Java for intermediate:

Java Math

The Java Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.max(*x*,*y*)

The `Math.max(x,y)` method can be used to find the highest value of *x* and *y*:

Example

```
Math.max(5, 10);
```

Math.min(*x*,*y*)

The `Math.min(x,y)` method can be used to find the lowest value of *x* and *y*:

Example

```
Math.min(5, 10);
```

Math.sqrt(*x*)

The `Math.sqrt(x)` method returns the square root of *x*:

Example

```
Math.sqrt(64);
```

Math.abs(*x*)

The `Math.abs(x)` method returns the absolute (positive) value of *x*:

Example

```java
Math.abs(-4.7);
```

Random Numbers

`Math.random()` returns a random number between 0.0 (inclusive), and 1.0 (exclusive):

Example

```java
Math.random();
```

To get more control over the random number, for example, if you only want a random number between 0 and 100, you can use the following formula:

Example

```java
int randomNum = (int)(Math.random() * 101);  // 0 to 100;
```

Java Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, Java has a `boolean` data type, which can store `true` or `false` values.

Boolean Values

A boolean type is declared with the `boolean` keyword and can only take the values `true` or `false`:

Example

```
boolean isJavaFun = true;

boolean isFishTasty = false;

System.out.println(isJavaFun);     // Outputs true

System.out.println(isFishTasty);   // Outputs false
```

However, it is more common to return boolean values from boolean expressions, for conditional testing (see below).

Boolean Expression

A Boolean expression returns a boolean value: `true` or `false`.

This is useful to build logic, and find answers.

For example, you can use a comparison operator, such as the **greater than** (`>`) operator, to find out if an expression (or a variable) is true or false:

Example

```
int x = 10;

int y = 9;

System.out.println(x > y); // returns true, because 10 is higher than
```

Or even easier:

Example

```
System.out.println(10 > 9); // returns true, because 10 is higher than 9
```

In the examples below, we use the **equal to** (`==`) operator to evaluate an expression:

Example

```
int x = 10;

System.out.println(x == 10); // returns true, because the value of x is
equal to 10
```

Example

```
System.out.println(10 == 15); // returns false, because 10 is not equal to
15
```

Real Life Example

Let's think of a "real life example" where we need to find out if a person is old enough to vote.

In the example below, we use the `>=` comparison operator to find out if the age (`25`) is **greater than** OR **equal to** the voting age limit, which is set to `18`:

Example

```
int myAge = 25;

int votingAge = 18;

System.out.println(myAge >= votingAge);
```

Example

Output "Old enough to vote!" if `myAge` is **greater than or equal to** `18`. Otherwise output "Not old enough to vote.":

```
int myAge = 25;

int votingAge = 18;
```

```
if (myAge >= votingAge) {

  System.out.println("Old enough to vote!");

} else {

  System.out.println("Not old enough to vote.");

}
```

Java If ... Else

Java Conditions and If Statements

You already know that Java supports the usual logical conditions from mathematics:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

The if Statement

Use the `if` statement to specify a block of Java code to be executed if a condition is `true`.

Syntax

```
if (condition) {

  // block of code to be executed if the condition is true

}
```

In the example below, we test two values to find out if 20 is greater than 18. If the condition is `true`, print some text:

Example

```
if (20 > 18) {

  System.out.println("20 is greater than 18");

}
```

We can also test variables:

Example

```
int x = 20;

int y = 18;

if (x > y) {

  System.out.println("x is greater than y");

}
```

Java Switch Statements

Instead of writing **many** `if..else` statements, you can use the `switch` statement.

The `switch` statement selects one of many code blocks to be executed:

Synta

```
switch(expression) {
  case x:

    // code block

    break;

  case y:

    // code block

    break;

  default:

    // code block

}
```

This is how it works:

- The `switch` expression is evaluated once.
- The value of the expression is compared with the values of each `case`.
- If there is a match, the associated block of code is executed.
- The `break` and `default` keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;

switch (day) {

  case 1:

    System.out.println("Monday");

    break;

  case 2:

    System.out.println("Tuesday");

    break;

  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

The break Keyword

When Java reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

The default Keyword

The `default` keyword specifies some code to run if there is no case match:

Example

```java
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

Java While Loop

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

Java While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

Syntax

```
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

Java For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

Another Example

This example will only print even values between 0 and 10:

Example

```
for (int i = 0; i <= 10; i = i + 2) {
  System.out.println(i);
}
```

Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

```
// Outer loop
for (int i = 1; i <= 2; i++) {
  System.out.println("Outer: " + i); // Executes 2 times
  // Inner loop
  for (int j = 1; j <= 3; j++) {
    System.out.println(" Inner: " + j); // Executes 6 times (2 * 3)
  }
}
```

For-Each Loop

There is also a "**for-each**" loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

Java Break

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example stops the loop when i is equal to 4:

Example

```java
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  System.out.println(i);
}
```

Java Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

Example

```java
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  System.out.println(i);
}
```

Break and Continue in While Loop

You can also use `break` and `continue` in while loops:

Break Example

```
int i = 0;
while (i < 10) {
  System.out.println(i);
  i++;
  if (i == 4) {
    break;
  }
}
```

Continue Example

```
int i = 0;
while (i < 10) {
  if (i == 4) {
    i++;
    continue;
  }
  System.out.println(i);
  i++;
}
```

Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

```
String[] cars;
```

We have now declared a variable that holds an array of strings. To insert values to it, you can place the values in a comma-separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You can access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[0]);

// Outputs Volvo
```

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

cars[0] = "Opel";

System.out.println(cars[0]);

// Now outputs Opel instead of Volvo
```

Array Length

To find out how many elements an array has, use the `length` property:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars.length);

// Outputs 4
```

Java Methods

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as `System.out.println()`, but you can also create your own methods to perform certain actions:

Example

Create a method inside Main:

```
public class Main {
  static void myMethod() {
    // code to be executed
  }
}
```

Example Explained

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, `myMethod()` is used to print a text (the action), when it is called:

Example

Inside `main`, call the `myMethod()` method:

```
public class Main {

  static void myMethod() {

    System.out.println("I just got executed!");

  }


  public static void main(String[] args) {

    myMethod();

  }

}


// Outputs "I just got executed!"
```

Java Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `String` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

Multiple Parameters

You can have as many parameters as you like:

Example

```
public class Main {
  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }
  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

Example

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

Consider the following example, which has two methods that add numbers of different type:

Example

```
static int plusMethodInt(int x, int y) {
  return x + y;

}


static double plusMethodDouble(double x, double y) {

  return x + y;

}


public static void main(String[] args) {

  int myNum1 = plusMethodInt(8, 5);

  double myNum2 = plusMethodDouble(4.3, 6.26);

  System.out.println("int: " + myNum1);

  System.out.println("double: " + myNum2);

}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `plusMethod` method to work for both `int` and `double`:

Example

```
static int plusMethod(int x, int y) {

  return x + y;

}


static double plusMethod(double x, double y) {

  return x + y;

}
```

```java
public static void main(String[] args) {
  int myNum1 = plusMethod(8, 5);
  double myNum2 = plusMethod(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
```