# Chapter 4: Making Decisions

## Making Decisions

One of my favorite quotes about decision making with computers is from two well-known computer scientists, John Hennessy and David Patterson. Dr. Hennessy is currently president of Stanford University, and Dr. Patterson is a professor and past chair of the computer science department at the University of California, Berkeley. In a textbook that they wrote together on computer architecture, they said, "What distinguishes a computer from a simple calculator is its ability to make decisions."

When you think about it, a simple calculator can do just about everything a computer can do: input data, output results, do calculations, even display graphs. But what it can't do is make decisions based on the data (unless it's one of the newer programmable calculators, of course).

In Java, we base decisions on the results of logical expressions, which we just looked at. Decisions in Java take the form of *if . . . else . . .* statements, where the ". . ." in each case is the logic we want to perform based on whether the condition we test is true or false. The complete format of the *if* statement is:

```
if (logical expression) {
    // statement(s) to run if the condition is true
}
else {
    // statement(s) to run if the condition is false
}
```

You don't have to include all of that, though. You can leave the whole *else* section out if you don't want to do anything when the condition is false. You can also leave the curly brackets out if you only have one statement to execute in either case. You only have to use them for multiple lines of code. But programmers normally include them anyway, just for clarity and to make it easier to add more code later.

## Repeating Actions With Loops

Java also gives us some convenient formats for repeating actions in *loops*. We talked about two of them in the first Java course, the *while* and *do-while* loops. Their format also includes a logical expression, which Java uses each time through the loop to decide whether to continue. So the equivalent of an *if* statement is hidden in each loop.

This is the *while* loop's format:

```
while (logical expression) {
    // statements to repeat while the condition is true
}
```

This type of loop evaluates the logical expression *before* beginning the loop, and if the condition is true, the loop runs its internal statements. If the expression is false the first time Java evaluates it, Java may never run the loop's statements.

This is the *do-while* loop's format:

```
do {
    // statements to repeat while the condition is true
} while (logical expression);
```

This type of loop evaluates the logical expression *after* finishing each iteration of the loop, and if the condition is true, it goes back and runs the loop again. That means this loop will *always* execute its statements at least once before testing its condition the first time.

Java has other types of loops that are more convenient to use in certain circumstances, and we'll learn about them in this course.

**Arrays**

Arrays are Java's way of storing a group of similar things in one name. You could, for example, build an array of integers, an array of strings, or an array of Booleans to use in your process. It's the same concept, regardless of the type of data you store in the array. The two rules that govern arrays are (1) all elements of an array must be the same type, and (2) once you declare an array, its size is fixed and you can't change it. Java uses square brackets ([]) to designate and use arrays. Here are some array declarations and instantiations to look at:

```
int[] x;
int[] y = new int[5];
String[] s;
float[] f = new float[1000];
x = new int[10000];
s = new String[99];
```

The first four lines declare arrays of different types; the second and fourth lines also create, or *instantiate*, the arrays. The fifth and sixth lines create arrays for the other two declarations.

Using an array's name by itself, like *x* or *s* above, refers to the entire array. Following the name with a number in brackets indicates a single element of the array. The references *x[22]* and *s[0]*, for example, refer to a single integer and string, respectively.

And as the last example reinforces, Java uses *zero-based* arrays. That means that the reference *s[0]* gets the first string in the s array, and *x[22]* gets the 23rd integer in the x array.

Java doesn't do well with invalid array indexes. If I use a negative number or a value that's too large as an index, my program will blow up in my face (so to speak). One of the most common array errors is to try to refer to the last element of an array using its size. For example, if I used *s[99]* in a program after creating the string array above, my program would crash because 99 is the size of the array, so its largest valid index is 98.

We'll talk more about arrays in this course, too, so don't worry if the concept is a bit fuzzy right now.

**Catching Errors (Exceptions)**

When Java encounters an error during runtime that it can't handle, it *throws an exception*. That's Java terminology for blowing up your program. Unless your code *catches* the exception and processes it, your program won't work.

The general form for exception handling is to put the code that might throw the exception into a *try block* and put the code to process the exception into a *catch block*. For example, a common error happens when the program reads something other than a number from an input source when the program expects a number. If that occurs in a Scanner class method, an InputMismatchException gets generated.

Here's a simple try-catch mechanism to take care of that error:

```
Scanner in = new Scanner(System.in)
try {
    int num = in.nextInt();
}
catch (InputMismatchException e) {
    System.out.println("Invalid input – not a number");
}
```

**Enumerations**

Java allows us to define data types, including the values that the type can hold (its *domain*), by using an *enumeration*. An enumeration is a way to define both a data type and all the possible values it can hold. Its name is often shortened to *enum*, which is the keyword used to define it.

A common use of enumerations is for types that have small domains, and a common example is the days of the week. I can set up an enumeration like this:

```
public enum Day {
     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
     THURSDAY, FRIDAY, SATURDAY
}
```

Once an enumeration is declared, I can use it and its values like this:

```
Day dayOfTheWeek;
   . . .
dayOfTheWeek = Day.MONDAY;
   . . .
if (dayOfTheWeek == Day.SUNDAY)
   . . .
```

You can find more about how to use enumerations using the link to Oracle's tutorial in the Supplementary Material at the end of the lesson.

### Switch

The last review topic I'll go over here is Java's switch statement, which can replace a number of if-else statements with a single control structure if the data being tested is one of Java's primitive types, a String, or an enumeration. Here's a short example showing how we could test an integer variable named month and set up a String with an appropriate month name:

```
switch (month) {
    case 1:  monthString = "January";
             break;
    case 2:  monthString = "February";
             break;
    case 3:  monthString = "March";
             break;
    case 4:  monthString = "April";
             break;
    case 5:  monthString = "May";
             break;
    case 6:  monthString = "June";
             break;
    case 7:  monthString = "July";
             break;
    case 8:  monthString = "August";
             break;
    case 9:  monthString = "September";
             break;
    case 10: monthString = "October";
             break;
    case 11: monthString = "November";
             break;
    case 12: monthString = "December";
             break;
    default: monthString = "Invalid month";
             break;
}
```

I think I've probably done enough of a review for you by now. There's just one short item in Chapter 5 before we quit.

© 2022 Cengage Learning, Inc. All Rights Reserved