESIGNING A CLASS

```
public class Display
//prints a string horizontally and vertically
{   private String string1;
    private int len;

    public Display(String s)//The constructor
    {   string1 = s;
        len = string1.length();
    }

    public void writeHoriz(int num)//inserts num blanks after each character.
    { for(int j = 0; j < len; j++)
        {  System.out.print(string1.charAt(j));
           for(int k = 0; k < num; k++)
              System.out.print(' ');
        }
        System.out.println();// allows next output to begin on a new line
    }

    public void writeVertical() //prints string vertically
    {    for(int j = 0; j < len; j++)
            System.out.println(string1.charAt(j) );
         System.out.println();
    }
}
```

Figure 2.16a. Prints a string with num embedded blanks between each character. Then prints it vertically. Both `string1` and `len` are instance variables and may be different for each instance. The value of `len` is used in both methods, so we make it an instance variable and calculate it in the constructor.

THE DRIVER FOR THE CLASS DISPLAY

```
public class Driver3
{       public static void main(String[] arg)
        {   Display obj1 = new Display("Jon");
            Display obj2 = new Display("Jay");
            obj1.writeHoriz(3);
            obj1.writeVertical();
            obj2.writeHoriz(1);
            obj2.writeVertical();
        }
}
```

Figure 2.16b. The driver for class `Display`. Since this is in a separate file, it must be saved as `Driver2.java`. Both `Driver3` and `Display` should be in the same directory for the program to executed simply.

```
J   o   n
J
o
n

J a y
J
a
y
```

Figure 2.16c. Running the program of Figure 2.16b.

12 USING this TO REFER TO THE IMPLICIT PARAMETER

We rewrite `writeHoriz` of Figure 2.16a so that the nested loop is now done in another method, `printBlanks`. In any non-static method you can call any other method in the same class simply by inserting the method name and if necessary, the parameters. So we could write `writeHoriz` as

```
public void writeHoriz(int num)
{       for(int j = 0; j < len; j++)
        {    System.out.print(string1.charAt(j));
             printBlanks(num);
        }
        System.out.println();
}
```

where `printBlanks` is given in Figure 2.17. In Figure 2.16b, two different objects were referenced by `obj1` and `obj2` respectively. Now when `obj1.writeHoriz(2)` is executed, `printBlanks` is executed for `obj1`; and when `obj2.writeHoriz(3)` is executed, `printBlanks` is executed for `obj2`. The reference `obj1` (or `obj2`) is called the *implicit* parameter since it doesn't appear explicitly in `writeHoriz`. To clarify that `printBlanks` is executed for the object dotted with `writeHoriz`, write `this.printBlanks(num)`. Here `this` refers to the object that is the implicit parameter. It's use here is optional and can therefore be omitted.

13 OVERLOADING THE CONSTRUCTOR

We've just seen that when `this` is used to refer to the implicit parameter, its appearance is optional. In Figure 2.18, it's not: You can initialize an instance variable by explicitly mentioning it at the instantiation, as when `string1` is initialized to "John" in `Display1 obj1 = new Display1("John")`, or have it initalized to another value by writing two versions of the constructor. One with a parameter, as we did in the previous few programs

```
public Display1(String s)
{    string1 = s;
     len = string1.length();
}
```

and one without a parameter.

```
public Display1()//overloaded constructor
{     this("Mary");//calls the other constructor
}
```

Here `this` followed by a parameter refers to the constructor (with the same name) that has one parameter and its type is `String`. Thus when no parameter is used to call the constructor, the constructor with the heading `public Display(String s)` is executed with the value of `s` being `"Mary"`. This is another case of overload – since both constructors have the same name. (Methods can be overloaded too.) The compiler determines

USING this, TO REFER TO THE IMPLICIT PARAMETER

```
    private void printBlanks(int n )
    {    for(int k = 0; k < n; k++)
                System.out.print(' ');
    }
    public void writeHoriz(int num)
    {   for(int j = 0; j < len; j++)
        {   System.out.print(string1.charAt(j));
            this.printBlanks(num);//implicit parameter
        }
        System.out.println();
    }
```

Figure 2.17. Rewriting `writeHorizontal` so that the inner loop is replaced by `printBlanks`. `this` refers to the object for which the class was instantiated. The object is called the *implicit* parameter since it doesn't appear explicitly in the method. Using `this` is optional; it can be omitted.

OVERLOADING THE CONSTRUCTOR

```
public class Display1 //overloading the constructor
{   private String string1;
    private int len;
    public Display1()//overloaded constructor
    {     this("Mary");//calls the other constructor
    }
    public Display1(String s)
    {    string1 = s;
         len = string1.length();
    }
    private void printBlanks(int n )//the entire method should be inserted here
    public void writeHoriz(int num) //the entire method should be inserted here
    public void writeVertical()     //the entire method should be inserted here
}
class Driver2
{   public static void main(String[] arg)
    {   Display1 obj1 = new Display1();
        obj1.writeHoriz(2);
        obj1.writeVertical();
    }
}
```

Figure 2.18. When no parameter is indicated in the constructor, the constructor with no parameters calls the one-parameter constructor with `this`("Mary"). When there is more than one constructor, the compiler determines which one will be invoked based on the number and type of parameters used during the instantiation. This is called *early binding* since it's done at compile-time.

which constructor will be invoked based on the number of parameters and their order and type used in the instantiation. This process is called *binding* and since it's made at compile-time, it's called *early binding*. Thus `Display1("John")` would bind with the one-parameter constructor and `Display1()` would bind with the parameter-less constructor. In both methods and constructors, their *signature* consists of their name, the number of formal parameters and their order and type but not the return type. Thus the two constructors in Figure 2.18 have different signatures. If, however, two methods on the one hand, or two constructors on the other, have the same signature, a compilation error will occur.

If the constructor's parameter is the same as the instance variable, `this` can be used to differentiate them. So if the constructor in Figure 2.16a is written as

```
public Display(String string1)
{    this.string1 = string1;
     len = string1.length();
}
```

`this.string1` indicates that the left side of the assignment statement refers to the instance variable. Why? Because it represents the object dotted with the class's field, i.e., the instance variable,

14 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. 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 byes. 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 succesive digit is assigned a higher numerical code than the previous one, and these codes are contiguous.

An `int` occupies four bytes of memory. Since the range of `char` values is narrower than that of `int`s, `char` values can be assigned to `int`s 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 literal if the value assigned to `i` is less than 65535 to a `char` variable: If the integer literal 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 a numerical operation. 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.

Control character sequence	Function
`\b`	backspace
`\n`	new line
`\r`	carriage return
`\t`	tab

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 hexidecimal number. For instance, the escape sequence `\u0007` is the Unicode representation for the ASCII code 7, the loudspeaker beep.

15 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.

16 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

```
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);
        System.out.println();
    }
}
```

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`.

```
abcdefghijklmnopqrstuvwxyz
zyxwvutsrqponmlkjihgfedcba
```

Figure 2.19b Running the program of Figure 2.19a. The second loop runs backwards because `letter--` decrements the loop index.

REVERSING A STRING–STATIC METHODS

```
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 `t`. Since `reverse` is static we access it by dotting the class name with the method, `Int.reverse(sample)`

j	`s.charAt(j)`	original `t`	final `t`
3	d	empty	d
2	c	d	dc
1	b	dc	dcb
0	a	dcb	dcba

Table for Figure 2.20a Shows how the program builds the string `t` using `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.charAt(j) + t` the character precedes the string in the concatenation. How this evolves is shown in the Table for Figure 2.21.

j	s.charAt(j)	original t	final t
0	a	empty	a
1	b	a	ba
2	c	ba	cba
3	d	cba	dcba

The Table for Figure 2.21. Shows how the final string evolves from `t = s.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.

s.charAt(j)	code for s.charAt(j)	code for '0'	digit	original sum	final sum
1	49	48	1	0	1
2	50	48	2	1	3
3	51	48	3	3	6
4	52	48	4	6	10
5	53	48	5	10	15

Table for Figure 2.22. Shows how the addition of the digits work using `digit = s.charAt(j) - '0'; sum = sum + 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 thowing 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

```
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

```
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 `sum = 10*sum + 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.

`digit`	original `sum`	`10*sum`	final `sum`
1	0	0	1
2	1	10	12
3	12	120	123
4	123	1230	1234
5	1234	12340	12345

Table for Figure 2.23. Shows how `digit = s.charAt(j) - '0'; sum = 10*sum + digit` works.

17 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 `long lo = 123456789012`, causes a compilation error unless the number is immediately followed by an uppercase or lowercase "L". So `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 `float fl = d` causes the "possible loss of precision" error. To correct it, cast `d` as a `float` by writing `float fl = (float)d`. This doesn't increase the precision, it justs warns you of the loss of it. All floating pointing values unless indicated otherwise are considered double precision. Thus `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 `float fl = 12.3f`. To assign the floating points types to any of the integer types you must use casting, as in `long lo = (long)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 `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);
        System.out.println();
        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

Type	range	bits
`byte`	-128 to 127	8
`short`	-32768 to 32767	16
`int`	-2147483648 to 2147483647	32
`long`	-9223372036854775808 to 9223372036854775807	64
`char`	0 to 65535	16
`float`	1.4E-45 to 2.4028235E38	32
`double`	4.9E-324 to 1.7976931348623157E308	64

Table 2.7

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)`. The first actual parameter has to be an `int` or be narrower than an `int`. Since 'a' is a `char`, it's ok. The second parameter has to be compatable with a `float` and the `int` 123 is. The third parameter has to be compatable with a `char`. In an assignment statement, 97 would be compatable with a `char`; but as a parameter, it isn't. So we must cast it as a `char`.

18 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 envolved.

PASSING PARAMETERS

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

Figure 2.24a. The type of the actual parameters must be compatable 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 `xint`. 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

Definition	Operators
parentheses	( )
postfix operators	[] x++ x- -
unary operators	+x -x
creation or cast	new (type)x
multiplicative	* / %
additive	+ -
relational	< > >= <=
equality (boolean)	= = !=
assignment	=

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%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.

`number`	`digit=number%10`	`reverse`	`10*reverse + digit`
341	1	0	1
34	4	1	14
3	3	14	143

The Table for Figure 2.25a.

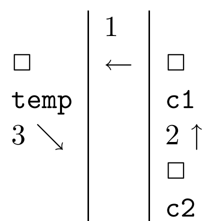
In `len = ("" + number).length()`, the `int` parameter is converted to a string and then its length is determined. This indicates the number of digits in the integer and thus the number of iterations in the loop. If there are too many iterations, the value of `reverse` will have trailing zeros, e.g.,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

19 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 `WIDTH/(m+n)` times, an integer, so we write `int width = WIDTH/(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 succesive 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



	1	
□	←	□
`temp`		`c1`
3 ↘		2 ↑
		□
		`c2`

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

```
 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.

```
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

20 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.

21 CREATING WRAPPER OBJECTS, POLYMORPHISM

You can assign a primitive type value to a primitive type variable, or assign a reference to another one. You can't assign a primitive type value to a reference type[7]. To overcome this, you "wrap" the primitive in the appropriate wrapper class, i.e, you create an object from the primitive. To do this follow the appropriate wrapper class name with a reference, insert "=", follow that with `new`, and end with the class name and in parentheses insert the primitive value, e.g., `Float floatObj = new Float(122.45)`. Now the object, here `floatObj`, represents the floating value and can be dotted with the non-static methods of the class. One such method is `floatValue`, i.e., `floatObj.floatValue()` here returns the original `float`, 122.45. Another non-static method is `hashCode`. The hash code is used to store information in tables so that data can be retrieved very quickly. In the wrapper classes, `hashCode` calculates an integer value to be used as a hash code for a primitive type. For other classes, it does the same thing for an object.

The program in Figure 2.27a instantiates objects of the `Float, Integer` and `Character` classes, and finally, an object of the `String` class. These objects are then used as actual parameters for `printHash`. Note that the formal parameter is of type `Object`. The `Object` class because it's the parent of all classes, is the widest of all classes – any class can be assigned to it. The first call, `printHash("Float", floatObj)` is eqivalent to `Object obj = floatObj`. When `obj` is dotted in `obj.hashCode()`, Java uses the version of `hashCode` for the `Float` wrapper class. In the second call, `printHash("Integer", intObj)`, Java uses the version of `hashCode` for the `Integer` wrapper class. The type of the implicit parameter `obj` in `obj.hashCode()` determines which version of `hashCode()` is used at run-time. How it's done is called *dynamic binding* or *late binding* because it's done at run-time; at compile-time, the Java compiler doesn't know which object to use to invoke `hashCode()`. Figure 2.27b shows the running of the program.

[7] In JDK v1.5 and higher versions you can assign a primitive type to a reference.

```
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**
***XX***XX***XX***XX***XX***XX***XX***XX***XX***XX
XXX**XXX**XXX**XXX**XXX**XXX**XXX**XXX**XXX**XXX**
***XX***XX***XX***XX***XX***XX***XX***XX***XX***XX
```

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

POLYMORPHISM

```
public class MakeHash
{    public static void printHash(String name, Object obj)
    {   System.out.println(name + " hash conversion:"+ obj.hashCode() );
    }
    public static void main(String[] asd)
    {   Float floatObj = new Float(122.45);
        printHash("Float", floatObj);
        Integer intObj = new Integer(12345);
        printHash("Integer", intObj);
        Character charObj = new Character('z');
        printHash("Character", charObj);
        String stObj = "hello";
        printHash("String", stObj);
    }
}
```

Figure 2.27a. The `Object` class is the parent of all classes and hence the widest. You can assign objects of every other class to it.

```
Float hash conversion:1123477094
Integer hash conversion:12345
Character hash conversion:122
String hash conversion:99162322
```

Figure 2.27b. Running the program of Figure 2.27a. Hashcodes are used to find items in memory quickly. The type of the implicit parameter determines which version of `hashCode` is used. So if the implicit parameter is a `Float`, the version of `hashCode` used will be the one for `Floats`. This is called *polymorphism*.

22 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 hexidecimal 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;
    }
}

class Driver3
{   public static void main(String[] asd)
    {   StringTest st = new StringTest('z');
        String s = "w/concatenation: " + st;
        System.out.println( st);
        System.out.println( s);
}
}
```

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.

```
character used is z
w/concatenation: character used is z
```

Figure 2.28b. Running Figure 2.28a.

THE toString METHOD IN THE Character CLASS

```
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.

23 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.

Function	Explanation	Return type
`abs(x)`	determines the absolute value	same as argument
`cos(x)`	x is a double & in radians	double
`exp(x)`	raises x to the base e	double
`log(x)`	x is a double	double
`max(x, y)`	maximum of two values	same as argument
`min(x, y)`	minimum of two values	same as argument
`random()`	$0 \leq \text{result} < 1$	double
`sin(x)`	x is a double & in radians	double
`sqrt(x)`	x is a double	double
`tan(x)`	x is a double & in radians	double
`toRadians(x)`	x in degrees	double

24 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

```
import javax.swing.JOptionPane;
public class Test
{  public static void main(String[] asd)//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.

25 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 `int first = (int)(2*Math.random() )`. We then send `first` to method `isMatch` where a second integer, either 0 or 1, is generated. In `boolean match = second == first` since the test for equality (==) has a higher precedence than the assignment (=), the boolean expression `second == first` is evaluated as true or false, depending on whether the two generated digits are equal or not. The result is assigned to `match` and then returned by the method. Since a primitive type is returned, we can place `isMatch` in a print statement in the `main` method and either true or false is printed. For every four pairs of digits generated, on the average 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 `return second == first` and eliminate the `match` variable.

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

26 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.

SOLVING THE PROBLEM, STEP I

Let's first focus on sorting the digits. Perhaps the word "sorting" makes our task too difficult. So let's reduce it to finding the smallest, middle and largest digits in the three-digit integer. First we separate the digits using the `%` and the `/` operators.

```
first = number%10;
number = number /10;
second = number% 10;
third = number /10;
```

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

FINDING THE MAXIMUM AND MINIMUM OF THREE VALUES

```
public static int largest(int one, int two, int three)
{   return Math.max(one, Math.max(two, three));
}

public static int smallest(int one, int two, int three)
{   return Math.min(one, Math.min(two, three));
}
```

Figure 2.31a. Using the `Math` class `max(x, y)` to find the maximum of three digits. The minimum of three digits is found the same way,

The `Math` class has a method that finds the larger of two digits `x` and `y`, and one that finds the smaller, namely, `max(x,y)` and `min(x,y)` respectively; but unfortunately they work with two arguments only. We, however, can use them to write our own methods `largest` and `smallest` as shown in Figure 2.31a, where, for instance, the statement `Math.max(one, Math.max(two, three))` finds the maximum of three integers. What have we done so far? We've broken the problem down into its smallest component and solved part of it.

The next question is "Can we use the tools we've developed so far to find the middle digit? After a little thought (O.K., a lot of thought) we realize we know how to find the sum of the digits in a string using method `sumDigits` shown in Figure 2.22. So if we can find the sum of three digits and know two of them, we know the third as is shown in Figure 2.31b. This is all done in method `generate`. When we write this method, we code a little and then test what we have written. To be sure that no mistakes have been made, insert `println` statements where appropriate.

GENERATING THE SORTED NUMBER, STEP II

Now that we have the three digits, we generate the number with the digits in ascending order using `smallNum = 100*small + 10*middle + big`; find the reverse of this (the digits in descending order) using `bigNum = ReverseInt.intReverse(smallNum)` as is also shown in Figure 2.31b.

THE PROBLEM SOLVED?, STEP III, TESTING THE PROGRAM

In `number = generate(number)`, the method `generate` returns the difference between `bigNum` and `smallNum`, assigns it to `number`, and repeats the calculation using this new value, as shown in Figure 2.31c. Figure 3.31d shows that the results converge to 495. At this point we may think that we're finished; but we haven't checked the program for other input. What happens if one of the digits is zero, for example 109. Figure 2.31e shows the results of running the program with this value. Note that the original value of `small` printed is 19, which is actually 019. The value of `big` should be 910; but instead it's 91. What happened? Instead of calculating `bigNum` the same way we calculated `smallNum`, we took a shortcut and simply reversed the digits in `number` without including the leading zero in the process. This is called a *logical error* because it results from a defect in our logic. Logical errors occur during run-time; whereas, syntax errors occur at compilation-time.

THE PROBLEM SOLVED, STEP IV

To solve the problem calculate `bigNum` digit by digit in `bigNum = 100*big + 10*middle + small`. This produces the correct answer. To test the program further, use different combinations of three-digit numbers that include one zero and one that includes two zeros. They all produce the correct result. Even any two-digit or any one-digit number input produces output that converges to 495 because the program assumes three-digit input and accordingly, when necessary, inserts leading zeros. Let's see how.

Let's say that `number` is 8. In

```
first = number%10;
number = number /10;
second = number% 10;
third = number /10;
```

`first` will be 8 and `number`, 0. This means that `second` and `third` which represent the leading digits, will be 0; but, it also means that `bigNum` will be 800. So the calculation precedes as usual.

A NUMBER THEORY PROBLEM

```
public static int generate(int number)
{   int difference, big, small, reverse, sum, smallNum, bigNum;
    int first, second, third, middle;

    first = number%10;
    number = number /10;
    second = number% 10;
    third = number /10;
    big = largest(first, second, third);
    small = smallest(first, second, third);
    sum = Int.sumDigits("" + number);
    middle =sum - big - small;
    smallNum = 100*small + 10*middle + big;
    bigNum = ReverseInt.intReverse(smallNum);
    System.out.println("big = " + bigNum + "; small = " + smallNum);
    difference = bigNum - smallNum;
    System.out.println("difference " +difference);
    number = difference;
}
```

Figure 2.31b. We find the middle digit by subtracting the largest and smallest from the sum of the three digits. The digits in a 3-digit `int` are sorted forming two `int`s one with the digits in ascending order and one with them in descending order. Subtract the former from the latter.

```
import javax.swing.JOptionPane;
public class Generate
{//Sorts digits forming smallNum and bigNum; calculate  bigNum- smallNum.

    public static int generate(int number)//insert method here
    public static int largest(int one, int two, int three)//insert method here
    public static int smallest(int one, int two, int three)//insert method here

    public static void main(String[] asd)
    {   final int MAX = 5;
        String str = JOptionPane.showInputDialog("Type your number");
        int number = Integer.parseInt(str);
        System.out.println("Original integer " + number);
        for(int j = 0; j < MAX; j++)
            number = generate(number);
        System.exit(0);
    }
}
```

Figure 2.31c. The process described in Figure 2.21b is repeated on `number` `MAX` times.

```
 491
big = 941; small = 149
difference 792
big = 972; small = 279
difference 693
big = 963; small = 369
difference 594
big = 954; small = 459
difference 495
big = 954; small = 459
difference 495
```

Figure 2.32d. Running the program of Figure2.31c. The differences converges rapidly to 495.

```
109
big = 91; small = 19
difference 72
big = 72; small = 27
difference 45
big = 54; small = 45
difference 9
big = 9; small = 9
difference 0
big = 0; small = 0
difference 0
```

Figure 2.32e. Running the program of Figure2.31c with input of 109. The results are wrong! Note that the original value of `smallNum` is 19 which is actually 019; the value of `bigNum` should be 910 but instead it's 91 because we used the wrong method to calculate it: We reversed the digits in 19.

27 POLYMORPHISM REDUX

In Java, in order to make a sorting method reusable, the array (let's call it **x**) which is the formal parameter of the method should be an array of **Comparable**, e.g.,

```
void sort(Comparable[] x)
```

This way the sort can be done for an array of any class (let's call it **y**) which **implements Comparable**. That array is used as the actual parameter of the sorting method, e.g.,

```
String[] y = new String[n];
sort(y);
```

In our example polymorphism causes the comparison of elements of the array to be done at *run-time* using the **compareTo** method of the **String** class in a statement like **x[i].compareTo(x[i+1])**. Thus the type of the implicit parameter **x[i]**, which here is the same type as the actual parameter **y** in **sort(y)**, determines which **compareTo** method is used. If, however, the class of the implicit parameter does not implement **Comparable**, the Java Virtual Machine goes up the inheritance chain until it finds the super class that does. If it doesn't find such a class, it produces an *incompatible types* compilation error.

Polymorphism occurs whenever an implicit parameter is dotted with a method that can take different forms, all of which have the same signature. Examples of other methods that are in this category are the **toString()** and **hashCode()** methods.

Using an array of **Object** as the formal parameter of the sorting method would cause a compilation error since the **Object** class does not have a **compareTo** method.

Comparable is an example of an abstract entity called an **interface**. An interface does not contain any concrete methods; for instance, the **Comparable** interface contains only the signature of the **compareTo** method. Until Java 1.5 the signature was **public int compareTo(Object o);** If you write a class that implements **Comparable**, you are forced to write a **compareTo** method and it must have the same signature described in the interface. This insures that the **compareTo** method you write produces a result compatable with the use of the **compareTo** method in the sort method.

28 THE COMPARABLE INTERFACE

Prior to Java 1.5 an example of a class that implements **Comparable** would be

```
class Huffman implements Comparable
{
    int freq;
    String letter;

    public int compareTo(Object obj)
```

```
    {
        Huffman h1 = (Huffman)obj;
        return this.letter.compareTo(h1.letter);
    }
}
```

The **this** with the dot can be eliminated.

Java 1.5 introduced generics so that errors that used to be found at run-time could now be detected at compile-time. In the case of the **Comparable** interface, generics also simplifies the programming. The **Comparable** interface now is:

```
public interface Comparable<T>
{
       public int compareTo(T obj);
}
```

The **T** indicates the name of the class in which the **compareTo** method is used. The **Huffman** class rewritten to that it takes advantage of generics has the following simplified form:

```
class Huffman implements Comparable<Huffman>
{
    int freq;
    String letter;

    public int compareTo(Huffman h1)
    {
      return letter.compareTo(h1.letter);
    }
}
```

Java 1.5 allows you to use the **Comparable** interface with or without generics.

29 CHAPTER REVIEW

Exercises

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.