Java Basics Unit

Lesson 5: Boolean Expressions and Controlling Program Flow

Lesson 5: Boolean Expressions and Controlling Program Flow

Overview

In this lesson, we're going to take a look at how we can control the flow of a program: how we can make decisions and repeat ourselves. You'll be introduced to conditional statements, loops, and boolean expressions.

What Can Our Code Do?

So far, our programs have simply executed from beginning to end, one statement after another. We've added some code that allows us to get input from the user, but even with that, our code executes in a straight line — the same way every time. There's got be more, right? Right! There is more but, surprisingly, not too much more… here is what we can do in code:

1. Execute in a straight line
2. Conditionally choose one path or another, based on some criteria
3. Repeat a set of statements a certain number of times, based on some criteria

That's it — there is no more. All programs are built from these options.

Boolean Expressions

How do we make these decisions? What do the criteria mentioned above consist of? At the heart of these criteria are **boolean expressions, conditional operators,** and **relational operators**.

Boolean expressions are similar to mathematical expressions in that they are code statements that evaluate to data values. The main difference is that boolean expressions always evaluate to either **true** or **false**. Another big difference is that boolean expressions are formed using boolean and relational operators instead of mathematical operators.

Relational operators operate, for the most part, on numerical operands but evaluate to boolean values:

Relational Operator	Meaning
Equal operand 1 == operand2	Evaluates to true if the two operands are equal, false otherwise
Not Equal operand1 != operand2	Evaluates to true if the two operands are not equal, false otherwise

Relational Operator	Meaning
Less Than operand1 < operand2	Evaluates to true if operand1 is less than operand2, false otherwise
Greater Than operand1 > operand2	Evaluates to true if operand1 is greater than operand2, false otherwise
Less Than or Equal operand1 <= operand2	Evaluates to true if operand1 is less than or equal to operand2, false otherwise
Greater Than or Equal operand1 >= operand2	Evaluates to true if operand1 is greater than or equal to operand2, false otherwise

Boolean operators operate on boolean operands and evaluate to boolean values:

Conditional Operator	Meaning
Negation !operand	Evaluates to true if the operand’s value is false, false otherwise
AND operand1 && operand2	Evaluates to true if both operand1 and operand2 are true, false otherwise
OR operand1 || operand2	Evaluates to true if either operand1 or operand2 or both are true, false otherwise
Exclusive OR (XOR) operand1^operand2	Evaluates to true if either operand1 or operand2 but not both are true, false otherwise

Conditional binary operator evaluation rules can be summed up in a **truth table**:

a	b	a && b	a || b	a^b
F	F	F	F	F
T	F	F	T	T
F	T	F	T	T
T	T	T	T	F

Note that AND (&&) and OR (||) are called **short circuit** operators because they will only evaluate the second operand if they have to. The & and | operators will always evaluate both operands.

Conditional Execution

Conditional execution allows us to make decisions in our code — it allows us to choose one path over another.

If Statement

The first construct we'll look at is the **if statement** (also known as the if-then statement). The if statement allows us to specify that a particular block of code should be run only if a certain condition is true. The basic form is:

```
if (condition) {
    // execute code if condition is true
}
```

A slightly more complex form is the **if-else statement** (also known as the if-then-else statement). This form tells our program to execute a certain block of code if a certain condition is true and to execute another block if the condition is false. It looks like this:

```
if (condition) {
      // execute code if condition is true
} else {
      // execute code if condition is false
}
```

The if-else statement can also be chained together so that you can check multiple conditions in your code as in the following code:

```
if (condition1) {
      // execute code if condition1 is true
} else if (condition2) {
      // execute code if condition2 is true
} else if (condition3) {
      // execute code if condition3 is true
} else {
      // execute code if all conditions above were false
}
```

Switch Statement

The **switch statement** is an alternative construct that allows for conditional execution. The switch statement checks a condition against any number of cases. If a case matches, then the code associated with that case is executed. The basic form is:

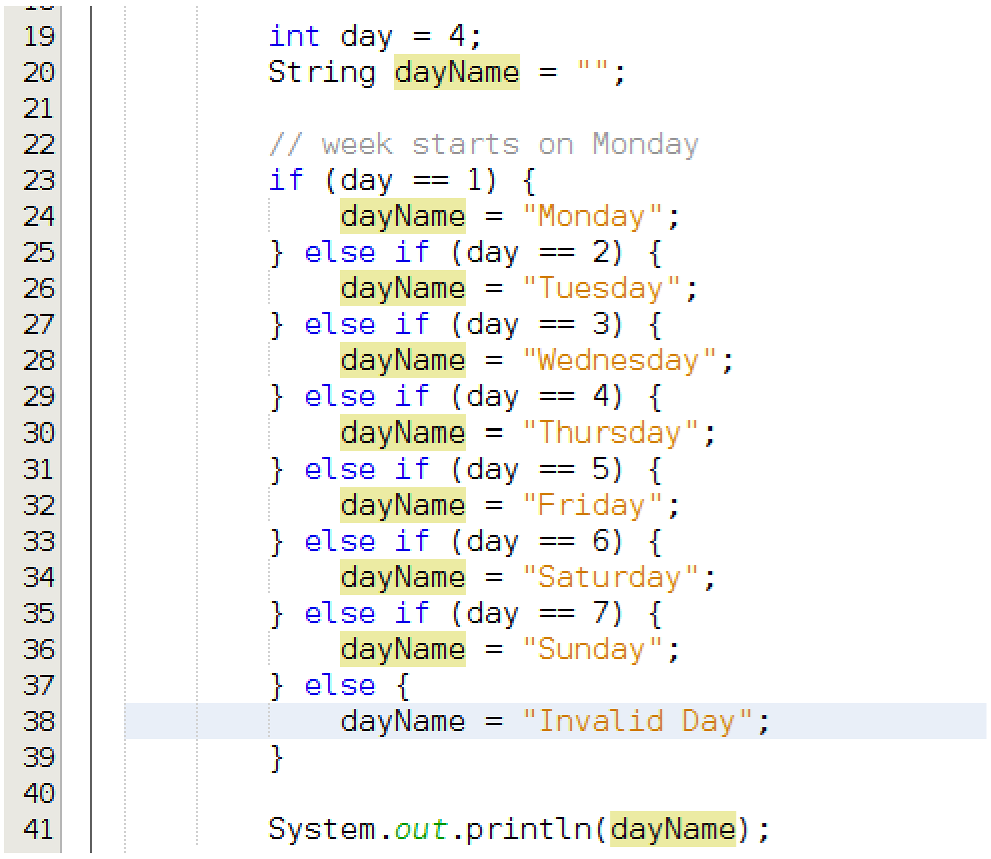
```
switch (expression) {
      case constant:
            // execute code if expression == constant
            break;
      case constant2:
            //execute code if expression == constant2
            break;
      default:
            //execute code if no match found
            break;
}
```

The main body of a switch statement is called the **switch block**. The switch block is comprised of one or more **case** or **default** labels. The code associated with the default label is executed if none of the cases match the expression.

A big difference between switch statements and if statements is that the conditions tested in an if statement must be boolean expressions, whereas the expressions evaluated in a switch statement can be byte, short, char, int, enumerated types (more on these later), Strings, or the wrapper classes for byte, short char, and int (more on this later). The switch statement simply checks if the evaluated expression matches any of the cases.

Another thing to notice about switch statements is the **break** statement. The break statement terminates the switch — in other words, after a break statement is executed, the program continues with the first statement after the switch block. The break statements are necessary because all statements after a matching case label are executed in order until either a break statement or the end of the switch block is encountered, regardless of the values of the subsequent case labels. This is known as **falling through**. We will see how we can take advantage of this feature in the demo code below.

To demonstrate conditional execution, we will look at some code that tells us the name of the day of the week given a number between 1 and 7 inclusive. First, we'll see how this is done using an if statement, then we'll see how this done using a switch. Finally, we'll see how we can use a switch statement (and the fall through feature) to tell us whether a given day is a weekday or not.

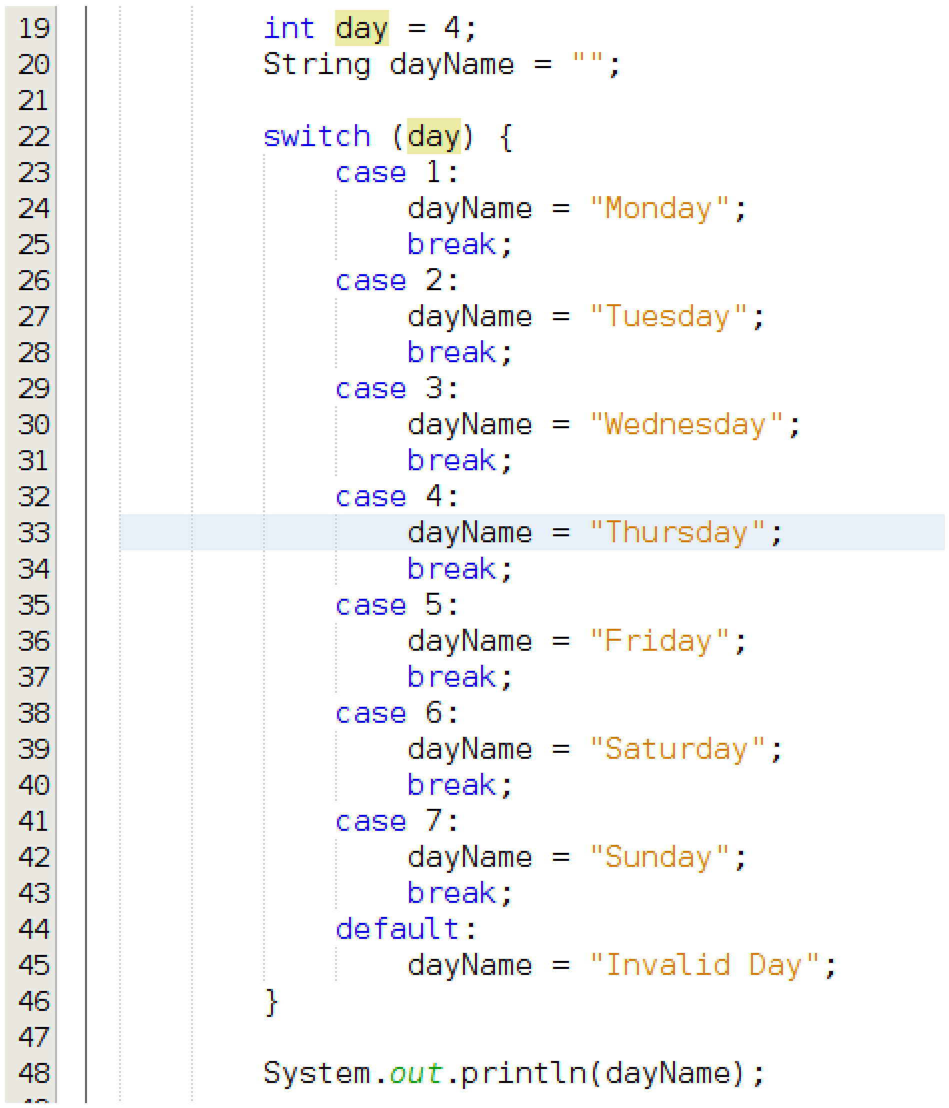


```java
int day = 4;
String dayName = "";

// week starts on Monday
if (day == 1) {
    dayName = "Monday";
} else if (day == 2) {
    dayName = "Tuesday";
} else if (day == 3) {
    dayName = "Wednesday";
} else if (day == 4) {
    dayName = "Thursday";
} else if (day == 5) {
    dayName = "Friday";
} else if (day == 6) {
    dayName = "Saturday";
} else if (day == 7) {
    dayName = "Sunday";
} else {
    dayName = "Invalid Day";
}

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

Figure 1: Day of Week with If Statement

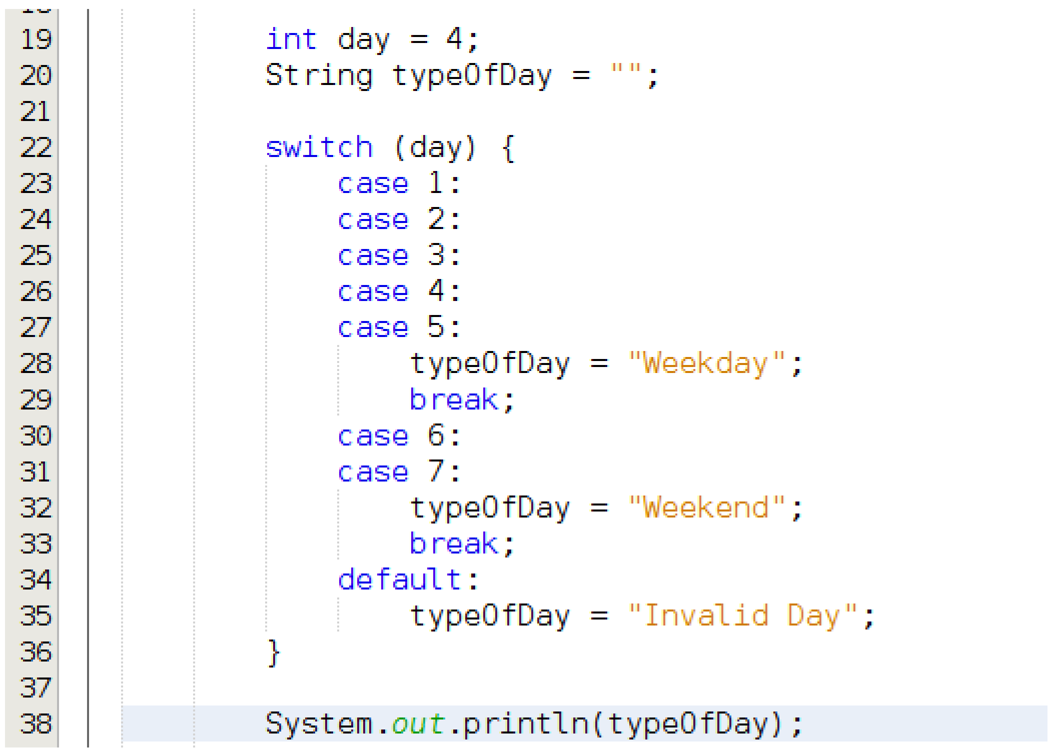


```java
int day = 4;
String dayName = "";

switch (day) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
        break;
    case 4:
        dayName = "Thursday";
        break;
    case 5:
        dayName = "Friday";
        break;
    case 6:
        dayName = "Saturday";
        break;
    case 7:
        dayName = "Sunday";
        break;
    default:
        dayName = "Invalid Day";
}

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

Figure 2: Day of Week with Switch Statement



```
int day = 4;
String typeOfDay = "";

switch (day) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        typeOfDay = "Weekday";
        break;
    case 6:
    case 7:
        typeOfDay = "Weekend";
        break;
    default:
        typeOfDay = "Invalid Day";
}

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

Figure 3: Type of Day with Switch Statement

Loops

There are three looping constructs in Java: do/while, while, and for loops. These constructs allow us to repeat execution of a code block as long as some condition is true.

Do/While Loop

The do/while loop will continue executing the statements contained in its block as long as the given condition is true. Because the condition check is at the bottom of the do/while loop, **the code in the do/while block is guaranteed to be run at least once**.

```
do {
      // code I want to repeat while condition is true
} while (condition);
```

While Loop

The while loop is similar to the do/while loop in that it will continue executing the statements contained in its block as long as the given condition is true. The condition check for the while loop is at the top of the loop which means that **the code in the while loop is not guaranteed to run**.

```
while (condition) {
      // code I want to repeat while condition is true
}
```

For Loop

The for statement (often referred to as the for loop) allows you to iterate over a range of values. The basic form is:

```
for (initialization; termination; increment) {
      // code I want to repeat a given number of times
}
```

The **initialization** statement is run only once as the loop execution begins.

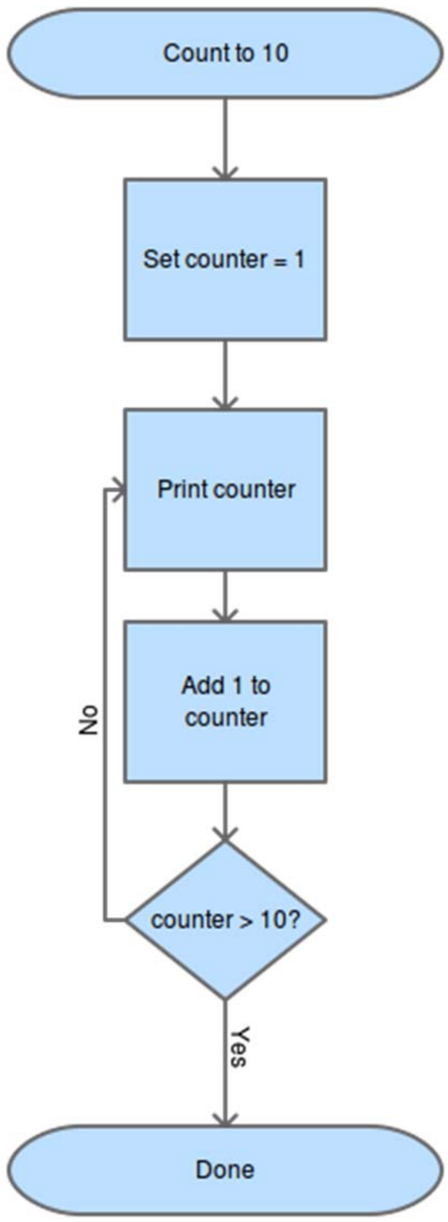
The **termination** expression is evaluated at the beginning of each loop; when it evaluates as false, the loop terminates.

The **increment** expression is evaluated after each loop iteration; you can increment or decrement values in this expression.

For loops are very useful in iterating over arrays or Collections of items — we'll explore that more when we cover these data structures.

Flowcharts

We now have a richer vocabulary, which means that we can create more complicated programs. Our programs were pretty simple when all we could do was execute on one statement after another in order. Now that we can make decisions and repeat ourselves, we need a way to describe our programs. We'll use simple flowcharts to describe our solutions. At this point, we only need the start/termination, process, and decision symbols as in this example:



Sidebar: Constants, Strings, Bitwise Operations, and Random Numbers

Here are quick notes on some subjects that will help you out with some of the assignments or that might come up in interviews.

Constants

On occasion, you will need to define constants, or "magic" numbers, in your code. Constants are similar to variables in that they are named pieces of data; however, once you set the value for a constant, it cannot be changed. For example, you may want to define a constant for pi or constant values for min/max limits in a program.

Java convention is that the name of constants should be in all caps and should use underscores to separate words in the name. You should also declare constants using the **final** keyword as in the following examples:

```
final int MAX_AGE = 99;

final float PI = 3.14f;
```

It is good practice to use constants rather than literal values in your code. The use of constants makes you code more readable but also makes it more maintainable. Consider the case where the business rules change and the MAX_AGE goes from 99 to 109. If you are using a constant for MAX_AGE, you only have to make this change in one place. If you are using literals, you will have to make the change everywhere you use the literal — what happens if you miss one?

Strings

Strings are used extensively in Java programming and the String class has many useful features such as being able to split a string or retrieve parts of a string. To understand what a given Java class can do, we look at the **Javadoc**. Javadoc is documentation describing the capabilities of a given class or method. This is a great time to get familiar with the String class and Javadoc. Search for "java se 8 String javadoc" and see what you find.

Bitwise Operations

We covered the logical boolean operators in this lesson. The Java language also has bitwise boolean operators and bit shift operators. These are not commonly used, but Oracle has a brief intro here:

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html

Random Numbers

Some of the labs require you to generate a random number over a particular range. For example, you might need to generate a random number between 1 and 10 inclusive. Java has a Random class that does just that for you. To see all of its capabilities, search for the Javadoc associated with the class. To get you started, here is a quick example that generates a random integer value between 1 and 10 inclusive:

```
Random rGen = new Random();
int rInt = rGen.nextInt(10) + 1;
```

It is left as an exercise for the reader to determine why we must add 1 to the value returned from nextInt.

Wrap-up

That's it for this lesson. Here's what we covered:

1. Relational and conditional operators
2. Conditional execution:
    a. if-then-else
    b. switch
3. Looping constructs:
    a. do/while
    b. while
    c. for
4. Constants, Strings, bitwise operations, and random numbers

In the next lesson, we'll take a deeper look at using the debugger.

Additional Resources

The topics covered in Lessons 03 and 04 are also covered in the following places online:

- Oracle Java Tutorial
- JavaWorld: Java 101