Description

It's time to revise flow of control in our programs, which lets us make decisions and repeat chunks of code. This week, we'll see how a Java program can decide to perform one of several alternative courses of action and loop. To do this, we make use of boolean variables, comparison operators, and structures called if, if...else, and switch; for, while and do-while. We'll touch on the conditional operator as well.

We'll also see how to nest loops inside each other. We'll also see using break and continue to exit a loop early. We'll take a look at **String's** class toString() method and implement it.

Boolean Data Type; Comparison operators

The boolean Primitive Data Type

- boolean is one of the eight primitive data types. We have already met the other seven.
- You declare a boolean variable in the same way that you declare any other variable: `boolean termUp;`
- A boolean variable can have only two possible values: true or false.
- true and false are not keywords, but they behave just like keywords. They have a special meaning for the compiler, they are all lower-case (True or TRUE are not boolean values) and they cannot be used as identifiers: `termUp = true;`
- boolean variables are often referred to as *flags* because they indicate whether something is true or not. For example, a boolean field called alive, if true, indicates that something is alive, but, if it's false, it indicates it's dead.
- When using booleans as fields in your classes, it's a Java convention to name the getter differently than with other data types. If I have a boolean field called termUp, the getter for it would be called isTermUp() (rather than getTermUp() as would be the case if termUp was another data type). Using is rather than get makes it more obvious that the getter will return a boolean value (either true or false):

```
public boolean isTermUp() {
        return termUp;
}
```

Comparison Operators

- *comparison operators* are used to create conditions (also called expressions) that evaluate to a boolean result (true or false). These are used in if statements and loops.
- There are six comparison operators: == (equals, note this is 2 equal signs), > (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to), and != (not equal) . Each of these is a binary operator, i.e. it takes two

operands. Each of these operators compares the two operands and returns a boolean result, either true or false.
- The symbols for most of the comparison operators use two characters. You cannot leave a space between these characters. Nor can you reverse the order of the characters. Less than or equal to must be written <=, not < = or =<.

Not Operator

- ! (not) is a unary operator. It takes one operand, which must have a boolean value (true or false), and gives a single boolean value as its result. If the boolean variable named "even" has the value true, then !even is false. If even has the value false, then !even is true. It can be used with conditions as well.

Compound Conditions

- Compound conditions work like And and Or in algebra. We use && (and) and || (or), which are binary operators. Each of these takes two operands, both of which must have boolean values (true or false), and combines them to give a single boolean value.
- Don't use the single ampersand (&) and single pipe (|) operators. Always use the double symbols (&&, ||).
- Here are the truth tables for &&, || and !:

TRUTH TABLES

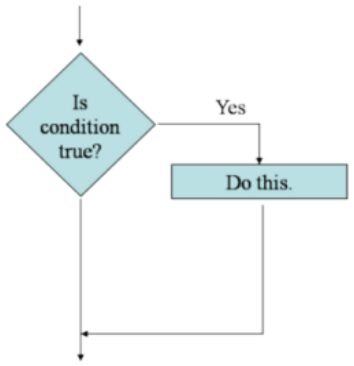
AND **OR** **NOT**

Exp_1	Exp_2	Exp_1 && Exp_2	Exp_1 || Exp_2
true	**true**	True	true
true	**false**	False	true
false	**true**	False	true
false	**false**	False	false

Exp	**!(Exp)**
True	false
False	true

<u>The if statement</u>

When we want to be able to decide at run time whether or not to execute a statement or group of statements, we can use an if statement to execute only the code we want and skip the rest. We do this using the keyword "if" and a boolean condition (expression or test). This is probably the most common use of booleans. The condition produces a boolean (true or false) value. The boolean value determines which statements are executed next.



Here's a really basic if in Java:

```
if (var > 0)
System.out.println(var + " is positive.");
```

var > 0 is a boolean condition. If the result of testing the boolean condition is true (if the value in the variable var is greater than 0), the statement(s) after the condition will execute. In this simple format, nothing happens if the condition is false.

Important points to note:

- “if” is a Java keyword. It must be all lower case. The compiler will not recognize “If” or “IF”.
- "if" must be followed by parentheses, and the parentheses contain a condition. The condition (the expression inside the parentheses) must evaluate to a boolean result, either true or false.
- The body of if statement can be either a single statement or several statements. If there is more than one statement, the statements must be enclosed in braces (this is called a block of code). If there is only one statement, the braces are optional.
- Follow the same rules for alignment and indentation that we used for classes and methods: the opening brace at the end of the header line, the closing brace aligned with the first column of the header (the “i” in “if”), and the entire body within the braces indented relative to the header. Take note of this alignment and indentation in the examples here and in the text.
- If the body of an if statement has only one line, indent this line, even if you are not using braces:

Remember once again that indentation is just to clarify the structure of the code for human readers. The compiler does not care about indentation.

Be careful not to put a semicolon at the end of the header line of if statement:

```
// VERY BAD
if (var > 0); //semi colon at end of condition
System.out.println(var + " is positive.");
```

This is not a syntax error, so your program will compile and run. It is, however, a logic error, so the program will give the wrong results. In the code above, the entire body of the if statement is just the semicolon. The line below is **not** part of the if statement at all, so it will always be executed, whether or not var is greater than zero.

Comparing to true / false

You never need to compare a boolean value to the constants true or false using the == or != operators. It is not a syntax error to do so, but it is a rookie move. If you have a boolean variable named “even” and you want to execute a statement when it is true, don’t write:

if (even == true)
// body of if statement

Just write:

if (even)
// body of if statement

Similarly, if you want to execute a statement only when even is false, don't write:

if (even == false)
// body of if statement

Instead, write:

if (!even)
// body of if statement

(Recall that ! is the boolean "not" operator. It turns true into false and vice versa.)

The if…else statement

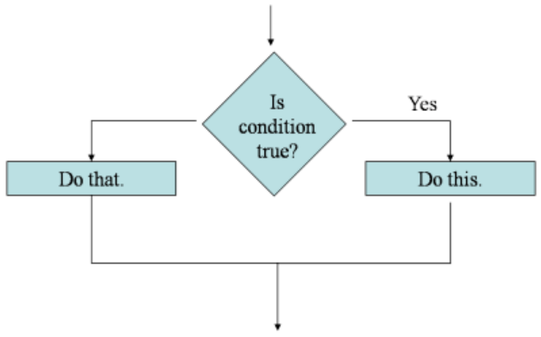
With the basic if statement, the body of the statement is executed if a condition is true, and is skipped over if that condition is false. To do this, we use an if…else statement:

if (var> 0)

System.out.println(var + " is positive");

else

Either the body of the if clause is executed (when the condition is true), or the body of the else clause is executed (when the condition is false), but not both



Example (if-else-if)

Let's code a lottery game where the user tries to guess a number between 1 and 50 to win money. If guess is not correct, we lose.

```
public class Lottery {
    private int guessNumber;
    private int lotteryNum;

    public int generateLotteryNum(Random random){
        return lotteryNum = random.nextInt(50) + 1;    }
```

Lottery number is generated randomly. For this purpose, we pass Random class reference variable as a parameter. Random's nextInt(int bounds) generates a number between 0 and bounds – 1. For instance, nextInt(10) generates any number between 0 and 9. As we want a number between 1 and 50, so add one to it.

```
public String pick(int guess){

      String displayString = "";

      int difference = Math.abs(lotteryNum - guessNumber);

          if (difference == 0) {

              displayString = "Exact match: You win $ 500";

          } else {

              displayString = "Sorry, you lose";

          }
```

Let's add more options in the game to make it interesting.

They win most if the number is exact, less if all digits are close (within 2 of a number), less again if digits are within 4 of a number and they lose if there are no matches at all. For instance, if number to guess is 34 then here are the different alternatives:

1. Win most if number entered is 34
2. Win half if number entered is 32, 33, 35 or 36
3. Win lesser if number entered is 30, 31, 37, 38

Lose if none of above

```
if (difference == 0) {

          displayString = "Exact match: You win $ 500";

       } else if (difference == 2 || difference == 1) {

          displayString = "Within 2: You win $250";

       } else if (difference == 4 || difference == 3) {

          displayString = "Within 4: You win $100";

       } else {
        displayString = "Sorry, you lose";
    }
```

Nested If statements and else-if

If statements can be nested inside one another. The snippet below shows a lottery game where the user tries to guess a number between 1 and 50 to win money. If-else-if block is nested inside else.

```
if(guess < 1 || guess > 50){
      displayString = "Invalid number, please enter again";
   }else {  //nesting
      guessNumber = guess;
 //Math.abs() - if an argument is positive then no changes are made otherwise -ve
argument becomes +ve
      int difference = Math.abs(lotteryNum - guessNumber);
      if (difference == 0) {
         displayString = "Exact match: You win $ 500";
      } else if (difference == 2 || difference == 1) {
         displayString = "Within 2: You win $250";
      } else if (difference == 4 || difference == 3) {
         displayString = "Within 4: You win $100";
      } else {
         displayString = "Sorry, you lose";
      }
   }
```

Let's try if else statements in BankAccount application.

The following method in BankAccount class is modified as:

```
public void setBalance(double bal) {

        if(bal > 0)

          balance = bal;

        else

           balance = MIN_BALANCE;

    }
```

Switch statement

Here is the lottery program from the previous section with a portion of it redone using switch instead of if:

```
public String pick(int guess){
        String displayString = "";
        if(guess < 1 || guess > 50){
            displayString = "Invalid number, please enter again";
        }else {
            guessNumber = guess;
            int difference = Math.abs(lotteryNum - guessNumber);
            switch(difference){
                case 0:
                   displayString = "Exact match: You win $ 500";
                   break;
                case 1:
                case 2:
                   displayString = "Within 2: You win $250";
                   break;
                case 3:
                case 4:
                   displayString = "Within 4: You win $100";
                   break;
                default:
                   displayString = "Sorry, you lose";
            }
        }
        return displayString;
```

“switch” is a Java keyword. It must be followed by an expression in parentheses that evaluates to an int (or another primitive type that can be converted to an int without doing a cast.) Usually this is simply an int or char variable, as in the above example.

The body of the switch statement has several case labels, which look like this:

```
case 1:
```

Notice that these labels end with a colon, not a semicolon. The switch statement compares the value of the expression in parentheses (the variable "result" in the above example) with each of the case labels, until it finds one that matches. It begins executing statements from this point, until either the end of the switch statement is reached or a break statement is executed. The break statement causes the switch to terminate.

You almost always want a break statement at the end of each case, but it is not a syntax error to leave it out. Copy the above code into a file called SwitchTest.java and compile and run it. What happens if the break statements are deleted? *The correct case statement(s) will execute, but so will all the others after it!*

If none of the cases match the switch expression, then execution begins at the “default:” label. This label is optional. If none of the cases match and there is no default label, then the switch does nothing at all.

The conditional operator

Conditional Operator

The value of a variable often depends on whether one boolean expression is true or false and on nothing else. For instance, one common operation is setting the value of a variable to the maximum of two quantities. In Java you might write:

```
if (a > b) {
max = a;
}
else {
max = b;
}
```

Setting a single variable to one of two states based on a single condition is such a common use of if-else that a shortcut has been devised for it, the conditional operator: ?. Using the conditional operator you can rewrite the above example in a single line like this:

```
max = (a > b) ? a : b;
```

? is a *ternary operator*, i.e. one that requires three operands. (a > b) ?a : b; is an expression which returns one of two values, a or b. The first operator is the condition.

During execution, the first operator, (a > b), is tested. If it is true, the second operand, a, is returned. If it is false, the third operand, b, is returned. The value is returned is dependent on the conditional test, a > b.

The condition can be any expression which returns a boolean value (true or false). The second and third operands will always have the same data type as each other: if the second operand is an int, the third will also be an int. If the second is a string, the third will be a string, etc. The conditional operator generates a result that equals either the second operand (if the first operand is true) or the third operand (if the first operand is false).

The for loop

For loops, the first type of loop we'll cover, are also called counted loops since they execute a specific number of times and they use a counter (often called an index) to keep track of the number of times they have executed. When we're using a **for** loop, we very often (though not always) need to do the following three tasks to control the execution of the loop:

1. Before the loop begins, initialize a variable that serves as a counter (index) for the iterations of the loop.
2. At the beginning of each iteration, test the value of the counter to see if we should do another iteration.
3. At the end of each iteration, change the value of the counter.

A for loop groups the above three tasks together in one line in your code.

The header line of a for loop is made up of the Java keyword "for" (as usual for keywords, it's all lower case), followed by parentheses. Inside the parentheses are exactly two semicolons – no more and no less:

```
for ( ; ; )
```

This divides the space in the parentheses into three regions. I call these the three "slots" of the loop (that's not an official Java name for them). You put Java code in each one of these slots. Each slot has a different role.

The code in the first slot is executed **once only**, before execution of the for loop begins. Usually the first slot contains an assignment statement to initialize a variable that serves as a loop counter (index):

```
for (ii = 0; ii < 100; ii++) {
System.out.println(ii);  //body of loop
}
```

This will initialize the variable ii by setting it to 0.

The code in the second slot is evaluated at the beginning of **every iteration** of the loop. The second slot must contain exactly one Java condition (expression or test) that evaluates to a boolean value (true or false). Usually this is a comparison operation. When this expression is true, the body of the loop is executed. When it is false, the loop is complete, and the program goes on to the next statement after the loop.

The code in the third slot is evaluated **after** each iteration of the loop. This slot usually contains exactly one statement, and this usually increments the value of the counter.

To summarize, here's how a for loop works:

1. Execute the code in the first slot (usually initializing the counter)
2. Test the condition in the second slot (always giving a boolean result). If false, quit the loop. If true, execute the body of the loop.
3. When you reach the end of the code in the body of the loop, execute the code in the third slot. Then repeat item (2) above (not item 1).

The complete life cycle of the loop is made up of execution of:

1, 2, body, 3, 2, body, 3, 2, body, … 3, 2

until 2 gives a result of false. The loop then terminates and the program continues execution at the statement after the body of the loop.

The usual purpose of the code in the third slot is to modify the value of the counter that was initialized in the first slot. There is nothing to stop you from also changing the value of the counter in the body of your loop, but your programs will be much clearer if you avoid doing this. The main reason for using a **for** loop instead of a while loop is that the for loop groups together all of the tasks that control the iterations of the loop. Changing the value of the loop counter in the body defeats this purpose.

By the way, it's risky to use floating-point numbers (float or double) as counters to control loop execution. Round-off errors can cause the loop to execute one more or one fewer times than expected. It's much safer to use **integer** data types as counters. There's nothing wrong with using floating-point types in the body of your loop, but avoid using them in the header statements.

As with if statements, be careful not to put a semicolon after the loop's header line - if you do, it won't give you a compile error, but the body of your loop will not be executed as part of the loop, but will be executed as the next statement once the loop (which has no body except the semi-colon) is over.

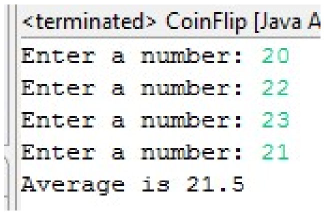
The for loop examples

Here are few more samples Java for loops to give you the idea.

The first one shows how to ask the user to enter 4 numbers, then find and print the average of the numbers:

```
//(Finding the average number) Here's a program that prompts the user to enter
4 numbers, then displays the average of the numbers.
Scanner keyboard = new Scanner(System.in);
double total = 0;
for (int i = 0; i< 4; i++) {
        System.out.print("Enter a number: ");
        double number = keyboard.nextDouble();
        total += number;
}
System.out.print("Average is " + total / 4);
```

Here's some sample output from a run of the loop above:

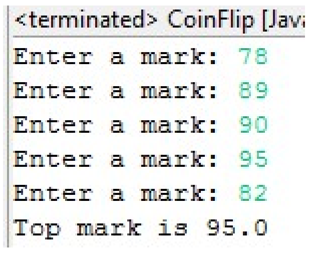


```
<terminated> CoinFlip [Java A
Enter a number: 20
Enter a number: 22
Enter a number: 23
Enter a number: 21
Average is 21.5
```

This next one shows how to ask the user to enter 5 marks, then find and print the highest mark:

```
//(Finding the highest mark) Here's a program that prompts the user to enter 5
marks and displays the highest mark.
Scanner keyboard = new Scanner(System.in);
doublehighMark = 0;
for (int i = 0; i< 5; i++) {
        System.out.print("Enter a mark: ");
        double mark = keyboard.nextDouble();
        if (mark > highMark) {
                highMark = mark;
          }
}
System.out.print("Top mark is " + highMark);
```

Here's some sample output from a run of the loop above:



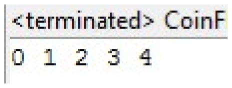
```
<terminated> CoinFlip [Jav
Enter a mark: 78
Enter a mark: 89
Enter a mark: 90
Enter a mark: 95
Enter a mark: 82
Top mark is 95.0
```

The snippet below store the numbers from 1 to 4 in a string. The first loop body is a single line, the second uses a block enclosed in {} since there are two statements in the loop. As with if statements, it's a good idea to put the braces in even if the body of the loop is only one line. That way, if you need to add more statements later, you won't forget to add the braces and introduce a bug:

```
//With a single statement
String numbers = "";
for (int i = 0; i < 5; i++)
   numbers += i + " ";
```

```
//With a block of statements
String numbers = "";
for (int i = 0; i < 5; i++) {
     numbers += i;
     numbers += " ";
 }
```

Here's some sample output from a run of the loop above:



```
<terminated> CoinF
0 1 2 3 4
```

The loop below adds the numbers 8, 6, 4 and 2. This one demonstrates starting a loop at a **higher** index value and **decreasing** it as the loop continues. This loop will print the number 20 to the console:

```
int sum = 0;
for (int j = 8; j > 0; j -= 2) {
        sum += j;
}
System.out.println(sum);
```

Nested Loops

As with if statements, loops can be nested. Here is a multiplication table example (explanation below):

```
// An example to illustrate nested for loops.
// The important part of this program is the section headed "Print table body"

// Display the table heading
String output = "          Multiplication Table\n";

// Display the number title
output += "     ";
for(int j = 1; j<= 9; j++)
output += "   " + j;

output += "\n";
output += "-------------------------------------------\n";
```

```
// Print table body
for(int i = 1; i<= 9; i++){
output += i + " | ";
for(int j = 1; j<= 9; j++){
// Display the product and align properly
if(i * j<10)
output += "    " + i * j;
else
output += "  " + i * j;
}
output += "\n";
}

// Display result
System.out.println(output);
```

The first part of this program is just printing a heading and drawing lines to outline the table. The important part of this example is the code where we have:

```
for(int i = 1; i<= 9; i++){
output += i + " | ";
for(int j = 1; j<= 9; j++){
// body of inner loop
}
// outer loop continues
}
```

The outer loop is executed nine times. On **each** iteration of the outer loop, the inner loop goes through its entire life cycle, with nine iterations. So, the inner loop is executed 81 times in total.

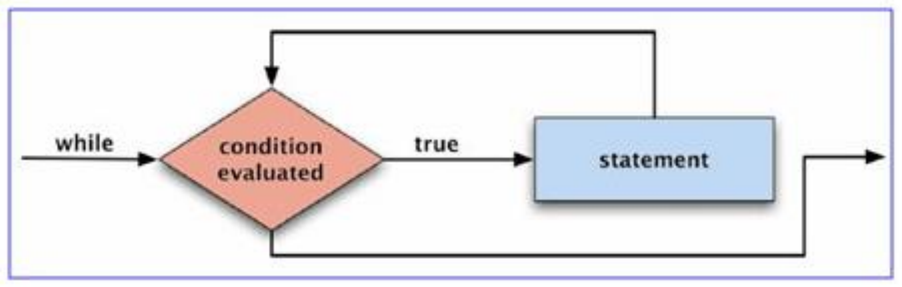
The while loop

The second type of loop Java provides is the while loop. While loops are useful when you don't know how many times the loop needs to execute.

Here is a simple snippet that counts from 1 to 5:

```
int n = 1;
while (n <= 5) {
  System.out.println("n = " + n);
  n++;
}
```

The syntax for a while loop is very similar to that of an if statement: Java keyword while (instead of if), followed by parentheses, and an expression inside the parentheses that gives a Boolean result. When the expression evaluates to true, the body of the statement is executed. When it is false, the body of the while statement is skipped, and the program continues with whatever comes next. Here's a while loop flowchart:



For the first time through the code, while works exactly the same way as if. The difference is what happens once the body of the statement has been executed. With if, the program continues onward to the next statement after the if structure. With while, it goes back to the beginning of the statement and tests the condition again. If it is still true, the body is executed a second time. This continues over and over, until the condition becomes false. So, the above program prints:

```
n = 1
n = 2
n = 3
n = 4
n = 5
```

After it prints n = 5, it executes n++, which changes the value of n to 6. It then goes back to the beginning of the while loop and tests the condition n <= 5. 6 is not less than or equal to 5, so the condition returns false, and the loop terminates.

Remember that each time the body of a loop is executed, we say that there is an *iteration* of the loop. The loop above goes through five iterations before it terminates.

Just as with if, the body of a while loop could contain a single statement, or several statements. If there are more than one, they must be enclosed in braces

The condition (the expression in the parentheses) is usually a comparison that involves one or more variables. For the loop to terminate there must be a statement in the body of the loop that changes the value of one of these variables. If there isn't, then if the condition in parentheses is true initially, it will always be true, and the loop will continue forever. For example, suppose we omitted the line n++ from the example above:

```
int n = 1;
while (n <= 5); {
  System.out.println("n = " + n);
}
```

Now n will always be 1, so the program will print out:

```
n = 1
n = 1
n = 1
n = 1
n = 1
......
```

and will never stop. This is called an *infinite loop*, and is a common programming error. If your program is in an infinite loop in NetBeans, you can stop it by clicking the red box in the Console window.

Be careful not to put a semicolon at the end of the while header line. If you write:

```
int ii = 1;
while (ii <= 5); {
  System.out.println("ii = " + ii);
  ii++;
}
```

then the code:

```
{
  System.out.println("ii = " + ii);
  ii++;
}
```

is not part of the loop at all. The body of the loop is just the semicolon, which does nothing. The variable ii is initialized to 1, and then tested to see if it is less than or equal to 5. This is true, so the body of the loop is executed. The body is just the semicolon. We're now at the end of the loop, so we go back to the beginning and test the condition again. We've done nothing to change ii, so it is still less than 5, so we execute the body (the semicolon) again. This will continue forever, and the program will never even reach the line that prints some output.

The do-while loop

The last type of Java loop we will cover is a variation of the while loop called the do-while loop.

The while loop we saw earlier tests its continuation condition **before** each iteration, whereas the do-while loop tests its continuation condition **after** each iteration. It is possible that the body of a while loop will never be executed at all (if the condition is false the first time it is tested), but the body of a do-while loop is **always** executed at least once.

As discussed earlier, it is important that you do not put a semicolon after the condition of a while loop. Doing so will give an infinite loop:

while (ii < 10); // WRONG!!!

In contrast, the condition of a do-while loop **must** be followed by a semicolon:

```
do {

// body of loop

} while (ii < 10); // semicolon needed here
```

In practice, do-while loops are used much less often than while loops or for loops (which we've already discussed). The most common use of them is when reading files (loop while there is still data in the file) and getting user input (loop while the input is invalid). There's nothing wrong with do-while loops - you should know their syntax, and they're occasionally very useful - but it turns out that you don't need them most of the time.

Here's an example of a do-while loop that keeps asking the user to input a number between 100 and 1000 as long as they enter a number NOT in that range:

```
do {
   System.out.println("Enter a number between 100 and 1000"));
   numTimes = scan.nextInt();
} while (numTimes < 100 || numTimes > 1000);
```

break and continue

Every so often, you will find that you need to change the normal processing in a loop. There are two ways to do this: break and continue.

break

Sometimes you need to exit a loop before it has completed processing on its own. This is done with the break statement. An example is when you have an interest calculating loop that will continue for the number of the years in the term. In the case below, assume the term is 10 years. At the end of each iteration, the user is asked if they would like to continue. If they answer No, you want to break out of the loop to stop the program. The following snippet stops the loop if the user answers No:

```
for (int counter = 0; counter < term; counter++)
{
     //code needed to calculate the interest for one year goes here.
     //ask user whether they want to continue
     System.out.println("Would you like to continue? (0 for no, 1 for yes");
     int iInput = scan.nextInt();
     if (iInput == 0)
     {
         //stop loop by breaking out of it
         break;
     }
}
```

Notice that here, if there is more code in the body of the loop after the break statement, the current iteration does NOT execute the remaining code in the loop body, and no more iterations of the loop get executed.

continue

Sometimes you need to skip only the **current** iteration of the loop, but continue with the remaining iterations. You do this with the continue statement. The following snippet skips the iteration if the random number is less than or equal to 7. The rest of the iterations will still be executed:

```
Random random = new Random();
for (int index = 1; index < 10; index++) {
     int number = random.nextInt(10);
     System.out.println(number);
     if (number <= 7) {
          continue;
     }
     System.out.println("This number is greater than 7");
}
```

toString()

We will add a new method called toString() to BankAccount. It returns a string. This is a method included in many classes, and it is used to describe the object in a way the programmer sees fit. In our case, it should include the account id and balance.

```
@Override
  public String toString(){
    return "Balance in account with ID " + getAccountId() + " is $" + getBalance();
  }
```