Chapter 6

Java's Building Blocks

In This Chapter

- Assigning values to things
- Making things store certain types of values
- Applying operators to get new values

I've driven cars in many cities, and I'm ready to present my candid reviews:

- Driving in New York City is a one-sided endeavor. A New York City driver avoids hitting another car but doesn't avoid being hit by another car. In the same way, New York pedestrians do nothing to avoid being hit. Racing into the path of an oncoming vehicle is commonplace. Anyone who doesn't behave this way is either a New Jersey driver or a tourist from the Midwest. In New York City, safety depends entirely on the car that's moving toward a potential target.
- A driver in certain parts of California will stop on a dime for a pedestrian who's about to jaywalk. Some drivers stop even before the pedestrian is aware of any intention to jaywalk.
- Boston's streets are curvy and irregular, and accurate street signs are rare. Road maps are outdated because of construction and other contingencies. So driving in Boston is highly problematic. You can't find your way around Boston unless you already know your way around Boston, and you don't know your way around Boston unless you've already driven around Boston. Needless to say, I can't drive in Boston.
- London is quite crowded, but the drivers are polite (to foreigners, at least). Several years ago, I caused three car accidents in one week on the streets of London. And after each accident, the driver of the other car apologized to me!

I was particularly touched when a London cabby expressed regret that an accident (admittedly, my fault) might stain his driving record. Apparently, the rules for London cabbies are quite strict.

This brings me to the subject of the level of training required to drive a taxicab in London. The cabbies start their careers by memorizing the London street map. The map has over 25,000 streets, and the layout has no built-in clues. Rectangular grids aren't the norm, and numbered streets are quite uncommon. Learning all the street names takes several years, and the cabbies must pass a test in order to become certified drivers.

This incredibly circuitous discussion about drivers, streets, and my tendency to cause accidents leads me to the major point of this section: Java's built-in types are easy to learn. In contrast to London's 25,000 streets, and the periodic table's 100-some elements, Java has only eight built-in types. They're Java's *primitive types*, and this chapter describes them all.

Info Is as Info Does

"Reality! To Sancho, an inn; to Don Quixote, a castle; to someone else, whatever!"

—Miguel de Cervantes, as updated for "Man of La Mancha"

When you think a computer is storing the letter `J`, the computer is, in reality, storing 01001010. For the letter `K`, the computer stores 01001011. Everything inside the computer is a sequence of 0s and 1s. As every computer geek knows, a 0 or 1 is a *bit*.

As it turns out, the sequence 01001010, which stands for the letter `J`, can also stand for the number 74. The same sequence can also stand for $1.0369608636003646 \times 10^{-43}$. In fact, if the bits are interpreted as screen pixels, the same sequence can be used to represent the dots shown in Figure 6-1. The meaning of 01001010 depends on the way the software interprets this sequence of 0s and 1s.

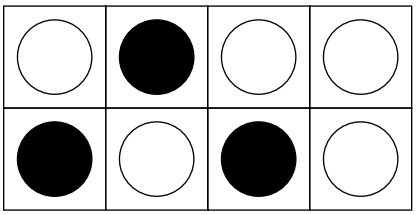


Figure 6-1: An extreme close-up of eight black-and-white screen pixels.

So how do you tell the computer what 01001010 stands for? The answer is in the concept of *type*.

The *type* of a variable is the range of values that the variable is permitted to store. Listing 6-1 illustrates this idea.

Listing 6-1: Goofing Around with Java Types

```
package com.allmycode.demos;

import javax.swing.JOptionPane;

public class TypeDemo1 {

  public static void main(String[] args) {
    int anInteger = 74;
    char aCharacter = 74;
    JOptionPane.showMessageDialog(null, anInteger,
        "An int variable", JOptionPane.PLAIN_MESSAGE);
    JOptionPane.showMessageDialog(null, aCharacter,
        "A char variable", JOptionPane.PLAIN_MESSAGE);
  }

}
```

A run of the code in Listing 6-1 looks like the displays in Figures 6-2 and 6-3.

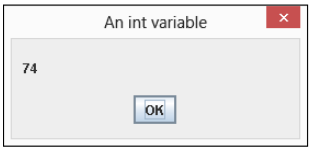


Figure 6-2: Displaying 01001010 as an `int` value.

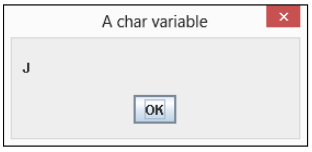


Figure 6-3: Displaying 01001010 as a `char` value.

In Figure 6-2, the computer interprets 01001010 as a whole number. But in Figure 6-3, the computer interprets the same 01001010 bits as the representation of the character `J`. The difference stems from the two *type declarations* at the start of the `main` method in Listing 6-1:

```
int anInteger = 74;
char aCharacter = 74;
```

Each of these declarations consists of three parts: a variable name, a type name, and an initialization. The next few sections describe these parts.

Variable names

The identifiers `anInteger` and `aCharacter` in Listing 6-1 are variable names, or simply variables. A *variable name* is a nickname for a value (like the value `74`).

I made up both variable names for the example in Listing 6-1, and I intentionally made up *informative* variable names. Instead of `anInteger` and `aCharacter` in Listing 6-1, I could have chosen `flower` and `goose`. But I use `anInteger` and `aCharacter` because informative names help other people read and understand my code. (In fact, informative names help me read and understand my own code!)

Like most of the names in a Java program, variable names can't have blank spaces. The only allowable punctuation symbol is the underscore character (_). Finally, you can't start a variable's name with a digit. For example, you can name your variable `close2Call`, but you can't name it `2Close2Call`.

If you want to look like a seasoned Java programmer, start every variable name with a lowercase letter, and use uppercase letters to separate words within the name. For example, `numberOfBunnies` starts with a lowercase letter and separates words by using the uppercase letters `O` and `B`. This mixing of upper- and lowercase letters is called *camel case* because of its resemblance to a camel's humps.

Type names

In Listing 6-1, the words `int` and `char` are *type names*. The word `int` (in the first type declaration) tells the computer to interpret whatever value `anInteger` has as a "whole number" value (a value with no digits to the right of the decimal point). And the word `char` (in the second type declaration) tells the computer to interpret whatever value `aCharacter` has as a

character value (a letter, a punctuation symbol, or maybe even a single digit). So in Listing 6-1, in the first call to `showMessageDialog`, when I display the value of `anInteger`, the computer displays the number `74`. And in the second call to `showMessageDialog`, when I display the value of `aCharacter`, the computer displays the letter `J`.

In Listing 6-1, the words `int` and `char` tell the computer what types my variable names have. The names `anInteger` and `aCharacter` remind me, the programmer, what kinds of values these variables have, but the names `anInteger` and `aCharacter` provide no type information to the computer. The declarations `int rocky = 74` and `char bullwinkle = 74` would be fine, as long as I used the variable names `rocky` and `bullwinkle` consistently throughout Listing 6-1.

Assignments and initializations

Both type declarations in Listing 6-1 end with an initialization. As the name suggests, an *initialization* sets a variable to its initial value. In both declarations, I initialize the variable to the value `74`.

You can create a type declaration without an initialization. For example, I can change the code in Listing 6-1 so that the first four lines inside the `main` method look like this:

```
int anInteger;
char aCharacter;
anInteger = 74;
aCharacter = 74;
```

A line like `anInteger = 74` is an *assignment*. An assignment changes a variable's value. An assignment isn't part of a type declaration. Instead, an assignment is separate from its type declaration (maybe many lines after the type declaration).

You can initialize a variable with one value and then, in an assignment statement, change the variable's value.

```
int year = 2008;
System.out.println(year);
System.out.println("Global financial crisis");
year = 2009;
System.out.println(year);
System.out.println("Obama elected US president");
year = 2010;
System.out.println(year);
System.out.println("Oil spill in the Gulf of Mexico");
```

Sometimes, you need a name for a value that doesn't change during the program's run. In such situations, the keyword `final` signals a variable whose value can't be reassigned.

```
final int NUMBER_OF_PLANETS = 9;
```

A `final` variable is a variable whose value doesn't vary. (As far as I know, no one's ever seriously suggested calling these things *invariables*.)

You can initialize a `final` variable's value, but after the initialization, you can't change the variable's value with an assignment statement. In other words, after you declare `final int NUMBER_OF_PLANETS = 9`, this assignment statement isn't legal:

```
NUMBER_OF_PLANETS = 8;
```

If Pluto is no longer a planet, you can't accommodate the change without changing the `9` in the `final int NUMBER_OF_PLANETS = 9` declaration.

In Java, the word `final` is one of Java's modifiers. A *modifier* is like an adjective in English. A modifier causes a slight change in the meaning of a declaration. For example, in this section, the word `final` modifies the `NUMBER_OF_PLANETS` declaration, making the value of `NUMBER_OF_PLANETS` unchangeable.

For more information about Java's modifiers, see Chapters 9 and 10.

As a rule, you use `final` variables to give friendly names to values that never (or rarely) change. For example, in a Java program, `6.626068e-34` stands for 6.626068×10^{-34}, which is the same as this:

```
0.00000000000000000000000000000006626068
```

In a quantum physics application, you probably don't want to retype the number `6.626068e-34` several times in your code. (You can type the number wrong even when you copy-and-paste.) To keep errors from creeping into your code, you declare

```
final double PLANCK_CONSTANT = 6.626068e-34;
```

From that point on, rather than typing 6.626068e-34 multiple times in your code, you can type only the name `PLANCK_CONSTANT` when needed.

You can use lowercase letters in any variable, including `final` variables. But Java programmers seldom write code this way. To keep from looking like a complete newbie, use only uppercase letters and digits in a `final` variable's name. Use underscores to separate words.

A loophole in the Java language specification allows you, under certain circumstances, to use an assignment statement to give a variable its initial value. For a variable, such as `amount`, declared inside of a method, you can write `final int amount;` on one line, and then `amount = 0;` on another line. Want my advice? Ignore this loophole. Don't even read this Technical Stuff icon!

Expressions and literals

In a computer program, an *expression* is a bunch of text that has a value. For example, in Listing 6-1, the number `74` and the words `anInteger` and `aCharacter` both have values. If I use the name `anInteger` in ten different places in my Java program, then I have ten expressions, and each expression has a value. If I decide to type `anInteger + 17` somewhere in my program, then `anInteger + 17` is an expression because `anInteger + 17` has a value. Listing 6-1 has a bunch of expressions other than the `74`, `anInteger` and `aCharacter` expressions, but I'll let you fish for all the expressions on your own.

A *literal* is a kind of expression whose value doesn't change from one Java program to another. For example, the expression `74` means "the numeric value 74" in every Java program. Likewise, the expression `'J'` means "the tenth uppercase letter in the Roman alphabet" in every Java program, and the word `true` means "the opposite of `false`" in every Java program. The expressions `true`, `74`, and `'J'` are literals. Similarly, the text `"An int variable"` in Listing 6-1 is a literal because, in any Java program, the text `"An int variable"` stands for the same three words.

In Java, single quotation marks stand for a character. You can change the second declaration in Listing 6-1 this way:

```
char aCharacter = 'J';
```

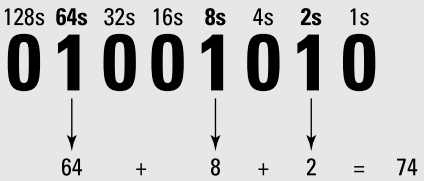
With this change, the program's run doesn't change. The dialog box shown in Figure 6-3 still contains the letter `J`.

In Java, a `char` value is a number in disguise. In Listing 6-1, you get the same result if the second type declaration is `char aCharacter = 'J'`. You can even do arithmetic with `char` values. For example, in Listing 6-1, if you change the second declaration to `char aCharacter = 'J' + 2`, you get the letter `L`.

The 01000001 01000010 01000011s

What does 01001010 have to do with the number 74 or with the letter J?

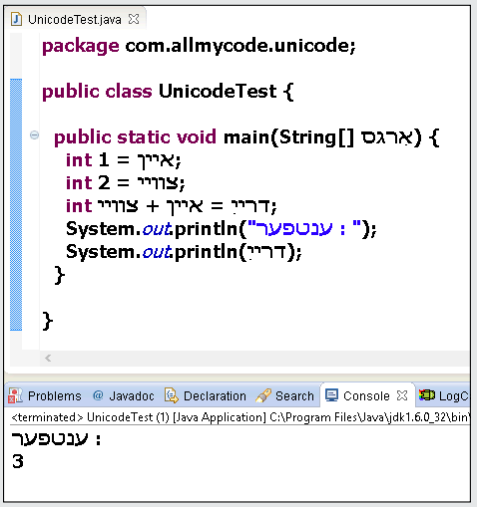
The answer for 74 involves the binary number representation. The familiar base-10 (decimal) system has a 1s column, a 10s column, a 100s column, a 1000s column, and so on. But the base-2 (binary) system has a 1s column, a 2s column, a 4s column, an 8s column, and so on. The figure shows how you get 74 from 01001010 using the binary column values.



The connection between 01001010 and the letter J might seem more arbitrary. In the early 1960s, a group of professionals devised the American Standard Code for Information Interchange (ASCII). In the ASCII representation, each character takes up 8 bits. You can see the representations for some of the characters in the sidebar table. For example, our friend 01001010 (which, as a binary number, stands for 74) is also the way the computer stores the letter J. The decision to make A be 01000001 and to make J be 01001010 has roots in the 20th century's typographic hardware. (The site www.wps.com/J/codes has some nice tidbits about all this.)

In the late 1980s, as modern communications led to increasing globalization, a group of experts began work on an enhanced code with up to 32 bits for each character. The lower eight Unicode bits have the same meanings as in the ASCII code, but with so many more bits, the Unicode standard has room for languages other than English. A Java char value is a 16-bit Unicode number, which means that, depending on the way you interpret it, a char is either a number between 0 and 65535 or a character in one of the many Unicode languages.

In fact, you can use non-English characters for identifiers in a Java program. In the figure, I use Eclipse to run a program with identifiers and output in Yiddish. The words in a few of the statements are out of order because I mix left-to-right and right-to-left languages. But otherwise, the stuff in the figure is a plain-old Java program!



Bits	*When Interpreted As an int*	*When Interpreted As a char*	*Bits*	*When Interpreted As an int*	*When Interpreted As a char*
00100000	32	space	00111111	63	?
00100001	33	!	01000000	64	@
00100010	34	"	01000001	65	A
00100011	35	#	01000010	66	B
00100100	36	$	01000011	67	C
00100101	37	%	.	.	.
00100110	38	&	.	.	.
00100111	39	'	*etc.*	*etc.*	*etc.*
00101000	40	(	01011000	88	X
00101001	41	)	01011001	89	Y
00101010	42	*	01011010	90	Z
00101011	43	+	01011011	91	[
00101100	44	,	01011100	92	\
00101101	45	-	01011101	93	]
00101110	46	.	01011110	94	^
00101111	47	/	01011111	95	_
00110000	48	0	01100000	96	`
00110001	49	1	01100001	97	a
00110010	50	2	01100010	98	b
00110011	51	3	01100011	99	c
00110100	52	4	.	.	.
00110101	53	5	.	.	.
00110110	54	6	*etc.*	*etc.*	*etc.*
00110111	55	7	01111000	120	x
00111000	56	8	01111001	121	y
00111001	57	9	01111010	122	z
00111010	58	:	01111011	123	{
00111011	59	;	01111100	124	|
00111100	60	<	01111101	125	}
00111101	61	=	01111110	126	~
00111110	62	>	01111111	127	delete

How to string characters together

In Java, a single character isn't the same as a string of characters. Compare the character `'J'` with the string `"An int variable"` in Listing 6-1. A *character* literal has single quotation marks; a *string* literal has double quotation marks.

In Java, a string of characters may contain more than one character, but a string of characters doesn't necessarily contain more than one character. (Surprise!) You can write

```
char aCharacter = 'J';
```

because a character literal has single quotation marks. And because `String` is one of Java's types, you can also write

```
String myFirstName = "Barry";
```

initializing the `String` variable `myFirstName` with the `String` literal `"Barry"`. Even though `"A"` contains only one letter, you can write

```
String myMiddleInitial = "A";
```

because `"A"`, with its double quotation marks, is a `String` literal.

But in Java, a single character isn't the same as a one-character string, so you can't write

```
//Don't do this:
char theLastLetter = "Z";
```

Even though it contains only one character, the expression `"Z"` is a `String` value, so you can't initialize a `char` variable with the expression `"Z"`.

Java's primitive types

Java has two kinds of types: primitive and reference. Primitive types are the atoms — the basic building blocks. In contrast, reference types are the things you create by combining primitive types (and by combining other reference types).

This chapter covers (almost exclusively) Java's primitive types. Chapter 9 introduces Java's reference types.

Throughout this chapter, I give some attention to Java's `String` type. The `String` type in reality belongs in Chapter 9 because Java's `String` type is a reference type, not a primitive type. But I can't wait until Chapter 9 to use strings of characters in my examples. So consider this chapter's `String` material to be an informal (but useful) preview of Java's `String` type.

Table 6-1 describes all eight primitive Java types.

Table 6-1 Java's Primitive Types

Type Name	*What a Literal Looks Like*	*Range of Values*
Integral types		
`byte`	`(byte)42`	–128 to 127
`short`	`(short)42`	–32768 to 32767
`int`	`42`	–2147483648 to 2147483647
`long`	`42L`	–9223372036854775808 to 9223372036854775807
Character type (which is, technically, an Integral type)		
`char`	`'A'`	Thousands of characters, glyphs, and symbols
Floating-point types		
`float`	`42.0F`	-3.4×10^{38} to 3.4×10^{38}
`double`	`42.0` or `0.314159e1`	-1.8×10^{308} to 1.8×10^{308}
Logical type		
`boolean`	`true`	true, false

You can divide Java's primitive types into three categories:

- **Integral**

  The *integral* types represent whole numbers — numbers with no digits to the right of the decimal point. For example, the number `42` in a Java program represents the `int` value 42, as in 42 cents or 42 clowns or 42 eggs. A family can't possibly have 2.5 children, so an `int` variable is a good place to store the number of kids in a particular family.

  The thing that distinguishes one integral type from another is the range of values you can represent with each type. For example, a variable of type `int` represents a number from –2147483648 to +2147483647.

When you need a number with no digits to the right of the decimal point, you can almost always use the `int` type. Java's `byte`, `short`, and `long` types are reserved for special range needs (and for finicky programmers).

- **Floating-point**

  The *floating-point* types represent numbers with digits to the right of the decimal point, even if those digits are all zeros. For example, an old wooden measuring stick might be 1.001 meters long, and a very precise measuring stick might be 1.000 meters long.

  The thing that distinguishes the two floating-point types (`double` and `float`) from one another is the range of values you can represent with the types. The `double` type has a much larger range and is much more accurate.

  In spite of their names, Java programmers almost always use `double` rather than `float`, and when you write an ordinary literal (such as `42.0`), that literal is a `double` value. (On the off chance that you want to create a `float` value, write `42.0F`.)

- **Logical**

  A `boolean` variable has one of two values: `true` or `false`. You can assign `74` to an `int` variable, and you can assign `true` (for example) to a `boolean` variable:

```
int numberOfPopsicles;
boolean areLemonFlavored;
numberOfPopsicles = 22;
areLemonFlavored = true;
```

  You can do arithmetic with numeric values, and you can do a kind of "arithmetic" with `boolean` values. For more information, see the next section.

Things You Can Do with Types

You can do arithmetic with Java's *operators*. The most commonly used arithmetic operators are `+` (addition), `–` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder upon division).

- **When you use an arithmetic operator to combine two `int` values, the result is another `int` value.**

  For example, the value of `4 + 15` is 19. The value of `14 / 5` is 2 (because 5 "goes into" 14 two times, and even though the remainder is bigger than ½, the remainder is omitted). The value of `14 % 5` is 4 (because 14 divided by 5 leaves a remainder of 4).

The same kinds of rules apply to the other integral types. For example, when you add a `long` value to a `long` value, you get another `long` value.

- **When you use an arithmetic operator to combine two `double` values, the result is another `double` value.**

  For example, the value of `4.0 + 15.0` is 19.0. The value of `14.0 / 5.0` is 2.8.

  The same kind of rule applies to `float` values. For example, a `float` value plus a `float` value is another `float` value.

- **When you use an arithmetic operator to combine an `int` value with a `double` value, the result is another `double` value.**

  Java *widens* the `int` value in order to combine it with the `double` value. For example, `4 + 15.0` is the same as `4.0 + 15.0`, which is 19.0. And `14 / 5.0` is the same as `14.0 / 5.0`, which is 2.8.

  This widening also happens when you combine two different kinds of integral values or two different kinds of floating-point values. For example, the number 9000000000000000000 is too large to be an `int` value, so

```
9000000000000000000L + 1
```

  is the same as

```
9000000000000000000L + 1L
```

  which is

```
9000000000000000001L
```

Two other popular operators are increment `++` and decrement `--`. The most common use of the increment and decrement operators looks like this:

```
x++;
y--;
```

But you can also place the operators before the variables:

```
++x;
--y;
```

Placing the operator after the variable is called *postincrementing* (or *postdecrementing*). Placing the operator before the variable is called *preincrementing* (or *predecrementing*).

Both forms (before and after the variable) have the same effect on the variable's value; namely, the increment `++` operator always adds 1 to the value, and the decrement `--` operator always subtracts 1 from the value. The only difference is what happens if you dare to display (or otherwise examine) the value of something like `x++`. Figure 6-4 illustrates this unsettling idea.

```
package come.allmycode.demos;

import javax.swing.JOptionPane;

public class IncrementTest {

  public static void main (String[] args) {
    int x = 10;
    JOptionPane.showMessageDialog (null, ++x);

    JOptionPane.showMessageDialog (null, x);

    JOptionPane.showMessageDialog (null, x++);

    JOptionPane.showMessageDialog (null, x);
  }
}
```

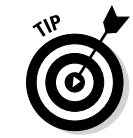
Displays 11 because the value of ++x is the same as the value of x+1

Displays 11 because ++x (in the previous statement) added to 1 to x

Displays 11 (SURPRISE!) because the value of x++ is the same as the value of x

Displays12 (SURPRISE!) because x++ (in the previous statement) added 1 to x

Figure 6-4: Preincrement and post-increment.



In practice, if you remember only that `x++` adds 1 to the value of `x`, you're usually okay.

The curious behavior shown in Figure 6-4 was inspired by assembly languages of the 1970s. These languages have instructions that perform increment and decrement operations on a processor's internal registers.

Add letters to numbers (Huh?)

You can add strings and char values to other elements and to each other. Listing 6-2 has some examples.

Listing 6-2: Java's Versatile Plus Sign

```
package com.allmycode.demos;

public class PlusSignTest {

  public static void main(String[] args) {
    int x = 74;
    System.out.println("Hello, " + "world!");
    System.out.println
      ("The value of x is " + x + ".");
    System.out.println
```

```
      ("The second letter of the alphabet is " +
                                        'B' + ".");
    System.out.println
      ("The fifth prime number is " + 11 + '.');
    System.out.println
      ("The sum of 18 and 21 is " + 18 + 21 +
                          ". Oops! That's wrong.");
    System.out.println
      ("The sum of 18 and 21 is " + (18 + 21) +
                               ". That's better.");

  }

}
```

The `String` type more appropriately belongs in Chapter 9 because Java's `String` type isn't a primitive type. Even so, I start covering the `String` type in this chapter.

When you run the code in Listing 6-2, you see the output shown in Figure 6-5.

Figure 6-5: A run of the code in Listing 6-2.

```
Hello, world!
The value of x is 74.
The second letter of the alphabet is B.
The fifth prime number is 11.
The sum of 18 and 21 is 1821. Oops! That's wrong.
The sum of 18 and 21 is 39. That's better.
```

Here's what's happening in Figure 6-5:

- **When you use the plus sign to combine two strings, it stands for string concatenation.**

  String concatenation is a fancy name for what happens when you display one string immediately after another. In Listing 6-2, the act of concatenating `"Hello, "` and `"world!"` yields the string

  ```
  "Hello, world!"
  ```

- **When you add a string to a number, Java turns the number into a string and concatenates the strings.**

  In Listing 6-2, the `x` variable is initialized to 74. The code displays `"The value of x is " + x` (a string plus an `int` variable). When adding the string `"The value of x is "` to the number 74, Java turns the `int` 74 into the string `"74"`. So `"The value of x is " + x` becomes `"The value of x is " + "74"`, which (after string concatenation) becomes `"The value of x is 74"`.

This automatic conversion of a number into a string is handy whenever you want to display a brief explanation along with a numeric value.

The computer's internal representation of the number 74 is 0000000000 00000000000000001001010 (with 1 in the 64s place, 1 in the 8s place, and 1 in the 2s place). In contrast, the computer's internal representation of the string `"74"` is 00000000001101110000000000110100. (For some clues to help you understand why these bits represent the `"74"` string, see the table accompanying this chapter's earlier sidebar "The 01000001 01000010 01000011s.") The bottom line, as far as Java is concerned, is that the number 74 and the string `"74"` aren't the same.

- **When you add a string to any other kind of value, Java turns the other value into a string and concatenates the strings.**

  The third `System.out.println` call in Listing 6-2 adds the `char` value `'B'` to a string. The result, as you can see in Figure 6-5, is a string containing the letter `B`.

- **The order in which the computer performs operations can affect the outcome.**

  The last two `System.out.println` calls in Listing 6-2 illustrate this point. In the next-to-last call, the computer works from left to right. The computer starts by combining `"The sum of 18 and 21 is "` with `18`, getting `"The sum of 18 and 21 is 18"`. Then, working its way rightward, the computer combines `"The sum of 18 and 21 is 18"` with `21` getting the screwy string `"The sum of 18 and 21 is 1821"`.

  In the last `System.out.println` call, I fix these problems by grouping `18` and `21` in parentheses. As a result, the computer starts by adding 18 and 21 to get 39. Then the computer combines `"The sum of 18 and 21 is "` with `39`, getting the more sensible string `"The sum of 18 and 21 is 39"`.

Java's exotic assignment operators

In a Java program, you can add 2 to a variable with a statement like this:

```
numberOfCows = numberOfCows + 2;
```

But to a seasoned Java developer, a statement of this kind is horribly *gauche*. You might as well wear white after Labor Day or talk seriously about a "nucular" reactor. Why?

Because Java has a fancy *compound assignment operator* that performs the same task in a more concise way. The statement

```
numberOfCows += 2;
```

adds 2 to `numberOfCows` and lets you easily recognize the programmer's intention. For a silly example, imagine having several similarly named variables in the same program:

```
int numberOfCows;
int numberOfCrows;
int numberOfCries;
int numberOfCrays;
int numberOfGrays;
```

Then the statement

```
numberOfCrows += 2;
```

doesn't force you to check both sides of an assignment. Instead, the `+=` operator makes the statement's intent crystal-clear.

Java's other compound assignment operators include `-=`, `*=`, `/=`, `%=`, and others. For example, to multiply `numberOfCows` by `numberOfDays`, you can write

```
numberOfCows *= numberOfDays;
```

A compound assignment, like `numberOfCrows += 2`, might take a tiny bit less time to execute than the cruder `numberOfCows = numberOfCows + 2`. But the main reason for using a compound assignment statement is to make the program easier for other developers to read and understand. The savings in computing time, if any, is usually minimal.

True bit

A `boolean` value is either `true` or `false`. Those are only two possible values, compared with the thousands of values an `int` variable can have. But these two values are quite powerful. (When someone says "You've won the lottery" or "Your shoe is untied," you probably care whether these statements are true or false. Don't you?)

When you compare things with one another, the result is a `boolean` value. For example, the statement

```
System.out.println(3 > 2);
```

puts the word `true` in Eclipse's Console view. In addition to Java's > (greater than) operator, you can compare values with < (less than), >= (greater than or equal), and <= (less than or equal).

You can also use a double-equal sign (==) to find out whether two values are equal to one another. The statement

```
System.out.println(15 == 9 + 9);
```

puts the word `false` in the Console view. You can also test for inequality. For example, the statement

```
System.out.println(15 != 9 + 9);
```

```
System.out.println(15 != 9 + 9);
```

puts the word `true` in the Console view. (A computer keyboard has no ≠ sign. To help you remember the != operator, think of the exclamation point as a work-around for making a slash through the equal sign.)

An expression whose value is either `true` or `false` is a *condition*. In this section, expressions such as `3 > 2` and `15 != 9 + 9` are examples of conditions.

The symbol to compare for equality isn't the same as the symbol that's used in an assignment or an initialization. Assignment or initialization uses a single equal sign (=), and comparison for equality uses a double equal sign (==). Everybody mistakenly uses the single equal sign to compare for equality several times in their programming careers. The trick is not to avoid making the mistake; the trick is to catch the mistake whenever you make it.

It's nice to display the word `true` or `false` in Eclipse's Console view, but `boolean` values aren't just for pretty displays. To find out how `boolean` values can control the sequence of steps in your program, see Chapter 8.

Java isn't like a game of horseshoes

Even when you correctly use the double equal sign, you have to be careful. Figure 6-6 shows you what happens in a paper-and-pencil calculation to convert 21 degrees Celsius to Fahrenheit. You get exactly 69.8.

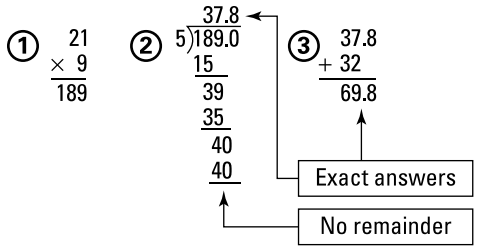


Figure 6-6: An exact Celsius-to-Fahrenheit conversion.

But when you add the following statement to a Java program, you see `false`, not `true`:

```
System.out.println(9.0 / 5.0 * 21 + 32.0 == 69.8);
```

Why isn't `9.0 / 5.0 * 21 + 32.0` the same as `69.8`? The answer is that Java's arithmetic operators don't use the decimal system — they use the *binary* system. And in binary arithmetic, things don't go as well as they do in Figure 6-6.

Figure 6-7 shows you how the computer divides 189.0 by 5. You might not understand (and you might not want to understand) how the computer computes the value 100101.110011001100110011 . . ., but when you stop after 64 bits or so, this answer isn't exactly 37.8. It's more like 37.800000000000004, which is slightly inaccurate. In a Java program, when you ask whether `9.0 / 5.0 * 21 + 32.0` is exactly equal to `69.8`, the computer says "No, that's false."

Avoid comparing `double` values or `float` values for equality (using `==`) or for inequality (using `!=`). Comparing strings for equality (as in the expression `"passw0rd" == "passw0rd"`) is also unadvisable.

For details about comparing strings, see Chapter 8.

```
      100101.110011001100110011 ... etc.
101)10111101.000000000
    101
     0111
     0101
      1001
      0101
       100 0
       010 1
         1 10
         1 01
           1000
           0101
             110
             101
               1000
```

Figure 6-7: A division problem that never ends.

Use Java's logical operators

Real-life situations might involve long chains of conditions. Here's an example I found in a letter from the U.S. Department of Education federal student loans department:

> *Interest starts to accrue daily prior to repayment on all unsubsidized loans beginning on the first disbursement date and on all unsubsidized loans first disbursed on or after July 1, 2012 and before July 1, 2014 at the beginning of the grace period*. . . .*
>
> **Grace Period — A 6-month period before the first payment on a subsidized or unsubsidized Stafford Loan is due. The grace period begins the day after the student graduates, leaves school, or drops below half-time status and ends the day before the repayment period begins.*

Whew! I'm glad I didn't miss any of the fine print!

The good news is that an app's conditions can be expressed using Java's `&&`, `||` and `!` operators. The story begins in Listing 6-3. Here, the listing's code computes the price for a movie theater ticket.

Listing 6-3: Pay the Regular Ticket Price?

```
package com.allmycode.tickets;

import javax.swing.JOptionPane;

public class Regular {

  public static void main(String[] args) {
    String ageString;
    int age;
    boolean chargeRegularPrice;

    ageString = JOptionPane.showInputDialog("Age?");
    age = Integer.parseInt(ageString);
    chargeRegularPrice = 18 <= age && age < 65;
    JOptionPane.showMessageDialog(null,
        chargeRegularPrice, "Regular price?",
        JOptionPane.INFORMATION_MESSAGE);
  }

}
```

Figure 6-8 shows a run of the code in Listing 6-3 with the value of `age` set to 17; Figure 6-9 shows a run with `age` set to 18.

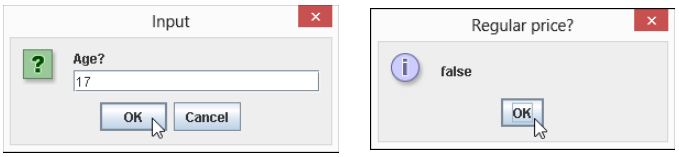


Figure 6-8: A youngster goes to the movies.

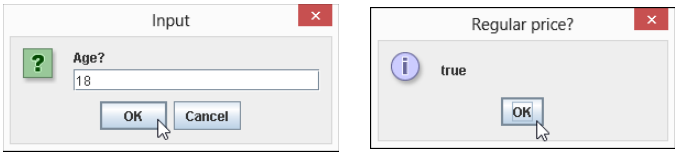


Figure 6-9: If you can drink alcohol in Moldova, you can pay full price at our theater!

Figures 6-8 and 6-9 might look peculiar because I've chosen to display the words `true` and `false` instead of more user-friendly messages (such as `Charge this bum the regular price!`). I do better when I cover Java's `if` statements in Chapter 8.

In Listing 6-3, the value of `chargeRegularPrice` is `true` or `false` depending on the outcome of the `18 <= age && age < 65` condition test. The `&&` operator stands for a logical *and* combination, so `18 <= age && age < 65` is `true` as long as `age` is greater than or equal to 18 *and* `age` is less than 65.

To create a condition like `18 <= age && age < 65`, you have to use the `age` variable twice. You can't write `18 <= age < 65`. Other people might understand what `18 <= age < 65` means, but Java doesn't understand it.

In the earlier section "Java isn't like a game of horseshoes," I warn against using the `==` operator to compare two `double` values with one another. If you absolutely must compare `double` values with one another, give yourself a little leeway. Rather than writing `fahrTemp == 69.8`, write something like this:

```
(69.7779 < fahrTemp) && (fahrTemp < 69.8001)
```

Listing 6-3 has two other interesting new features. One feature is the use of `JOptionPane.showInputDialog`. This method displays a dialog box like the first box shown earlier, in Figure 6-8 (and the first box shown in Figure 6-9). The box has its own text field for the user's input. Normally, the user types

something in the text field and then presses OK. Whatever the user types in the text field becomes the value of the call to `JOptionPane.showInputDialog`, as shown in Figure 6-10.

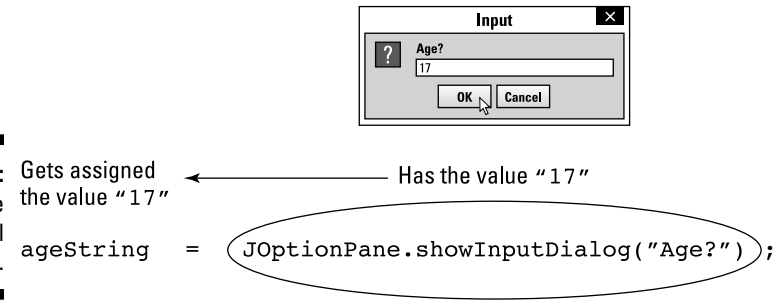


Figure 6-10: An entire method call has a value.

In Figure 6-10, notice that the entire method call `JOptionPane.showInputDialog("Age?")` becomes synonymous with the string `"17"` (or with whatever the user types in the text field in the dialog box). So the statement

```
ageString = JOptionPane.showInputDialog("Age?");
```

effectively becomes the following statement:

```
ageString = "17";
```

The `showInputDialog` method always returns a string of characters, so in Listing 6-3, it's important that I declare `appString` to be of type `String`. The problem is that a string of characters isn't the same as a number. You can't use the < operator to compare `"17"` with `"18"`. Java doesn't do arithmetic on strings of characters, even when those strings happen to look like numbers.

Before comparing the user's input with the numbers 18 and 65, you have to turn the user's input into a number. (You have to turn a string like `"17"` into an `int` value like 17.) To do that, you call Java's `Integer.parseInt` method:

- The `Integer.parseInt` method's parameter is a `String` value.
- The value of a call to the `Integer.parseInt` is an `int` value.

So, in Listing 6-3, the statement

```
age = Integer.parseInt(ageString);
```

assigns an `int` value to the variable `age`. That's good because, in the listing, `age` is declared to be of type `int`.

Listing 6-4 illustrates Java's || operator. (In case you're not sure, you type the || operator by pressing the | key twice.) The || operator stands for a logical *or* combination, so `age < 18 || 65 <= age` is `true` as long as `age` is less than 18 *or* `age` is greater than or equal to 65.

Listing 6-4: Pay the Discounted Ticket Price?

```
package com.allmycode.tickets;

import javax.swing.JOptionPane;

public class Discount {

  public static void main(String[] args) {
    String ageString;
    int age;
    boolean chargeDiscountPrice;

    ageString = JOptionPane.showInputDialog("Age?");
    age = Integer.parseInt(ageString);
    chargeDiscountPrice = age < 18 || 65 <= age;
    JOptionPane.showMessageDialog(null,
        chargeDiscountPrice, "Discount price?",
        JOptionPane.INFORMATION_MESSAGE);
  }

}
```

Runs of the code from Listing 6-4 are shown in Figures 6-11 and 6-12.

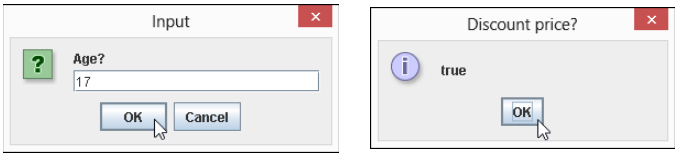


Figure 6-11: Ah, to be young again!

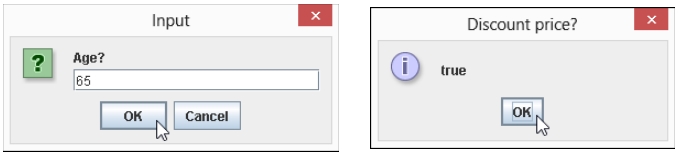


Figure 6-12: Ah, to be old at last!

Listing 6-5 adds Java's ! operator to the logical stew. If you're unfamiliar with languages like Java, you have to stop thinking that the exclamation point means, "Yes, definitely." Instead, Java's ! operator means *not*. In Listing 6-5, with `isSpecialShowing` being `true` or `false`, the expression

`!isSpecialShowing` stands for the opposite of `isSpecialShowing`. That is, when `isSpecialShowing` is `true`, `!isSpecialShowing` is `false`. And when `isSpecialShowing` is `false`, `!isSpecialShowing` is `true`.

Listing 6-5: What about Special Showings?

```
package com.allmycode.tickets;

import javax.swing.JOptionPane;

public class Discount2 {

  public static void main(String[] args) {
    String ageString;
    int age;
    boolean chargeDiscountPrice;
    String specialShowingString;
    boolean isSpecialShowing;

    ageString = JOptionPane.showInputDialog("Age?");
    age = Integer.parseInt(ageString);

    specialShowingString = JOptionPane.showInputDialog
        ("Special showing (true/false)?");
    isSpecialShowing =
        Boolean.parseBoolean(specialShowingString);
    chargeDiscountPrice =
        (age < 18 || 65 <= age) && !isSpecialShowing;

    JOptionPane.showMessageDialog(null,
        chargeDiscountPrice, "Discount price?",
        JOptionPane.INFORMATION_MESSAGE);
  }

}
```

Runs of the code from Listing 6-5 are shown in Figures 6-13 and 6-14.

The primary condition in Listing 6-5 grants the discount price to kids and to seniors as long as the current feature isn't a "special showing" — one that the management considers to be a hot item, such as the first week of the run of a highly anticipated movie. When there's a special showing, no one gets the discounted price.

In Figures 6-13 and 6-14, I artificially force the user to type the word `true` or the word `false` (without quotation marks) in an input text field. Figure 6-15 shows how the user's response becomes a string of characters that's deposited into my `specialShowingString` variable.

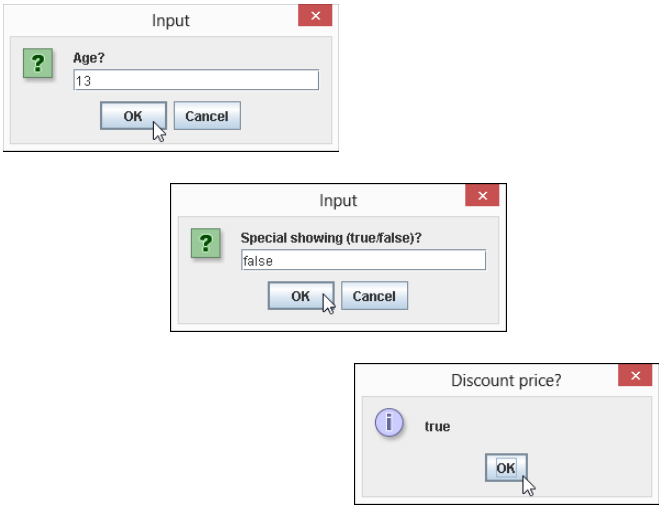


Figure 6-13: A special price for a not-so-special showing.

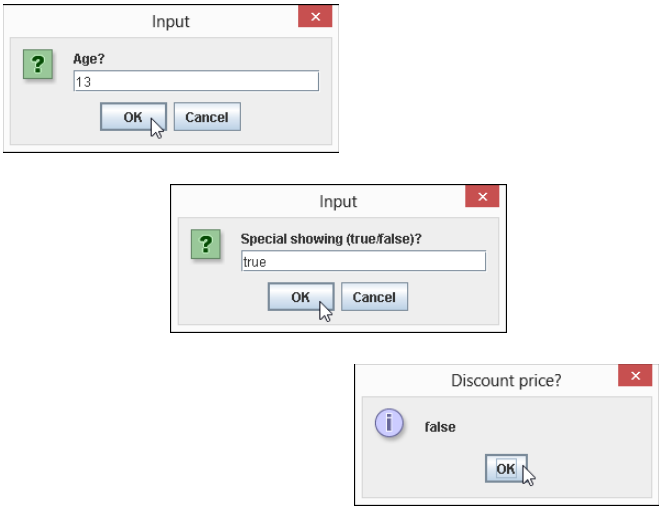


Figure 6-14: A special showing with a not-so-special price.

In the next statement in Listing 6-5, the method `Boolean.parseBoolean` does for `boolean` values what `Integer.parseInt` does for `int` values. The `Boolean.parseBoolean` method turns the value of `specialShowing` `String` (the string `"true"` or `"false"`) into an honest-to-goodness `boolean` value. To this `boolean` value, the computer can apply the `!` operator and, if needed, the `&&` and `||` operators.

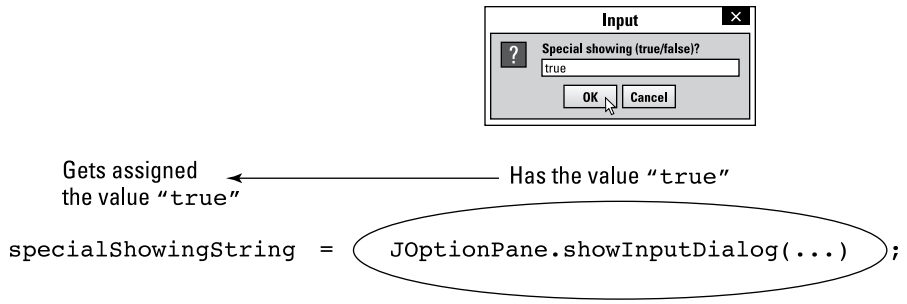


Figure 6-15: Getting the word *true* from the user's input.

For any condition you want to express, you always have several ways to express it. For example, rather than test `numberOfCats != 3`, you can be more long-winded and test `!(numberOfCats == 3)`. Rather than test `myAge < yourAge`, you can get the same answer by testing `yourAge > myAge` or `!(myAge >= yourAge)`. Rather than type `a != b && c != d`, you can get the same result with `!(a == b || c == d)`. (A guy named Augustus DeMorgan told me about this last trick.)

Parenthetically speaking . . .

The big condition in Listing 6-5 (the condition `(age < 18 || 65 <= age) && !isSpecialShowing`) illustrates the need for (and the importance of) parentheses (but only when parentheses are needed (or when they help people understand your code)).

When you don't use parentheses, Java's *precedence rules* settle arguments about the meaning of the expression. They tell you whether the line

```
age < 18 || 65 <= age && !isSpecialShowing
```

stands for the expression

```
(age < 18 || 65 <= age) && !isSpecialShowing
```

or for this one:

```
age < 18 || (65 <= age && !isSpecialShowing)
```

According to the precedence rules, in the absence of parentheses, the computer evaluates `&&` before evaluating `||`. If you omit the parentheses, the computer first checks to find out whether `65 <= age && !isSpecialShowing`. Then the computer combines the result with a test of the `age < 18` condition. Imagine a 16-year-old kid buying a movie ticket on the day of a special showing. The condition `65 <= age && !isSpecialShowing` is

`false`, but the condition `age < 18` is `true`. Because one of the two conditions on either side of the `||` operator is `true`, the whole nonparenthesized condition is `true` — and, to the theater management's dismay, the 16-year-old kid gets a discount ticket.

Sometimes, you can take advantage of Java's precedence rules and omit the parentheses in an expression. But I have a problem: I don't like memorizing precedence rules, and when I visit Java's online language specifications document (`docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html`), I don't like figuring out how the rules apply to a particular condition.

When I create an expression like the one in Listing 6-5, I almost always use parentheses. In general, I use parentheses if I have any doubt about the way the computer behaves without them. I also add parentheses when doing so makes the code easier to read.

Sometimes, if I'm not sure about stuff and I'm in a curious frame of mind, I write a quick Java program to test the precedence rules. For example, I run Listing 6-5 with and without the condition's parentheses. I send a 16-year-old kid to the movie theater when there's a special showing and see whether the kid ever gets a discount ticket. This little experiment shows me that the parentheses aren't optional.