

## Primitive Data Types

The building blocks for all objects in Java are the *primitive data types* in the Java language. In this section, we'll meet Java's primitive data types and learn how to name, store, retrieve, assign, and operate on these types. We'll introduce the concept of a *variable*. A variable provides a convenient way to name an instance of a data type. For a variable to exist, however, it first must be *declared*. For a variable to be used, it first must have a value *assigned* to it. To perform meaningful programming tasks with variables, we need a set of *operations* that can be performed with variables. Strings are not primitive data types in Java; however, their treatment is so special that we include them in this chapter. These and other concepts are introduced in this chapter.

Java includes the following eight primitive data types:

<code>int</code>	an integer number (32 bits)
<code>long</code>	an integer number (64 bits)
<code>short</code>	an integer number (16 bits)
<code>byte</code>	an integer number (8 bits)
<code>double</code>	a real number (64 bits)
<code>float</code>	a real number (32 bits)
<code>char</code>	a character (16 bits)
<code>boolean</code>	a boolean (1 bit)

The first six are examples of numbers, and we'll begin with these.

### ***Integers and Floating-Point Numbers***

There are two general categories of numbers: integers and real numbers. Integers are whole numbers and have no fractional or decimal component. So, 2 is an integer, but 2.718 is not. Many quantities are adequately represented by integers. For example, the number of passengers in a car could be 4 or 5, but not 4.5. Many quantities, however, need a fractional or decimal component, such as the ratio of the circumference of a circle to its diameter, known as *pi*. For these quantities, a real number is required. Because of the way numbers are stored and represented in computer systems, real numbers are more commonly called *floating-point numbers*.

In the preceding list, the first four data types are integers and the next two are floating-point numbers. The most common type of integer is `int`, but Java also supports long integers (`long`), short integers (`short`), and byte integers (`byte`). The most common type of floating-point number is `double`, for "double-precision", but Java also supports single-precision floating-point numbers (`float`).

As you might guess, the variations on integers and floating-point numbers differ in their size and internal representation on a computer. From a numeric perspective, they differ in their *range* and *precision*. Range is the spread between the smallest and largest quantity represented. Precision is the closeness to a single quantity that can be represented. For integers, the precision is always one. For floating-point numbers, the precision is the number of significant, or meaningful, digits

that can be represented. To use  $\pi$  as an example, the three digits 3.14 may be adequate to some, but mathematicians can quantify  $\pi$  with hundreds of digits of precision. Unfortunately, such precision is not possible for the primitive type `double` in Java. With a `double`, we get about 15 digits of precision. So, the best we can do for  $\pi$  is about 3.141592653589793.<sup>1</sup> **Table 1** summarizes the range and precision for Java's integer and floating-point data types.<sup>2</sup>

**Table 1. Range and Precision for Primitive Data Types**

Type	Bits	Range		Precision
		Smallest	Largest	
<code>int</code>	32	-2147483648	2147483647	1
<code>double</code>	64	4.9E-324	1.7976931348623157E308	15 digits
<code>long</code>	64	-9223372036854775808	9223372036854775807	1
<code>short</code>	16	-32768	32767	1
<code>byte</code>	8	-128	127	1
<code>float</code>	32	1.4E-45	3.4028235E38	7 digits

The ranges for floating-point numbers are shown in scientific notation in **Table 1**, where " $E_n$ " means  $10^n$ . A `double` variable can be as small as  $-4.9 \times 10^{324}$  or as large as  $+1.8 \times 10^{308}$  with about 15 digits of precision.

The number of bits in each primitive data type determines the memory required to hold data of that type. To store one thousand numbers, each as a `double`, for example,  $1,000 \times 64 = 64,000$  bits = 8,000 bytes of memory are needed. If only a few digits of precision are necessary, storing each value as a `float` is a space-saving option to consider. In most cases today, memory is plentiful and cheap; so, conserving space may not be an issue.

In comparing integers with floating-point numbers, the speed of operations should be considered. Primitive operations such as multiplication and division are considerably faster with integers than with floating-point numbers, and this should be considered when choosing a data type to store information. Today's desktop computers have extremely high frequency CPUs, so execution speed may not be an issue. Bear in mind, however, that memory requirements are important for very large quantities of data, and execution speed is important for programs that perform vast and complex computations.

### Characters

Besides numbers, computers must store and manipulate characters, such as letters of the alphabet, digits, punctuation symbols, etc. The primitive data type `char` serves this purpose. In Java a character is coded as the symbol for the character enclosed in single quotes. Examples include `'A'`, `'a'`, `'9'`, `'%'`, and `'='`.

Since memory locations store a series of bits, each with state 0 or state 1, there is no obvious way to store characters in a computer's memory. The trick is to devise a coding system whereby each

<sup>1</sup> This is the value assigned to the constant `Math.PI` in Java's `Math` class in the `java.lang` package.

<sup>2</sup> The values in **Table 1** were obtained by printing the defined constants in Java's wrapper classes. See the program `FindRanges.java` for more details.

character is represented by a certain pattern of bits. A well-established coding system for characters is the [American Standard Code for Information Interchange](#), or [ASCII](#) (pronounced *ass-key*). ASCII is a 7-bit coding system. Of its  $2^7 = 128$  codes, 95 are for graphic symbols (e.g., letters, digits, punctuation), and 33 are for control characters (e.g., enter, tab, delete, end-of-file).

Unfortunately, 7-bit ASCII codes provide little room for expansion. International requirements of computing technology necessitate a coding system that can accommodate all the major scripts of the world. A 16-bit coding system known as the [Unicode Standard](#) was developed by an industry consortium to meet such needs. The ASCII character set is a subset of Unicodes, occupying the first 128 positions in the Unicode set. For example, the character 'A' is represented by the Unicode `\u0041`. The code `0041` is the hexadecimal value corresponding to the letter A. This is equivalent to the decimal value 65.

## Escape Sequences

Besides graphic characters that are printable, the `char` data type can hold values that control input/output devices or modify the appearance or layout of information. These are known as [control characters](#). Since control characters are not printable, they cannot be scripted as a symbol enclosed in single quotes, like other characters. Instead they are represented as [escape sequences](#). An escape sequence is a two-character pattern beginning with the backslash character (`\`). The most common Java escape sequences are given in Table 2.

Table 2. Java Escape Sequences

Escape Sequence	Interpretation
<code>\b</code>	backspace
<code>\t</code>	tab
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\"</code>	"
<code>\\</code>	<code>\</code>
<code>\'</code>	'

So, the tab character is coded as `'\t'`, the newline character as `'\n'`, and so on.

## Booleans

A `boolean` data type can take on one of two values: `true` or `false`. Since only two states must be represented, only 1 bit of memory storage is required to store a boolean value. The term "boolean" is used in deference to the 19th century mathematician George Boole who pioneered the branch of mathematics known as *Boolean algebra* or *combinatorics*. Note that the terms `true` and `false` are reserved words in Java and cannot be used for other purposes.

## Variables and Constants

We are now ready to put Java's primitive data types to work. A primitive data type may appear in a program two ways: as a [variable](#) or as a [constant](#). A constant, sometimes called a [literal](#), is simply an explicit instance of a value. For example, `3` is an integer constant, `3.14` is a floating point constant, `'a'` is a character constant, and `true` is a boolean constant. Integer constants

can be expressed in hexadecimal notation, if desired. For example `0x00FF` is an integer constant equal to 255. The pattern "0x" means that what follows is interpreted in hexadecimal notation.

A variable is an abstraction that represents a value. A key difference between a constant and a variable is that a constant, once coded in a program, cannot change. A variable holds a value, but this value can change as a program executes.

Before a variable is used, it must be *declared* and it must be *initialized* to a value. The following four Java statements declare integer, floating-point, character, and boolean variables:

```
int count;
double temperature;
char c;
boolean status;
```

Each line above is an example of a Java *statement*. Note that a statement ends with a semicolon. In fact, each line above is a particular type of Java statement known as a *declaration*. The first line declares a variable named `count` of type `int`. The second line declares a variable named `temperature` of type `double`. And so on. The variable's name is chosen by the programmer. Obviously, one should choose names that suggest the variable's purpose. The name "temperature" gives a pretty good indication of what the variable represents.

So, once declared, how do we initialize a variable to a value? The following eight program statements show the declaration of variables followed immediately by the initialization of the variables:

```
// declare four variables
int count;
double temperature;
char c;
boolean status;

// initialize the four variables to values
count = 7;
temperature = 98.6;
c = 'A';
status = true;
```

Note that a single character is coded as the character symbol enclosed in single quotes. Declaration and initialization can be combined:

```
int count = 7;
double temperature = 98.6;
char c = 'A';
boolean status = true;
```

It is also possible to declare and initialize more than one variable at the same time, for example

```
int i = 3, j = 4, k = 5;
```

A declaration can appear anywhere in a Java program, provided the declaration precedes the first use of the variable in an expression. There are different opinions on where declarations should appear. Some feel they should be grouped together at the beginning of a method or a block of code. Others feel they should appear close to their first use. It is largely a matter of personal style, and we'll not adhere strictly to either approach in these notes.

## Comments

Comments are notations included in a source program to help explain an operation. There are two ways to include comments in Java programs. When two forward slash characters ( // ) appear on a line, the rest of the line is a comment and is ignored by the compiler. This technique is useful for short comments, such as a one-line comment or a comment appended to the end of a program statement. Long comments, perhaps spanning many lines of source code, can use " /\* " and " \*/ " to delimit the comment, such as

```
/* this is a comment */
```

Comments that span many lines are usually called a *comment block*.

## Identifiers and Reserved Words

Whenever a programmer creates a name for a variable or other component of a program (e.g., the name of a class), the result is an *identifier*. So, in the preceding examples, `temperature` was an identifier, `status` was an identifier, and so on. Clearly, there must be rules on what constitutes a legal or illegal identifier. And there are. A Java identifier can consist of any letters or digits, as well as an underscore ( \_ ), or a dollar sign ( \$ ). A digit character, however, cannot be the first character in an identifier. The following are examples of legal identifiers:

```
number_of_items
inputMode
X99
temp$filename$suffix
```

The following are examples of illegal identifiers:

```
99gretzky      (Illegal: can't start with a digit)
odd-ball       (Illegal: can't use a dash)
this.that      (Illegal: can't use a period)
```

There is no size restriction on identifiers, so if you want an identifier with a hundred characters, you can have it. Java is case sensitive, however, so beware that the following two identifiers are different:

```
pressure
Pressure
```

As a programmer, you can make-up identifiers that are as cryptic or as meaningful as you wish. Obviously, well-chosen identifiers help to clarify program statements. Besides the few rules noted above, an identifier must not conflict with Java's *reserved words*. Think of the problems if an integer variable were named `float`. The statements

```
int float; // wrong!
float = 3; // wrong!
```

would make the compiler's job quite difficult. Java's 59 reserved words are summarized in [Figure 1](#).

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

**Figure 1. Java's reserved words**

### ***Naming Conventions***

Identifiers are used for more than just the names of variables. They are also used for the names of methods, classes, and defined data constants. For each of these, naming conventions exist, and adhering to these will contribute to making your code understandable. Java's naming conventions are listed in [Figure 2](#).

Type of Identifier	Examples	Convention
variable	temperature moreData	<ul style="list-style-type: none"> <li>• starts with lowercase character</li> <li>• multiple words joined together</li> <li>• first letter of words (except first) in uppercase</li> </ul>
method	length() toLowerCase()	<ul style="list-style-type: none"> <li>• same rules as for variables</li> <li>• parentheses distinguish methods from variables</li> <li>• exception: constructor methods begin with an uppercase character</li> </ul>
class	String MenuBar	<ul style="list-style-type: none"> <li>• starts with uppercase character</li> <li>• multiple words joined together</li> <li>• first letter of words in uppercase</li> </ul>
constant	FACTOR SCREEN_WIDTH	<ul style="list-style-type: none"> <li>• all characters in uppercase</li> <li>• multiple words joined with underscore character</li> </ul>

**Figure 2. Conventions for naming identifiers**