Java 2: Weekly Videos and Curriculum

Site:	CCP	Printed by:	Elliot Hill
Course:	Backend Software Development (2024)	Date:	Thursday, November 7, 2024, 4:00 PM
Book:	Java 2: Weekly Videos and Curriculum		

Table of contents

1. Boolean Operators and Conditionals

Boolean Operators

07:05

Using programming to automate tasks means the computer needs a way to make decisions. Decisions require comparing and evaluating information available and then deciding which way to proceed. **Boolean values** are essential in these comparisons. With each decision we need the computer to make, we eventually answer in a **yes** or **no** manner - in Java, that **yes** or **no** is represented by: `true` or `false`

Boolean Expression -- Legally Old Enough To Drive Example:

Imagine that we want to know if a person is old enough to drive. To determine that, we need to compare the individual's age: `currentAge` to the age required to drive: `ageRequiredToDrive`

> If the person's `currentAge` is great than or equal to the `ageRequiredToDrive` then the person can drive. If not, the person cannot legally drive. `true` and `false` are the only two options.

Here is an example of how this could be represented in Java:

```
int ageRequiredToDrive = 16;

int currentAge = 14;

boolean canPersonDrive = (currentAge >= ageRequiredToDrive);

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

Some Observations:

- The code above will compare our two variables: `ageRequiredToDrive` and `currentAge`
- We use an `if` statement to determine if `currentAge` is greater than or equal to `ageRequiredToDrive`

  - If the value stored `currentAge` is greater than or equal to `ageRequiredToDrive` the result of the Boolean expression (the operation using the `>=` operator) based on the values assigned to those variables would evaluate to `true`

  - If the value stored in `currentAge` was less than `ageRequiredToDrive`, then the result would be `false`.

- The result of the Boolean expression is assigned to the variable `canPersonDrive` and then printed to the console. In this case, the Boolean expression evaluates to `false`.
- **Note**: If `currentAge` were 16, 17, or another higher number, it would evaluate to `true`

List of **Boolean operators**:

- Less than: `<`
- Greater than: `>`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`
- Equal To (type matters): `==`
- Not Equal To (type matters): `!=`

Conditionals

15:31

Now, simply printing out whether or not a Boolean expression evaluates to `true` or `false` doesn't completely help the computer in making a decision. To make a decision the computer needs to be told that if a Boolean expression evaluates to `true`, then do something, otherwise do something else or even nothing at all. To do this, we use **conditionals**. The most common **conditional** is an `if` statement. `if` statements have the following syntax:

```
if (/*Boolean expression*/) {

  //code to run if Boolean expression in parentheses evaluates to true

}
```

`if` Statement -- Legally Old Enough To Drive Example:

The Boolean expression inside of the parentheses following the `if` statement evaluates first, and if it is `true`, then all the code in between the following opening and closing curly braces will execute. If the Boolean expression evaluates to `false`, then the code in between the curly braces is skipped and does not execute. Using the previous example, we could do something like this:

```
int ageRequiredToDrive = 16;

int currentAge = 14;

boolean canPersonDrive = currentAge >= ageRequiredToDrive;

if (canPersonDrive) {

  System.out.println("This person can drive!");

}
```

In this example, nothing will happen because `canPersonDrive` is `false`.

Coding Challenge: Try increasing the `currentAge` to 16 or higher and run it again!

We can also place the Boolean expression directly inside the parentheses instead of creating a variable to hold the value, if we want. This code produces the same result as the previous version.

```
int ageRequiredToDrive = 16;

int currentAge = 14;

if (currentAge >= ageRequiredToDrive) {

  System.out.println("This person can drive!");

}
```

`if` - `else` Statement -- Legally Old Enough To Drive Example:

What if we want to do something else if the Boolean expression evaluates to false rather than simply doing nothing? Then we can use an `else` statment. An `else` statement follows an `if` statement and will execute only if the preceding `if` statement's Boolean expression evaluates to `false`

```
int ageRequiredToDrive = 16;
int currentAge = 14;
if (currentAge >= ageRequiredToDrive) {
  System.out.println("This person can drive");
} else {
  System.out.println("This person cannot legally drive");
}
```

If the `currentAge` is greater than or equal to `ageRequiredToDrive`, then the code in the first block will execute and "This person can drive" will be printed. However, if the expression evaluates to `false` (as it will in this case since `currentAge` is only 14), "This person cannot legally drive" will be printed. Thus we've enabled the computer to make a decision based on comparing data.

`if` - `else if` - `else` Statement -- How Many Eggs Example:

Sometimes, there are more than two options in a decision. For example, what if the decision to be made was how many eggs to purchase based on how much each dozen costs?

1. If a dozen of eggs costs $3 or more, we may only want to purchase one dozen.
2. If they are less than $3 but greater than $2 per dozen, we may buy 2 dozen.
3. If they are less than $2, we may buy 3 dozen.
4. And finally, If they are less than a dollar, we want to buy 4 dozen.

To do this, we can add some `else if` statements to our decision:

- `else if` statements work similarly to `if` statements in that they contain a set of parentheses with a Boolean expression and will only execute if that expression evaluates to `true`
- However, they also function like an `else` statement in that they will not run if the previous `if` or `else if` Boolean expression is `true`.

Once one of the Boolean expressions evaluates to `true`, that code block will run and the rest will be skipped. If none evaluate to `true`, the final `else` statement is the default code that will run. For example:

`if – else if – else` Example:

```
double costOfEggs = 2.12;
int numberOfDozensOfEggsToPurchase = 0;
if (costOfEggs > 3) {
      numberOfDozensOfEggsToPurchase = 1;
} else if (costOfEggs > 2) {
      numberOfDozensOfEggsToPurchase = 2;
} else if (costOfEggs > 1) {
```

```
            numberOfDozensOfEggsToPurchase = 3;
        } else {
            numberOfDozensOfEggsToPurchase = 4;
        }
        System.out.println("I will buy " + numberOfDozensOfEggsToPurchase + " dozen eggs.");
```

If we have a logical decision flow that has many paths, we could use a bunch of `else if` statements, with a single `else` statement at the very end that defines the default code to execute if all of the previous Boolean expressions in the `if` and `else if` statements evaluate to `false`

`switch` Statement -- Grade Range Example:

There is also another programming construct we can use to create logical paths with multiple options in a similar fashion. This construct is called a `switch` statement and is used to evaluate a variable and then provide multiple different code blocks that could be executed based on the value of the variable.

`switch` Grade Range Example:

```
        char grade = 'D';
        switch (grade) {
            case 'A':
                System.out.println("90-100");
                break;
            case 'B':
                System.out.println("80-89");
                break;
            case 'C':
                System.out.println("70-79");
                break;
            case 'D':
                System.out.println("60-69");
                break;
            default:
                System.out.println("0-59");
        }
```

Resources:

- The Java Tutorials -- if - else - if else
- The Java Tutorials -- switch

2. Loops

11:22

Example:

Let's say we need to do something over and over again until some condition is met. For example, if we are baking a cake and a recipe calls for 5 cups of flour, we need to scoop a cup of flour and put it into our mixing bowl over and over until the bowl has 5 cups of flour. A simple example of this in Java could look something like this:

```
int cupsOfFlour = 0;
System.out.println("Scooping a cup of flour into the bowl.");
cupsOfFlour += 1;
System.out.println("There are " + cupsOfFlour + " cups of flour in the bowl.");
System.out.println("Scooping a cup of flour into the bowl.");
cupsOfFlour += 1;
System.out.println("There are " + cupsOfFlour + " cups of flour in the bowl.");
System.out.println("Scooping a cup of flour into the bowl.");
cupsOfFlour += 1;
System.out.println("There are " + cupsOfFlour + " cups of flour in the bowl.");
System.out.println("Scooping a cup of flour into the bowl.");
cupsOfFlour += 1;
```

```
System.out.println("There are " + cupsOfFlour + " cups of flour in the bowl.");

System.out.println("Scooping a cup of flour into the bowl.");

cupsOfFlour += 1;

System.out.println("There are " + cupsOfFlour + " cups of flour in the bowl.");
```

There are two problems with the code above. While it does exactly what we're looking for.

1. It contains a lot of **duplicate code**
2. It is not **dynamic**.

For the first problem, it contains **duplicate code** - as developers, we never want duplicate code. Duplicated code means more places to make changes if we ever need to change anything, and more places to make changes means more opportunities for something to be missed or a mistake to be made.

> Follow the **DRY** principle - **Don't Repeat Yourself**.

For the second problem, what if the requirement changed to 6 or 7 cups? What if it changed to 4? Both situations would cause us to need to change the code. This is a brittle solution that breaks as soon as the number of cups of flour required changes. Which leads to the problem, what do we do when we need code that will repeat until a condition is met? Enter, **Loops**.

Loops do just that. Similar to an `if` statement, loops contain parentheses with specific conditions and a body denoted by curly braces that will execute again and again until the condition in the parentheses evaluates to `false`. There are many different kinds of loops. Let's take a look at our first loop, the `while` Loop.

`while` Loop Example:

```
int cupsOfFlour = 0;

while (cupsOfFlour < 5) {

  System.out.println("Scooping a cup of flour into the bowl");

  cupsOfFlour += 1;

  System.out.println("There are " + cupsOfFlour + " cups of flour in the bowl.");

}
```

The above solution is a much cleaner way to do the exact same thing as the 16 lines in our first example and it solves both problems. 1) it does not contain duplicate code; it is DRY. And 2) we can change the condition to 6, 7, 4, or 10000000 by simply switching the number in the Boolean expression.

The way this works is that the Boolean expression 'cupsOfFlour < 5' will evaluate and if it evaluates to `true` the code inside the body (between the opening and closing curly brace) will execute. After it executes, the Boolean expression will then evaluate again. If it is `true`, the body will again execute. Each execution is known as an iteration. The loop will continue to iterate until the Boolean expression evaluates to `false`. This is why the line 'cupsOfFlour += 1;' is so important; because without it, cupsOfFlour will remain 0 and the Boolean expression will never be `false`, thus resulting in an infinite loop (a loop that never ends).

The next type of loop is a `for` Loop. `for` Loops contain a bit more syntax, but do more with fewer lines. Let's look at the same example as above converted to a `for` Loop:

`for` Loop Example:

```
for (int cupsOfFlour = 1; cupsOfFlour <= 5; cupsOfFlour++) {
  System.out.println("Scooping a cup of flour into the bowl.");
  System.out.println("There are " + cupsOfFlour + " cups of flour in the bowl.");
}
```

Notice this loop is two lines shorter and does the exact same thing. A `for` Loop has three sections inside its parentheses separated by two semicolons.

1. The first section is where we can declare any variables to be used in the loop. In this situation, we set our `cupsOfFlour` variable (the most common variable here in a `for` loop is `i` - read on for an example).
2. The second section is where we put our Boolean expression, in this case `cupsOfFlour <= 5` that determines whether or not the loop performs an iteration.
3. The final section is the post-iteration, it is what happens after the loop completes an iteration. In this case, we use `cupsOfFlour++` here, which is essentially the same thing as `cupsOfFlour += 1`

As stated above, `i` is the most common variable name in a `for` Loop. Here is an example that prints from 0 to 9:

`for` Loop Example #2:

```
for (int i = 0; i < 10; i++) {
  System.out.println(i);
}
```

We can also use loops and conditionals together, they don't have to be separate. We can have loops inside of `if` statements, or vice versa. Here is an example of an `if` statement inside of a loop that prints out every number from 0 to 99 divisible by 3.

`for` Loop Example #3:

```
for (int i = 0; i < 100; i++) {
  if (i % 3 == 0) {
    System.out.println(i);
  }
}
```

Another type of loop is called a `do while` loop. This loop functions much like a `while` loop, except that a `while` loop has the possibility of never running if its Boolean expression evaluates to `false` the first time, and a `do while` loop will always execute at least once since the expression is at the end. Let's take a look at an example.

`do while` Loop Example:

```
int i = 10;
do {
  i++;
  System.out.println(i);
} while (i < 3);
```

As we can see here, `i` is already greater than 3, but this loop will still iterate once and then exit.

Resources:

- The Java Tutorials -- for Statement
- The Java Tutorials -- while and do-while Statements

3. User Input

User Input

12:48

We need data to tell our programs to make decisions with; and up to this point, we've been hard coding data into variables to explore coding constructs. However, the original source for most data is **user input**. In order to make decisions based on responses or data entered from a user, let's take a look at one way we can prompt a user to enter some data and then store the data in a variable to use in our code.

Note: this method of interacting with a user is a temporary method for the purpose of being able to receive user input and is not the recommended way in live, production code to interact with users. We will learn additional ways later on.

To receive **user input** in Java, we can use the Scanner object. It is in java.util, so we need to import it.

For example:

```
import java.util.*;
class UserInputDemo {
    public static void main(String[] args)  {
        // System.in is a standard input stream
        Scanner sc= new Scanner(System.in);
        System.out.print("Enter username:  ");
        String username = sc.nextLine();
        System.out.print("Enter password: ");
        String password = sc.nextLine();
        if (username.equals("samy123") && password.equals("12345")) {
                System.out.println("Welcome back " + username);
            } else {
                System.out.println("Inaccurate credentials!");
```

```
            } // end of else
        }   // end of main()
    }  // end of UserInputDemo class
```

The above code will prompt a user for a username and a password and if the username and password match "samy123" and "12345" relatively, the user will be welcomed. If not, the user will see a message saying their credentials are not correct.

With the above example, we have to run the program each time to run it again. So, if a user puts in the incorrect credentials, they see the message and nothing happens until they run the application again.

To have a better user experience, we would most likely want to prompt the user for their credentials again after a failed login attempt. We can do this with a **loop**.

Let's try the following:

```
boolean loggedIn = false;
Scanner sc= new Scanner(System.in);
while (!loggedIn) {
    System.out.print("Enter username: ");
    String username = sc.nextLine();
    System.out.print("Enter password: ");
    String password = sc.nextLine();
    if (username.equals("samy123") && password.equals("12345")) {
        System.out.println("Welcome back " + username);
        loggedIn = true;
     } else {
        System.out.println("Inaccurate credentials!");
    } // end of else
} // end of while
```

In the above code, we use a Boolean variable as a flag to determine whether the user is logged in or not. If the user enters the wrong credentials, nothing happens to the `loggedIn` variable, so it remains `false` causing the loop to iterate again and again until the user enters the correct credentials, at which point we update the `loggedIn` variable to `true`, which will cause the loop to stop iterating.

Coding Challenge:

We could also add a login attempt count that would enable the user to only enter the incorrect credentials a certain number of times before displaying a message like "You are locked out!" and ending the loop.

Challenge: See if you can figure out how to implement this enhancement, using the above code as a starting point.