JAVA LEARNING

By :-Kuldeep Singh

ABSTRACT

Learning JAVA easily and fast and make your self like an a pro

BY :- Kuldeep Singh

Java

Java Tutorial

Java Methods

[Java Exceptions](#)

[Java RegEx](#)

[Java Threads](#)

[Java Lambda](#)

Java File Handling

[Java Files](#)

[Java Create/Write Files](#)

[Java Read Files](#)

[Java Delete Files](#)

Java How To

[Add Two Numbers](#)

[Count Words](#)

[Reverse a String](#)

[Sum of Array Elements](#)

[Area of Rectangle](#)

[Even or Odd Number](#)

Java Reference

[Java Keywords](#)

[Java String Methods](#)

[Java Math Methods](#)

Java Reference

[Java Keywords](#)

[Java String Methods](#)

How to download Java

Download Java from the official Java web site: https://www.oracle.com

Download Bluej from the official Bluej web site: https://www.bluej.org

Download Visual studio from the official Visual studio web site: https://code.visualstudio.com

Java Introduction

What is Java?

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than **3 billion** devices run Java.

It is used for:

- Mobile applications (specially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection
- And much, much more!

Why Use Java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming language in the world
- It has a large demand in the current job market
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has a huge community support (tens of millions of developers)

- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As Java is close to C++ and C#, it makes it easy for programmers to switch to Java or vice versa.

Java Getting Started

Java Install

Some PCs might have Java already installed.

To check if you have Java installed on a Windows PC, search in the start bar for Java or type the following in Command Prompt (cmd.exe):

C:\Users*Your Name*>java -version

If Java is installed, you will see something like this (depending on version):

java version "11.0.1" 2018-10-16 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.1+13-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.1+13-LTS, mixed mode)

If you do not have Java installed on your computer, you can download it for free at oracle.com.

Note: In this tutorial, we will write Java code in a text editor. However, it is possible to write Java in an Integrated Development Environment, such as IntelliJ IDEA, Netbeans or Eclipse, which are particularly useful when managing larger collections of Java files.

Setup for Windows

To install Java on Windows:

1. Go to "System Properties" (Can be found on Control Panel > System and Security > System > Advanced System Settings)
2. Click on the "Environment variables" button under the "Advanced" tab
3. Then, select the "Path" variable in System variables and click on the "Edit" button
4. Click on the "New" button and add the path where Java is installed, followed by **\bin**. By default, Java is installed in C:\Program Files\Java\jdk-11.0.1 (If nothing else was specified when you installed it). In that case, You will have to add a new path with: **C:\Program Files\Java\jdk-11.0.1\bin**
   Then, click "OK", and save the settings

5. At last, open Command Prompt (cmd.exe) and type **java** **-version** to see if Java is running on your machine.

Java Quickstart

In Java, every application begins with a class name, and that class must match the filename.

Let's create our first Java file, called Main.java, which can be done in any text editor (like Notepad).

The file should contain a "Hello World" message, which is written with the following code:

Main.java

```java
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello World");

  }

}
```

Don't worry if you don't understand the code above - we will discuss it in detail in later chapters. For now, focus on **how** to run the code above.

Save the code in Notepad as "Main.java". Open Command Prompt (cmd.exe), navigate to the directory where you saved your file, and type "javac Main.java":

C:\Users*Your Name*>javac Main.java

This will compile your code. If there are no errors in the code, the command prompt will take you to the next line. Now, type "java Main" to run the file:

C:\Users*Your Name*>java Main

The output should read:

Hello World

Java Syntax

Java Syntax

In the previous chapter, we created a Java file called Main.java, and we used the following code to print "Hello World" to the screen:

Main.java

```java
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello World");

  }

}
```

Example explained

Every line of code that runs in Java must be inside a class. In our example, we named the class **Main**. A class should always start with an uppercase first letter.

Note: Java is case-sensitive: "MyClass" and "myclass" has different meaning.

The name of the java file **must match** the class name. When saving the file, save it using the class name and add ".java" to the end of the filename. To run the example above on your computer, make sure that Java is properly installed: Go to the Get Started Chapter for how to install Java. The output should be:

Hello World

The main Method

The main() method is required and you will see it in every Java program:

```java
public static void main(String[] args)
```

Any code inside the main() method will be executed. Don't worry about the keywords before and after main. You will get to know them bit by bit while reading this tutorial.

For now, just remember that every Java program has a class name which must match the filename, and that every program must contain the main() method.

System.out.println()

Inside the main() method, we can use the println() method to print a line of text to the screen:

```java
public static void main(String[] args) {

  System.out.println("Hello World");

}
```

Note: The curly braces {} marks the beginning and the end of a block of code.

System is a built-in Java class that contains useful members, such as out, which is short for "output". The println() method, short for "print line", is used to print a value to the screen (or a file).

Don't worry too much about System, out and println(). Just know that you need them together to print stuff to the screen.

You should also note that each code statement must end with a semicolon (;).

Java Output / Print

Print Text

You learned from the previous chapter that you can use the println() method to output values or print text in Java:

Example :

```java
System.out.println("Hello World!");
```

You can add as many println() methods as you want. Note that it will add a new line for each method:

Example

```java
System.out.println("Hello World!");

System.out.println("I am learning Java.");
```

```java
System.out.println("It is awesome!");
```

Double Quotes

When you are working with text, it must be wrapped inside double quotations marks "".

If you forget the double quotes, an error occurs:

Example

```java
System.out.println("This sentence will work!");

System.out.println(This sentence will produce an error);
```

The Print() Method

There is also a print() method, which is similar to println().

The only difference is that it does not insert a new line at the end of the output:

Example

```java
System.out.print("Hello World! ");

System.out.print("I will print on the same line.");
```

Note that we add an extra space (after "Hello World!" in the example above), for better readability.

Java Comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line Comments

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is ignored by Java (will not be executed).

This example uses a single-line comment before a line of code:

```
// This is a comment
System.out.println("Hello World");
```

This example uses a single-line comment at the end of a line of code:

Java Multi-line Comments

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be ignored by Java.

This example uses a multi-line comment (a comment block) to explain the code:

Example

```
/* The code below will print the words Hello World
to the screen, and it is amazing */
System.out.println("Hello World");
```

Single or multi-line comments?

It is up to you which you want to use. Normally, we use // for short comments, and /* */ for longer.

Java Variables

Variables are containers for storing data values.

In Java, there are different **types** of variables, for example:

- String - stores text, such as "Hello". String values are surrounded by double quotes
- int - stores integers (whole numbers), without decimals, such as 123 or -123
- float - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- boolean - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, you must specify the type and assign it a value:

type variableName = value;

Where *type* is one of Java's types (such as int or String), and *variableName* is the name of the variable (such as **x** or **name**). The **equal sign** is used to assign values to the variable.

To create a variable that should store text, look at the following example:

Example

Create a variable called **name** of type String and assign it the value "**John**":

String name = "John";

System.out.println(name);

To create a variable that should store a number, look at the following example:

Create a variable called **myNum** of type int and assign it the value **15**:

int myNum = 15;

System.out.println(myNum);

You can also declare a variable without assigning the value, and assign the value later:

Example :

int myNum;

myNum = 15;

System.out.println(myNum);

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

Example

Change the value of myNum from 15 to 20:

```java
int myNum = 15;

myNum = 20;  // myNum is now 20

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

Final Variables

If you don't want others (or yourself) to overwrite existing values, use the final keyword (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

Example :

```java
final int myNum = 15;

myNum = 20;  // will generate an error: cannot assign a value to a final variable
```

Other Types

A demonstration of how to declare variables of other types:

Example :

```java
int myNum = 5;

float myFloatNum = 5.99f;

char myLetter = 'D';

boolean myBool = true;

String myText = "Hello";
```

You will learn more about data types in the next section.

Java Data Types

As explained in the previous chapter, a variable in Java must be a specified data type:

Example :

```java
int myNum = 5;               // Integer (whole number)
```

```java
float myFloatNum = 5.99f;    // Floating point number

char myLetter = 'D';         // Character

boolean myBool = true;       // Boolean

String myText = "Hello";     // String
```

Data types are divided into two groups:

- Primitive data types -
  includes byte, short, int, long, float, double, boolean and char
- Non-primitive data types - such as String, Arrays and Classes (you will learn more about these in a later chapter)

Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data Type	Size	Description
Byte	1 byte	Stores whole numbers from -128 to 127
Short	2 bytes	Stores whole numbers from -32,768 to 32,767
Int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
Long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
Double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
Boolean	1 bit	Stores true or false values
Char	2 bytes	Stores a single character/letter or ASCII values

Java Type Casting

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  byte -> short -> char -> int -> long -> float -> double

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  double -> float -> long -> int -> char -> short -> byte

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example :

```
public class Main {
```

```
public static void main(String[] args) {

  int myInt = 9;

  double myDouble = myInt; // Automatic casting: int to double


  System.out.println(myInt);      // Outputs 9

  System.out.println(myDouble);   // Outputs 9.0

}

}
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example :

```
public class Main {

  public static void main(String[] args) {

    double myDouble = 9.78d;

    int myInt = (int) myDouble; // Manual casting: double to int


    System.out.println(myDouble);   // Outputs 9.78

    System.out.println(myInt);      // Outputs 9

  }

}
```

Java Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + **operator** to add together two values:

```java
int x = 100 + 50;
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example :

```java
int sum1 = 100 + 50;      // 150 (100 + 50)

int sum2 = sum1 + 250;    // 400 (150 + 250)

int sum3 = sum2 + sum2;   // 800 (400 + 400)
```

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y

*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x = 10;
```

The **addition assignment** operator (+=) adds a value to a variable:

Example

```
int x = 10;

x += 5;
```

A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x – 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Java Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either true or false. These values are known as *Boolean values*, and you will learn more about them in the Booleans and If..Else chapter.

In the following example, we use the **greater than** operator (>) to find out if 5 is greater than 3:

Example

```
int x = 5;

int y = 3;

System.out.println(x > y); // returns true, because 5 is higher than 3
```

Operator	Name	Example
==	**Equal to**	**x == y**
!=	**Not equal**	**x != y**
>	**Greater than**	**x > y**
<	**Less than**	**x < y**
>=	**Greater than or equal to**	**x >= y**
<=	**Less than or equal to**	**x <= y**

Java Logical Operators

You can also test for true or false values with logical operators.

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	**Logical and**	**Returns true if both statements are true**	**x < 5 && x < 10**
||	**Logical or**	**Returns true if one of the statements is true**	**x < 5 || x < 4**

!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Java Strings

Strings are used for storing text.

A String variable contains a collection of characters surrounded by double quotes:

Example :

Create a variable of type String and assign it a value:

String greeting = "Hello";

String Length

String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the length() method:

Example

String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

System.out.println("The length of the txt string is: " + txt.length());

More String Methods

There are many string methods available, for example toUpperCase() and toLowerCase():

Example :

String txt = "Hello World";

System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"

System.out.println(txt.toLowerCase());   // Outputs "hello world"

Finding a Character in a String

The indexOf() method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

```
String txt = "Please locate where 'locate' occurs!";

System.out.println(txt.indexOf("locate")); // Outputs 7
```

Java counts positions from zero.
0 is the first position in a string, 1 is the second, 2 is the third ...

Complete String Reference

For a complete reference of String methods, go to our [Java String Methods Reference](#).

The reference contains descriptions and examples of all string methods.

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**Get your own Java Server**

```
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*:

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

Math.abs(*x*)

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

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

Random Numbers

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

```
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:

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

Complete Math Reference

For a complete reference of Math methods, go to our [Java Math Methods Reference](#).

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 9
```

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 :

```java
int myAge = 25;

int votingAge = 18;

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

Cool, right? An even better approach (since we are on a roll now), would be to wrap the code above in an if...else statement, so we can perform different actions depending on the result:

Example :

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

```java
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.");

}
```

Booleans are the basis for all Java comparisons and conditions.

You will learn more about conditions (if...else) in the next chapter.

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

}
```

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate an error.

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:

```
int x = 20;

int y = 18;

if (x > y) {

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

}
```

Example explained :

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the > operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

Syntax :

```
if (condition) {

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

} else {

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

}
```

Example :

```
int time = 20;

if (time < 18) {

  System.out.println("Good day.");

} else {

  System.out.println("Good evening.");

}
```

Example explained :

In the example above, time (20) is greater than 18, so the condition is false. Because of this, we move on to the else condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

The else if Statement

Use the else if statement to specify a new condition if the first condition is false.

Syntax :

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

Example :

```
int time = 22;
if (time < 10) {
  System.out.println("Good morning.");
} else if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

Example explained :

In the example above, time (22) is greater than 10, so the **first condition** is false. The next condition, in the else if statement, is also false, so we move on to the else condition since **condition1** and **condition2** is both false - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

Java Switch

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:

Syntax

```
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) {
```

```java
  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.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

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"
```

Note that if the default statement is used as the last statement in a switch block, it does not need a break

Java While

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:

```
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++;

}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax :

```
do {

  // code block to be executed

}
while (condition);
```

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

```java
int i = 0;
do {

  System.out.println(i);

  i++;

}

while (i < 5);
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

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 :

```java
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 :

```java
for (int i = 0; i < 5; i++) {

  System.out.println(i);

}
```

Example explained :

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Another Example :

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

Example :

```java
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 :

```java
// 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)
  }
}
```

Java Break and Continue

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:

```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 :

```java
int i = 0;

while (i < 10) {

  System.out.println(i);

  i++;

  if (i == 4) {
```

```
    break;

  }

}
```

```
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:

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

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

// Outputs Volvo
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

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

Example :

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

Example :

```java
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 :

```java
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:

```java
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!"
```

A method can also be called multiple times:

Example :

```java
public class Main {

  static void myMethod() {

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

  }


  public static void main(String[] args) {

    myMethod();

    myMethod();

    myMethod();

  }

}


// I just got executed!
```

In the next chapter, [Method Parameters](#), you will learn how to pass data (parameters) into a method.

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 :

```java
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
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: fname is a **parameter**, while Liam, Jenny and Anja are **arguments**.

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
```

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

Example :

```java
public class Main {

  static int myMethod(int x) {

    return 5 + x;

  }


  public static void main(String[] args) {

    System.out.println(myMethod(3));

  }

}
// Outputs 8 (5 + 3)
```

This example returns the sum of a method's **two parameters**:

Example :

```java
public class Main {

  static int myMethod(int x, int y) {

    return x + y;

  }


  public static void main(String[] args) {

    System.out.println(myMethod(5, 3));

  }

}
// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example :

```java
public class Main {
```

```java
  static int myMethod(int x, int y) {

    return x + y;

  }


  public static void main(String[] args) {

    int z = myMethod(5, 3);

    System.out.println(z);

  }

}
// Outputs 8 (5 + 3)
```

A Method with If...Else

It is common to use if...else statements inside methods:

Example :

```java
public class Main {


  // Create a checkAge() method with an integer variable called age
  static void checkAge(int age) {


    // If age is less than 18, print "access denied"
    if (age < 18) {

      System.out.println("Access denied - You are not old enough!");


    // If age is greater than, or equal to, 18, print "access granted"
    } else {

      System.out.println("Access granted - You are old enough!");

    }
```

```
  }


  public static void main(String[] args) {

    checkAge(20); // Call the checkAge method and pass along an age of 20

  }

}



// Outputs "Access granted - You are old enough!"
```

Java Method Overloading

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;

}
```

```java
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 :

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

  return x + y;

}


static double plusMethod(double x, double y) {

  return x + y;

}


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);

}
```

Note: Multiple methods can have the same name as long as the number and/or type of parameters are different.

Java Scope

In Java, variables are only accessible inside the region they are created. This is called **scope**.

Method Scope

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared:

Example :

```java
public class Main {
  public static void main(String[] args) {

    // Code here CANNOT use x

    int x = 100;

    // Code here can use x
    System.out.println(x);
  }
}
```

Block Scope

A block of code refers to all of the code between curly braces {}.

Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared:

Example

```java
public class Main {
  public static void main(String[] args) {

    // Code here CANNOT use x
```

```
{ // This is a block


  // Code here CANNOT use x


  int x = 100;


  // Code here CAN use x
  System.out.println(x);


} // The block ends here


// Code here CANNOT use x


}
}
```

A block of code may exist on its own or it can belong to an if, while or for statement. In the case of for statements, variables declared in the statement itself are also available inside the block's scope.

Java Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

Use recursion to add all of the numbers up to 10.

```java
public class Main {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
    }
  }
}
```

Example Explained :

When the sum() function is called, it adds parameter k to the sum of all numbers smaller than k and returns the result. When k becomes 0, the function just returns 0. When running, the program follows these steps:

Halting Condition

Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself. In the previous example, the halting condition is when the parameter k becomes 0.

It is helpful to see a variety of different examples to better understand the concept. In this example, the function adds a range of numbers between a start and an end. The halting condition for this recursive function is when **end** is not greater than **start**:

Example :

Use recursion to add all of the numbers between 5 to 10.

```java
public class Main {

  public static void main(String[] args) {

    int result = sum(5, 10);

    System.out.println(result);

  }

  public static int sum(int start, int end) {

    if (end > start) {

      return end + sum(start, end - 1);

    } else {

      return end;

    }

  }

}
```

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

Java OOP

Java - What is OOP?

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.
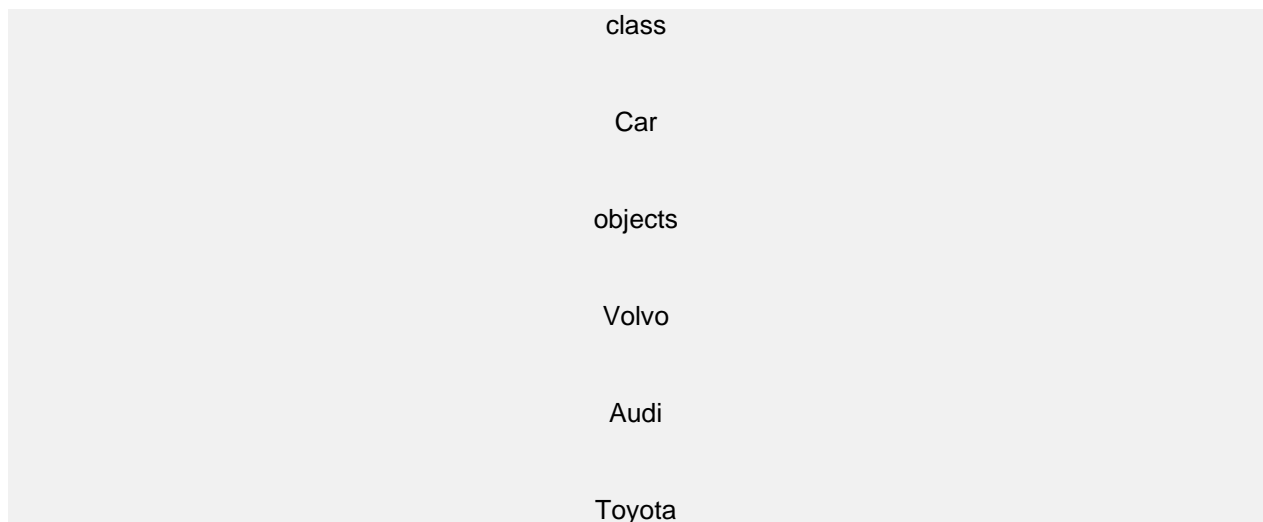
Java - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

class

Fruit

objects

Apple

Banana

Mango

Another example:

class

Car

objects

Volvo

Audi

Toyota
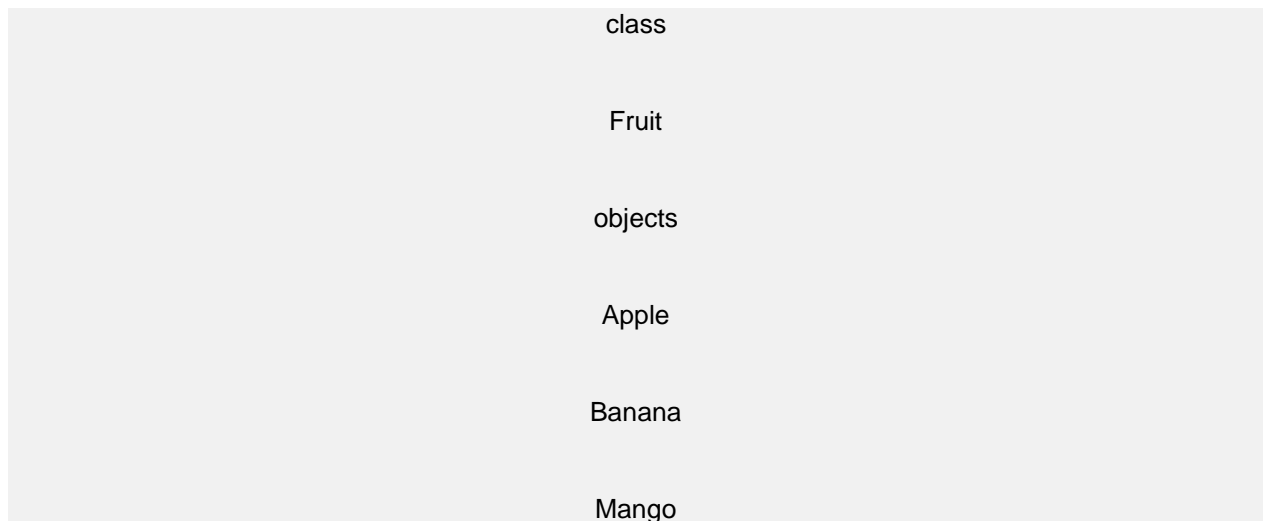
So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

You will learn much more about [classes and objects](#) in the next chapter.

Java Classes and Objects

Java Classes/Objects

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Main.java :

Create a class named "Main" with a variable x:

```
public class Main {

  int x = 5;

}
```

Remember from the [Java Syntax chapter](#) that a class should always start with an uppercase first letter, and that the name of the java file should match the class name.

Create an Object

In Java, an object is created from a class. We have already created the class named Main, so now we can use this to create objects.

To create an object of Main, specify the class name, followed by the object name, and use the keyword new:

Multiple Objects

You can create multiple objects of one class:

```java
public class Main {
  int x = 5;


  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);

  }

}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

- Main.java
- Second.java

Main.java :

```java
public class Main {
  int x = 5;

}
```

Second.java :

```java
class Second {
  public static void main(String[] args) {
    Main myObj = new Main();
```

```
    System.out.println(myObj.x);

  }

}
```

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
5
```

You will learn much more about classes and objects in the next chapters.

Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Example :

Create a constructor:

```
// Create a Main class

public class Main {

  int x;  // Create a class attribute


  // Create a class constructor for the Main class

  public Main() {

    x = 5;  // Set the initial value for the class attribute x

  }


  public static void main(String[] args) {
```

```
    Main myObj = new Main(); // Create an object of class Main (This will call the constructor)

    System.out.println(myObj.x); // Print the value of x

  }

}


// Outputs 5
```

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

Example :

```
public class Main {

  int x;


  public Main(int y) {

    x = y;

  }


  public static void main(String[] args) {

    Main myObj = new Main(5);

    System.out.println(myObj.x);
```

```
  }

}


// Outputs 5
```

You can have as many parameters as you want:

Example :

```java
public class Main {

  int modelYear;

  String modelName;


  public Main(int year, String name) {

    modelYear = year;

    modelName = name;

  }


  public static void main(String[] args) {

    Main myCar = new Main(1969, "Mustang");

    System.out.println(myCar.modelYear + " " + myCar.modelName);

  }

}


// Outputs 1969 Mustang
```

Java Modifiers

By now, you are quite familiar with the public keyword that appears in almost all of our examples:

```
public class Main
```

The public keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- **Access Modifiers** - controls the access level
- **Non-Access Modifiers** - do not control access level, but provides other functionality

Access Modifiers

For **classes**, you can use either public or *default*:

Modifier	Description
Public	The class is accessible by any other class
Default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
Public	The code is accessible for all classes

Private	The code is only accessible within the declared class
Default	The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the [Packages chapter](#)
Protected	The code is accessible in the same package and **subclasses**. You will learn more about subclasses and superclasses in the [Inheritance chapter](#)

Non-Access Modifiers

For **classes**, you can use either final or abstract:

Modifier	Description
final	The class cannot be inherited by other classes (You will learn more about inheritance in the [Inheritance chapter](#))
abstract	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the [Inheritance](#) and [Abstraction](#) chapters)

For **attributes and methods**, you can use the one of the following:

Modifier	Description

Final	Attributes and methods cannot be overridden/modified
static	Attributes and methods belongs to the class, rather than an object
abstract	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters
transient	Attributes and methods are skipped when serializing the object containing them
synchronized	Methods can only be accessed by one thread at a time
volatile	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Final

If you don't want the ability to override existing attribute values, declare attributes as final:

Example :

```
public class Main {

  final int x = 10;
```

```java
  final double PI = 3.14;


  public static void main(String[] args) {

    Main myObj = new Main();

    myObj.x = 50; // will generate an error: cannot assign a value to a final variable

    myObj.PI = 25; // will generate an error: cannot assign a value to a final variable

    System.out.println(myObj.x);

  }

}
```

Static

A static method means that it can be accessed without creating an object of the class, unlike public:

Example :

An example to demonstrate the differences between static and public methods:

```java
public class Main {

 // Static method

 static void myStaticMethod() {

   System.out.println("Static methods can be called without creating objects");

 }


 // Public method

 public void myPublicMethod() {

   System.out.println("Public methods must be called by creating objects");

 }


 // Main method

 public static void main(String[ ] args) {
```

```
    myStaticMethod(); // Call the static method

    // myPublicMethod(); This would output an error


    Main myObj = new Main(); // Create an object of Main

    myObj.myPublicMethod(); // Call the public method

  }

}
```

Abstract

An abstract method belongs to an abstract class, and it does not have a body. The body is provided by the subclass:

Example :

```
// Code from filename: Main.java

// abstract class
abstract class Main {

  public String fname = "John";

  public int age = 24;

  public abstract void study(); // abstract method

}


// Subclass (inherit from Main)

class Student extends Main {

  public int graduationYear = 2018;

  public void study() { // the body of the abstract method is provided here

    System.out.println("Studying all day long");

  }

}

// End code from filename: Main.java
```

```
// Code from filename: Second.java

class Second {

 public static void main(String[] args) {

   // create an object of the Student class (which inherits attributes and methods from Main)

   Student myObj = new Student();


   System.out.println("Name: " + myObj.fname);

   System.out.println("Age: " + myObj.age);

   System.out.println("Graduation Year: " + myObj.graduationYear);

   myObj.study(); // call abstract method
 }

}
```

Java Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as private
- provide public **get** and **set** methods to access and update the value of a private variable

Get and Set

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods.

The get method returns the variable value, and the set method sets the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

Example :

```java
public class Person {
  private String name; // private = restricted access


  // Getter
  public String getName() {
    return name;
  }


  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

Example explained

The get method returns the value of the variable name.

The set method takes a parameter (newName) and assigns it to the name variable. The this keyword is used to refer to the current object.

However, as the name variable is declared as private, we **cannot** access it from outside this class:

Example :

```java
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.name = "John";  // error
    System.out.println(myObj.name); // error
  }
}
```

If the variable was declared as public, we would expect the following output:

John

However, as we try to access a private variable, we get an error:

```
MyClass.java:4: error: name has private access in Person
    myObj.name = "John";
         ^
MyClass.java:5: error: name has private access in Person
    System.out.println(myObj.name);
                 ^
2 errors
```

Instead, we use the getName() and setName() methods to access and update the variable:

Example :

```java
public class Main {

  public static void main(String[] args) {

    Person myObj = new Person();

    myObj.setName("John"); // Set the value of the name variable to "John"

    System.out.println(myObj.getName());

  }

}


// Outputs "John"
```

Why Encapsulation?

- Better control of class attributes and methods
- Class attributes can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

Java Packages

Java Packages & API

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

Syntax :

import *package*.*name*.*Class*;   // Import a single class

import *package*.*name*.*;   // Import the whole package

Import a Class

If you find a class you want to use, for example, the Scanner class, **which is used to get user input**, write the following code:

Example :

import java.util.Scanner;

In the example above, java.util is a package, while Scanner is a class of the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read a complete line:

Example :

Using the Scanner class to get user input:

```java
import java.util.Scanner;


class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");


    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);

  }

}
```

Import a Package

There are many packages to choose from. In the previous example, we used the Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the java.util package:

Example :

```java
import java.util.*;
```

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example :
```
└── root
  └── mypack
    └── MyPackageClass.java
```

To create a package, use the package keyword:

MyPackageClass.java

```java
package mypack;

class MyPackageClass {

  public static void main(String[] args) {

    System.out.println("This is my package!");

  }

}
```

Save the file as **MyPackageClass.java**, and compile it:

C:\Users*Your Name*>javac MyPackageClass.java

Then compile the package:

C:\Users*Your Name*>javac -d . MyPackageClass.java

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

C:\Users*Your Name*>java mypack.MyPackageClass

The output will be:

This is my package!

Java Inheritance

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

Example :

```java
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                     // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}


class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {


    // Create a myCar object
    Car myCar = new Car();


    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();
```

```
    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName
from the Car class

    System.out.println(myCar.brand + " " + myCar.modelName);

  }

}
```

Did you notice the protected modifier in Vehicle?

We set the **brand** attribute in **Vehicle** to a protected access modifier. If it was set to private, the Car class would not be able to access it.

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The final Keyword

If you don't want other classes to inherit from a class, use the final keyword:

If you try to access a final class, Java will generate an error:

```
final class Vehicle {

  ...

}




class Car extends Vehicle {

  ...

}
```

The output will be something like this:

Main.java:9: error: cannot inherit from final Vehicle
class Main extends Vehicle {
            ^
1 error)

Java Inheritance

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

Example :

```java
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}


class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {


    // Create a myCar object
    Car myCar = new Car();


    // Call the honk() method (from the Vehicle class) on the myCar object
```

```
    myCar.honk();


    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName
from the Car class

    System.out.println(myCar.brand + " " + myCar.modelName);

  }

}
```

Did you notice the protected modifier in Vehicle?

We set the **brand** attribute in **Vehicle** to a protected access modifier. If it was set to private, the Car class would not be able to access it.

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The final Keyword

If you don't want other classes to inherit from a class, use the final keyword:

If you try to access a final class, Java will generate an error:

```
final class Vehicle {

  ...

}


class Car extends Vehicle {

  ...

}
```

The output will be something like this:

```
Main.java:9: error: cannot inherit from final Vehicle
class Main extends Vehicle {
          ^
1 error)
```

Java Inner Classes

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

Example :

```java
class OuterClass {
  int x = 10;


  class InnerClass {
    int y = 5;

  }

}


public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);

  }

}


// Outputs 15 (5 + 10)
```

Private Inner Class

Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

```java
class OuterClass {
  int x = 10;


  private class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

If you try to access a private inner class from an outside class, an error occurs:

```
Main.java:13: error: OuterClass.InnerClass has private access in OuterClass
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
```

Static Inner Class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

Example :

```java
class OuterClass {
  int x = 10;


  static class InnerClass {
```

```
    int y = 5;

  }

}

public class Main {

  public static void main(String[] args) {

    OuterClass.InnerClass myInner = new OuterClass.InnerClass();

    System.out.println(myInner.y);

  }

}
// Outputs 5
```

Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

Example :

```
class OuterClass {

  int x = 10;


  class InnerClass {

    public int myInnerMethod() {

      return x;

    }

  }

}
public class Main {

  public static void main(String[] args) {
```

```
  OuterClass myOuter = new OuterClass();

  OuterClass.InnerClass myInner = myOuter.new InnerClass();

  System.out.println(myInner.myInnerMethod());

 }

}


// Outputs 10
```

Java Abstraction

Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The abstract keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {

 public abstract void animalSound();

 public void sleep() {

   System.out.println("Zzz");

 }

}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class:

Remember from the Inheritance chapter that we use the extends keyword to inherit from a class.

Example :

```java
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
```

```
}
```

Java Interface

Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An interface is a completely "**abstract class**" that is used to group related methods with empty bodies:

Example :

```
// interface

interface Animal {

  public void animalSound(); // interface method (does not have a body)

  public void run(); // interface method (does not have a body)

}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class:

Example :

```
// Interface

interface Animal {

  public void animalSound(); // interface method (does not have a body)

  public void sleep(); // interface method (does not have a body)

}
```

```java
// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}


class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example :

```java
interface FirstInterface {
  public void myMethod(); // interface method
}


interface SecondInterface {
  public void myOtherMethod(); // interface method
}


class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}


class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
```

```
  myObj.myOtherMethod();

 }

}
```

Java Enums

An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

Example :

```
enum Level {

  LOW,

  MEDIUM,

  HIGH

}
```

You can access enum constants with the **dot** syntax:

Level myVar = Level.MEDIUM;

Enum is short for "enumerations", which means "specifically listed".

Enum inside a Class

You can also have an enum inside a class:

Example :

```
public class Main {

 enum Level {

  LOW,

  MEDIUM,

  HIGH

 }
```

```
public static void main(String[] args) {

  Level myVar = Level.MEDIUM;

  System.out.println(myVar);

 }

}
```

The output will be:

MEDIUM

Enum in a Switch Statement

Enums are often used in switch statements to check for corresponding values:

Example :

```
enum Level {

  LOW,

  MEDIUM,

  HIGH

}


public class Main {
 public static void main(String[] args) {

  Level myVar = Level.MEDIUM;


  switch(myVar) {

   case LOW:

    System.out.println("Low level");

    break;

   case MEDIUM:
```

```
      System.out.println("Medium level");

    break;

  case HIGH:

    System.out.println("High level");

    break;

  }

 }

}
```

The output will be:

Medium level

Loop Through an Enum

The enum type has a `values()` method, which returns an array of all enum constants. This method is useful when you want to loop through the constants of an enum:

Example :

```
for (Level myVar : Level.values()) {

  System.out.println(myVar);

}
```

The output will be:

LOW
MEDIUM
HIGH

Difference between Enums and Classes

An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).

An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Why And When To Use Enums?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

Java User Input (Scanner)

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read Strings:

Example :

```java
import java.util.Scanner;  // Import the Scanner class


class Main {

  public static void main(String[] args) {

    Scanner myObj = new Scanner(System.in);  // Create a Scanner object

    System.out.println("Enter username");


    String userName = myObj.nextLine();  // Read user input

    System.out.println("Username is: " + userName); // Output user input

  }

}
```

Input Types

In the example above, we used the nextLine() method, which is used to read Strings. To read other types, look at the table below:

In the example below, we use different methods to read data of various types:

Method	Description
nextBoolean()	Reads a boolean value from the user

nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

Example :

```java
import java.util.Scanner;


class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);


    System.out.println("Enter name, age and salary:");


    // String input
    String name = myObj.nextLine();


    // Numerical input
```

```
    int age = myObj.nextInt();

    double salary = myObj.nextDouble();


    // Output input by user

    System.out.println("Name: " + name);

    System.out.println("Age: " + age);

    System.out.println("Salary: " + salary);

  }

}
```

Java Date and Time

Java Dates

Java does not have a built-in Date class, but we can import the java.time package to work with the date and time API. The package includes many date and time classes. For example:

Class	Description
LocalDate	Represents a date (year, month, day (yyyy-MM-dd))
LocalTime	Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
LocalDateTime	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)

DateTimeFormatter	Formatter for displaying and parsing date-time objects

Display Current Date

To display the current date, import the java.time.LocalDate class, and use its now() method:

Example :

```
import java.time.LocalDate; // import the LocalDate class


public class Main {

  public static void main(String[] args) {

    LocalDate myObj = LocalDate.now(); // Create a date object

    System.out.println(myObj); // Display the current date

  }

}
```

The output will be:

2023-09-09

Display Current Time

To display the current time (hour, minute, second, and nanoseconds), import the java.time.LocalTime class, and use its now() method:

Example :

```
import java.time.LocalTime; // import the LocalTime class


public class Main {

  public static void main(String[] args) {

    LocalTime myObj = LocalTime.now();

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

```
  }
}
```

The output will be:

17:33:26.631446

Display Current Date and Time

To display the current date and time, import the java.time.LocalDateTime class, and use its now() method:

Example :

```java
import java.time.LocalDateTime; // import the LocalDateTime class


public class Main {

  public static void main(String[] args) {

    LocalDateTime myObj = LocalDateTime.now();

    System.out.println(myObj);

  }

}
```

The output will be:

2023-09-09T17:33:26.632547

Formatting Date and Time

The "T" in the example above is used to separate the date from the time. You can use the DateTimeFormatter class with the ofPattern() method in the same package to format or parse date-time objects. The following example will remove both the "T" and nanoseconds from the date-time:

Example :

```java
import java.time.LocalDateTime; // Import the LocalDateTime class

import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter class
```

```java
public class Main {

  public static void main(String[] args) {

    LocalDateTime myDateObj = LocalDateTime.now();

    System.out.println("Before formatting: " + myDateObj);

    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");


    String formattedDate = myDateObj.format(myFormatObj);

    System.out.println("After formatting: " + formattedDate);

  }

}
```

The output will be:

Before Formatting: 2023-09-09T17:33:26.632027
After Formatting: 09-09-2023 17:33:26

The ofPattern() method accepts all sorts of values, if you want to display the date and time in a different format. For example:

Value	Example
yyyy-MM-dd	"1988-09-29"
dd/MM/yyyy	"29/09/1988"
dd-MMM-yyyy	"29-Sep-1988"
E, MMM dd yyyy	"Thu, Sep 29 1988"

Java ArrayList

The ArrayList class is a resizable [array](#), which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want. The syntax is also slightly different:

Example :

Create an ArrayList object called **cars** that will store strings:

```
import java.util.ArrayList; // import the ArrayList class


ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

If you don't know what a package is, read our [Java Packages Tutorial](#).

Add Items

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

Example :

```
import java.util.ArrayList;


public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Access an Item

To access an element in the ArrayList, use the get() method and refer to the index number:

```
cars.get(0);
```

Remember: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Item

To modify an element, use the set() method and refer to the index number:

Example :

```
cars.set(0, "Opel");
```

Remove an Item

To remove an element, use the remove() method and refer to the index number:

Example :

```
cars.remove(0);
```

To remove all the elements in the ArrayList, use the clear() method:

Example :

```
cars.clear();
```

ArrayList Size

To find out how many elements an ArrayList have, use the size method:

Example :

```
cars.size();
```

Loop Through an ArrayList

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

Example :

```
public class Main {

  public static void main(String[] args) {

    ArrayList<String> cars = new ArrayList<String>();

    cars.add("Volvo");
```

```
    cars.add("BMW");

    cars.add("Ford");

    cars.add("Mazda");

    for (int i = 0; i < cars.size(); i++) {

      System.out.println(cars.get(i));

    }

  }

}
```

You can also loop through an ArrayList with the **for-each** loop:

Example :

```
public class Main {

  public static void main(String[] args) {

    ArrayList<String> cars = new ArrayList<String>();

    cars.add("Volvo");

    cars.add("BMW");

    cars.add("Ford");

    cars.add("Mazda");

    for (String i : cars) {

      System.out.println(i);

    }

  }

}
```

Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

Example :

Create an ArrayList to store numbers (add elements of type Integer):

```java
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

Sort an ArrayList

Another useful class in the java.util package is the Collections class, which include the sort() method for sorting lists alphabetically or numerically:

Example :

Sort an ArrayList of Strings:

```java
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
```

```java
    cars.add("BMW");

    cars.add("Ford");

    cars.add("Mazda");

    Collections.sort(cars);  // Sort cars

    for (String i : cars) {

      System.out.println(i);

    }

  }

}
```

Example :

Sort an ArrayList of Integers:

```java
import java.util.ArrayList;

import java.util.Collections;  // Import the Collections class


public class Main {

  public static void main(String[] args) {

    ArrayList<Integer> myNumbers = new ArrayList<Integer>();

    myNumbers.add(33);

    myNumbers.add(15);

    myNumbers.add(20);

    myNumbers.add(34);

    myNumbers.add(8);

    myNumbers.add(12);


    Collections.sort(myNumbers);  // Sort myNumbers


    for (int i : myNumbers) {

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

```
    }

  }

}
```

Java LinkedList

In the previous chapter, you learned about the ArrayList class. The LinkedList class is almost identical to the ArrayList:

Example :

```java
// Import the LinkedList class

import java.util.LinkedList;


public class Main {

  public static void main(String[] args) {

    LinkedList<String> cars = new LinkedList<String>();

    cars.add("Volvo");

    cars.add("BMW");

    cars.add("Ford");

    cars.add("Mazda");

    System.out.println(cars);

  }

}
```

ArrayList vs. LinkedList

The LinkedList class is a collection which can contain many objects of the same type, just like the ArrayList.

The LinkedList class has all of the same methods as the ArrayList class because they both implement the List interface. This means that you can add items, change items, remove items and clear the list in the same way.

However, while the ArrayList class and the LinkedList class can be used in the same way, they are built very differently.

How the ArrayList works

The ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

How the LinkedList works

The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

When To Use

Use an ArrayList for storing and accessing data, and LinkedList to manipulate data.

LinkedList Methods

For many cases, the ArrayList is more efficient as it is common to need access to random items in the list, but the LinkedList provides several methods to do certain operations more efficiently:

Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list

getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

Java HashMap

In the ArrayList chapter, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (int type). A HashMap however, store items in "**key/value**" pairs, and you can access them by an index of another type (e.g. a String).

One object is used as a key (index) to another object (value). It can store different types: String keys and Integer values, or the same type, like: String keys and String values:

Example :

Create a HashMap object called **capitalCities** that will store String **keys** and String **values**:

```
import java.util.HashMap; // import the HashMap class


HashMap<String, String> capitalCities = new HashMap<String, String>();
```

Add Items

The HashMap class has many useful methods. For example, to add items to it, use the put() method:

Example :

```
// Import the HashMap class

import java.util.HashMap;


public class Main {

  public static void main(String[] args) {

    // Create a HashMap object called capitalCities

    HashMap<String, String> capitalCities = new HashMap<String, String>();
```

```
  // Add keys and values (Country, City)

  capitalCities.put("England", "London");

  capitalCities.put("Germany", "Berlin");

  capitalCities.put("Norway", "Oslo");

  capitalCities.put("USA", "Washington DC");

  System.out.println(capitalCities);

  }

}
```

Access an Item

To access a value in the HashMap, use the get() method and refer to its key:

Example :

```
capitalCities.get("England");
```

Remove an Item

To remove an item, use the remove() method and refer to the key:

Example :

```
capitalCities.remove("England");
```

To remove all items, use the clear() method:

Example :

```
capitalCities.clear();
```

HashMap Size

To find out how many items there are, use the size() method:

Example :

```
capitalCities.size();
```

Loop Through a HashMap

Loop through the items of a HashMap with a **for-each** loop.

Note: Use the keySet() method if you only want the keys, and use the values() method if you only want the values:

Example :

```
// Print keys

for (String i : capitalCities.keySet()) {

  System.out.println(i);

}
```

Example :

```
// Print values

for (String i : capitalCities.values()) {

  System.out.println(i);

}
```

Example :

```
// Print keys and values

for (String i : capitalCities.keySet()) {

  System.out.println("key: " + i + " value: " + capitalCities.get(i));

}
```

Other Types

Keys and values in a HashMap are actually objects. In the examples above, we used objects of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

Example :

Create a HashMap object called **people** that will store String **keys** and Integer **values**:

```
// Import the HashMap class

import java.util.HashMap;


public class Main {

  public static void main(String[] args) {
```

```java
// Create a HashMap object called people
HashMap<String, Integer> people = new HashMap<String, Integer>();



// Add keys and values (Name, Age)
people.put("John", 32);
people.put("Steve", 30);
people.put("Angie", 33);


for (String i : people.keySet()) {
  System.out.println("key: " + i + " value: " + people.get(i));

 }

}
```

Java HashSet

A HashSet is a collection of items where every item is unique, and it is found in the java.util package:

Example :

Create a HashSet object called **cars** that will store strings:

```java
import java.util.HashSet; // Import the HashSet class


HashSet<String> cars = new HashSet<String>();
```

Add Items

The HashSet class has many useful methods. For example, to add items to it, use the add() method:

```java
// Import the HashSet class

import java.util.HashSet;


public class Main {

  public static void main(String[] args) {

    HashSet<String> cars = new HashSet<String>();

    cars.add("Volvo");

    cars.add("BMW");

    cars.add("Ford");

    cars.add("BMW");

    cars.add("Mazda");

    System.out.println(cars);

  }

}
```

Note: In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

Check If an Item Exists

To check whether an item exists in a HashSet, use the contains() method:

Example :

```java
cars.contains("Mazda");
```

Remove an Item

To remove an item, use the remove() method:

Example :

```java
cars.remove("Volvo");
```

To remove all items, use the clear() method:

```
cars.clear();
```

HashSet Size

To find out how many items there are, use the size method:

```
cars.size();
```

Loop Through a HashSet

Loop through the items of an HashSet with a **for-each** loop:

```
for (String i : cars) {

  System.out.println(i);

}
```

Other Types

Items in an HashSet are actually objects. In the examples above, we created items (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

Use a HashSet that stores Integer objects:

```
import java.util.HashSet;



public class Main {

  public static void main(String[] args) {


    // Create a HashSet object called numbers

    HashSet<Integer> numbers = new HashSet<Integer>();



    // Add values to the set
```

```java
    numbers.add(4);

    numbers.add(7);

    numbers.add(8);


    // Show which numbers between 1 and 10 are in the set

    for(int i = 1; i <= 10; i++) {

      if(numbers.contains(i)) {

        System.out.println(i + " was found in the set.");

      } else {

        System.out.println(i + " was not found in the set.");

      }

    }

  }

}
```

Java Iterator

An Iterator is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the java.util package.

Getting an Iterator

The iterator() method can be used to get an Iterator for any collection:

Example :

```java
// Import the ArrayList class and the Iterator class

import java.util.ArrayList;

import java.util.Iterator;
```

```
public class Main {

  public static void main(String[] args) {


    // Make a collection

    ArrayList<String> cars = new ArrayList<String>();

    cars.add("Volvo");

    cars.add("BMW");

    cars.add("Ford");

    cars.add("Mazda");


    // Get the iterator

    Iterator<String> it = cars.iterator();


    // Print the first item

    System.out.println(it.next());

  }

}
```

Looping Through a Collection

To loop through a collection, use the hasNext() and next() methods of the Iterator:

Example :

```
while(it.hasNext()) {

  System.out.println(it.next());

}
```

Removing Items from a Collection

Iterators are designed to easily change the collections that they loop through. The remove() method can remove items from a collection while looping.

Example :

Use an iterator to remove numbers less than 10 from a collection:

```java
import java.util.ArrayList;

import java.util.Iterator;



public class Main {

  public static void main(String[] args) {

    ArrayList<Integer> numbers = new ArrayList<Integer>();

    numbers.add(12);

    numbers.add(8);

    numbers.add(2);

    numbers.add(23);

    Iterator<Integer> it = numbers.iterator();

    while(it.hasNext()) {

      Integer i = it.next();

      if(i < 10) {

        it.remove();

      }

    }

    System.out.println(numbers);

  }

}
```

Note: Trying to remove items using a **for loop** or a **for-each loop** would not work correctly because the collection is changing size at the same time that the code is trying to loop.

Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects):

Example :

```
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid

ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

Creating Wrapper Objects

To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

Example :

```java
public class Main {

  public static void main(String[] args) {

    Integer myInt = 5;

    Double myDouble = 5.99;

    Character myChar = 'A';

    System.out.println(myInt);

    System.out.println(myDouble);

    System.out.println(myChar);

  }

}
```

Since you're now working with objects, you can use certain methods to get information about the specific object.

For example, the following methods are used to get the value associated with the corresponding wrapper object: intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

This example will output the same result as the example above:

Example :

```java
public class Main {

  public static void main(String[] args) {

    Integer myInt = 5;

    Double myDouble = 5.99;

    Character myChar = 'A';

    System.out.println(myInt.intValue());

    System.out.println(myDouble.doubleValue());

    System.out.println(myChar.charValue());

  }

}
```

Another useful method is the toString() method, which is used to convert wrapper objects to strings.

In the following example, we convert an Integer to a String, and use the length() method of the String class to output the length of the "string":

```java
public class Main {

  public static void main(String[] args) {

    Integer myInt = 100;

    String myString = myInt.toString();

    System.out.println(myString.length());

  }

}
```

Java Exceptions - Try...Catch

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

Syntax :

```java
try {

  // Block of code to try

}
```

```java
catch(Exception e) {

  // Block of code to handle errors

}
```

Consider the following example:

This will generate an error, because **myNumbers[10]** does not exist.

```java
public class Main {

  public static void main(String[ ] args) {

    int[] myNumbers = {1, 2, 3};

    System.out.println(myNumbers[10]); // error!

  }

}
```

The output will be something like this:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
      at Main.main(Main.java:4)

If an error occurs, we can use try...catch to catch the error and execute some code to handle it:

Example :

```java
public class Main {

  public static void main(String[ ] args) {

    try {

      int[] myNumbers = {1, 2, 3};

      System.out.println(myNumbers[10]);

    } catch (Exception e) {

      System.out.println("Something went wrong.");

    }

  }

}
```

The output will be:

Something went wrong.

Finally

The finally statement lets you execute code, after try...catch, regardless of the result:

Example :

```java
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

The output will be:

Something went wrong.
The 'try catch' is finished.

The throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

Example :

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```java
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
```

```java
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");

  }

  else {

    System.out.println("Access granted - You are old enough!");

  }

}


public static void main(String[] args) {

  checkAge(15); // Set age to 15 (which is below 18...)

  }

}
```

The output will be:

Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.
    at Main.checkAge(Main.java:4)
    at Main.main(Main.java:12)

If **age** was 20, you would **not** get an exception:

Example :

```java
checkAge(20);
```

The output will be:

Access granted - You are old enough!

Java Regular Expressions

What is a Regular Expression?

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Java does not have a built-in Regular Expression class, but we can import the java.util.regex package to work with regular expressions. The package includes the following classes:

- Pattern Class - Defines a pattern (to be used in a search)
- Matcher Class - Used to search for the pattern
- PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

Example :

Find out if there are any occurrences of the word "w3schools" in a sentence:

```java
import java.util.regex.Matcher;

import java.util.regex.Pattern;


public class Main {

  public static void main(String[] args) {

    Pattern pattern = Pattern.compile("w3schools", Pattern.CASE_INSENSITIVE);

    Matcher matcher = pattern.matcher("Visit W3Schools!");

    boolean matchFound = matcher.find();

    if(matchFound) {

      System.out.println("Match found");

    } else {

      System.out.println("Match not found");

    }

  }

}
// Outputs Match found
```

Example Explained :

In this example, The word "w3schools" is being searched for in a sentence.

First, the pattern is created using the Pattern.compile() method. The first parameter indicates which pattern is being searched for and the second parameter has a flag to indicates that the search should be case-insensitive. The second parameter is optional.

The matcher() method is used to search for the pattern in a string. It returns a Matcher object which contains information about the search that was performed.

The find() method returns true if the pattern was found in the string and false if it was not found.

Flags

Flags in the compile() method change how the search is performed. Here are a few of them:

- Pattern.CASE_INSENSITIVE - The case of letters will be ignored when performing a search.
- Pattern.LITERAL - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.
- Pattern.UNICODE_CASE - Use it together with the CASE_INSENSITIVE flag to also ignore the case of letters outside of the English alphabet

Regular Expression Patterns

The first parameter of the Pattern.compile() method is the pattern. It describes what is being searched for.

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find one character from the options between the brackets
[^abc]	Find one character NOT between the brackets
[0-9]	Find one character from the range 0 to 9

Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
|	Find a match for any one of the patterns separated by | as in: cat|dog|fish
.	Find just one instance of any character
^	Finds a match as the beginning of a string as in: ^Hello
$	Finds a match at the end of the string as in: World$
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers

Quantifiers define quantities:

Quantifier	Description
n+	Matches any string that contains at least one *n*

n*	Matches any string that contains zero or more occurrences of *n*
n?	Matches any string that contains zero or one occurrences of *n*
n{x}	Matches any string that contains a sequence of *X* *n*'s
n{x,y}	Matches any string that contains a sequence of X to Y *n*'s
n{x,}	Matches any string that contains a sequence of at least X *n*'s

Java Threads

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

Creating a Thread

There are two ways to create a thread.

It can be created by extending the Thread class and overriding its run() method:

Extend Syntax :

```java
public class Main extends Thread {

  public void run() {

    System.out.println("This code is running in a thread");

  }
```

```
}
```

Another way to create a thread is to implement the Runnable interface:

Implement Syntax :

```java
public class Main implements Runnable {

  public void run() {

    System.out.println("This code is running in a thread");

  }

}
```

Running Threads

If the class extends the Thread class, the thread can be run by creating an instance of the class and call its start() method:

Extend Example :

```java
public class Main extends Thread {

  public static void main(String[] args) {

    Main thread = new Main();

    thread.start();

    System.out.println("This code is outside of the thread");

  }

  public void run() {

    System.out.println("This code is running in a thread");

  }

}
```

If the class implements the Runnable interface, the thread can be run by passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:

Implement Example :

```java
public class Main implements Runnable {

  public static void main(String[] args) {
```

```
    Main obj = new Main();

    Thread thread = new Thread(obj);

    thread.start();

    System.out.println("This code is outside of the thread");

  }

  public void run() {

    System.out.println("This code is running in a thread");

  }

}
```

Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

Example

A code example where the value of the variable **amount** is unpredictable:

```
public class Main extends Thread {

  public static int amount = 0;


  public static void main(String[] args) {

    Main thread = new Main();

    thread.start();

    System.out.println(amount);

    amount++;

    System.out.println(amount);

  }
```

```java
  public void run() {

    amount++;

  }

}
```

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the isAlive() method of the thread to check whether the thread has finished running before using any attributes that the thread can change.

Example

Use isAlive() to prevent concurrency problems:

```java
public class Main extends Thread {

  public static int amount = 0;


  public static void main(String[] args) {

    Main thread = new Main();

    thread.start();

    // Wait for the thread to finish

    while(thread.isAlive()) {

    System.out.println("Waiting...");

  }

  // Update amount and print its value

  System.out.println("Main: " + amount);

  amount++;

  System.out.println("Main: " + amount);

  }

  public void run() {

    amount++;

  }
```

```
}
```

Java Lambda Expressions

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Syntax :

The simplest lambda expression contains a single parameter and an expression:

```
parameter -> expression
```

To use more than one parameter, wrap them in parentheses:

```
(parameter1, parameter2) -> expression
```

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

```
(parameter1, parameter2) -> { code block }
```

Using Lambda Expressions

Lambda expressions are usually passed as parameters to a function:

Example :

Use a lambda expression in the ArrayList's forEach() method to print every item in the list:

```java
import java.util.ArrayList;


public class Main {

  public static void main(String[] args) {

    ArrayList<Integer> numbers = new ArrayList<Integer>();

    numbers.add(5);

    numbers.add(9);
```

```
    numbers.add(8);

    numbers.add(1);

    numbers.forEach( (n) -> { System.out.println(n); } );

  }

}
```

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.

Example :

Use Java's Consumer interface to store a lambda expression in a variable:

```
import java.util.ArrayList;

import java.util.function.Consumer;


public class Main {

  public static void main(String[] args) {

    ArrayList<Integer> numbers = new ArrayList<Integer>();

    numbers.add(5);

    numbers.add(9);

    numbers.add(8);

    numbers.add(1);

    Consumer<Integer> method = (n) -> { System.out.println(n); };

    numbers.forEach( method );

  }

}
```

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

Example :

Create a method which takes a lambda expression as a parameter:

```java
interface StringFunction {

  String run(String str);

}


public class Main {

  public static void main(String[] args) {

    StringFunction exclaim = (s) -> s + "!";

    StringFunction ask = (s) -> s + "?";

    printFormatted("Hello", exclaim);

    printFormatted("Hello", ask);

  }

  public static void printFormatted(String str, StringFunction format) {

    String result = format.run(str);

    System.out.println(result);

  }

}
```

Java Files

File handling is an important part of any application.

Java has several methods for creating, reading, updating, and deleting files.

Java File Handling

The File class from the java.io package, allows us to work with files.

To use the File class, create an object of the class, and specify the filename or directory name:

Example :

```java
import java.io.File;  // Import the File class
```

```
File myObj = new File("filename.txt"); // Specify the filename
```

If you don't know what a package is, read our Java Packages Tutorial.

The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory

mkdir()	Boolean	Creates a directory

Java Create and Write To Files

Create a File

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists. Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

Example :

```java
import java.io.File;  // Import the File class

import java.io.IOException;  // Import the IOException class to handle errors


public class CreateFile {

  public static void main(String[] args) {

    try {

      File myObj = new File("filename.txt");

      if (myObj.createNewFile()) {

        System.out.println("File created: " + myObj.getName());

      } else {

        System.out.println("File already exists.");

      }

    } catch (IOException e) {

      System.out.println("An error occurred.");

      e.printStackTrace();

    }
```

```
  }

}
```

The output will be:

File created: filename.txt

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

Example :

```
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

Example :

```
import java.io.FileWriter;   // Import the FileWriter class

import java.io.IOException;  // Import the IOException class to handle errors


public class WriteToFile {
  public static void main(String[] args) {

    try {

      FileWriter myWriter = new FileWriter("filename.txt");

      myWriter.write("Files in Java might be tricky, but it is fun enough!");

      myWriter.close();

      System.out.println("Successfully wrote to the file.");

    } catch (IOException e) {

      System.out.println("An error occurred.");

      e.printStackTrace();

    }
```

```
  }

}
```

The output will be:

Successfully wrote to the file.

Java Read Files

Read a File

In the previous chapter, you learned how to create and write to a file.

In the following example, we use the Scanner class to read the contents of the text file we created in the previous chapter:

Example :

```java
import java.io.File;  // Import the File class

import java.io.FileNotFoundException;  // Import this class to handle errors

import java.util.Scanner; // Import the Scanner class to read text files


public class ReadFile {

 public static void main(String[] args) {

  try {

   File myObj = new File("filename.txt");

   Scanner myReader = new Scanner(myObj);

   while (myReader.hasNextLine()) {

    String data = myReader.nextLine();

    System.out.println(data);

   }

   myReader.close();

  } catch (FileNotFoundException e) {

   System.out.println("An error occurred.");
```

```
      e.printStackTrace();

    }

  }

}
```

The output will be:

Files in Java might be tricky, but it is fun enough!

Get File Information

To get more information about a file, use any of the File methods:

Example :

```java
import java.io.File;  // Import the File class


public class GetFileInfo {
  public static void main(String[] args) {

    File myObj = new File("filename.txt");

    if (myObj.exists()) {

      System.out.println("File name: " + myObj.getName());

      System.out.println("Absolute path: " + myObj.getAbsolutePath());

      System.out.println("Writeable: " + myObj.canWrite());

      System.out.println("Readable " + myObj.canRead());

      System.out.println("File size in bytes " + myObj.length());

    } else {

      System.out.println("The file does not exist.");

    }

  }

}
```

The output will be:

File name: filename.txt
Absolute path: C:\Users\MyName\filename.txt
Writeable: true

Note: There are many available classes in the Java API that can be used to read and write files in Java: FileReader, BufferedReader, Files, Scanner, FileInputStream, FileWriter, BufferedWriter, FileOutputStream, etc. Which one to use depends on the Java version you're working with and whether you need to read bytes or characters, and the size of the file/lines etc.

Tip: To delete a file, read our Java Delete Files chapter.

Java Delete Files

Delete a File

To delete a file in Java, use the delete() method:

Example :

```java
import java.io.File; // Import the File class

public class DeleteFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the file.");
    }
  }
}
```

The output will be:

Deleted the file: filename.txt

Delete a Folder

You can also delete a folder. However, it must be empty:

```java
import java.io.File;



public class DeleteFolder {

  public static void main(String[] args) {

    File myObj = new File("C:\\Users\\MyName\\Test");

    if (myObj.delete()) {

      System.out.println("Deleted the folder: " + myObj.getName());

    } else {

      System.out.println("Failed to delete the folder.");

    }

  }

}
```

The output will be:

Deleted the folder: Test

Java How To Add Two Numbers

Add Two Numbers

Learn how to add two numbers in Java:

Example :

```java
int x = 5;

int y = 6;

int sum = x + y;

System.out.println(sum); // Print the sum of x + y
```

Add Two Numbers with User Input

Learn how to add two numbers with user input:

Example :

```java
import java.util.Scanner; // Import the Scanner class


class MyClass {
  public static void main(String[] args) {
    int x, y, sum;
    Scanner myObj = new Scanner(System.in); // Create a Scanner object
    System.out.println("Type a number:");
    x = myObj.nextInt(); // Read user input


    System.out.println("Type another number:");
    y = myObj.nextInt(); // Read user input


    sum = x + y;  // Calculate the sum of x + y
    System.out.println("Sum is: " + sum); // Print the sum
  }
}
```

Java How To Count Words

Count Number of Words in a String

You can easily count the number of words in a string with the following example:

Example :

```java
String words = "One Two Three Four";

int countWords = words.split("\\s").length;
```

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

Java How To Reverse a String

Reverse a String

You can easily reverse a string by characters with the following example:

Example :

```java
String originalStr = "Hello";

String reversedStr = "";


for (int i = 0; i < originalStr.length(); i++) {

  reversedStr = originalStr.charAt(i) + reversedStr;

}


System.out.println("Reversed string: "+ reversedStr);
```

Java How To Calculate the Sum of Elements

Calculate the Sum of an Array

Get the sum of array elements:

Example :

```java
int[] myArray = {1, 5, 10, 25};

int sum = 0;

int i;


// Loop through the array elements and store the sum in the sum variable
```

```java
for (i = 0; i < myArray.length; i++) {

  sum += myArray[i];

}


System.out.println("The sum is: " + sum);
```

Java How To Get the Area of a Rectangle

Area of Rectangle

The area of a rectangle can be found by multiplying the length of the rectangle by the width:

Example :

```java
int length = 5;

int width = 2;

int area = length * width;

System.out.println("Area of rectangle: " + area);
```

Java How To Find Even or Odd Numbers

Check Whether a Number is Even or Odd

Find out if a number is even or odd:

Example :

```java
int number = 5;
```

```
// Find out if the number above is even or odd

if (number % 2 == 0) {

  System.out.println(number + " is even.");

} else {

  System.out.println(number + " is odd.");
```

Java Keywords

Java Reserved Keywords

Java has a set of keywords that are reserved words that cannot be used as variables, methods, classes, or any other identifiers:

Note: true, false, and null are not keywords, but they are literals and reserved words that cannot be used as identifiers.

Keyword	Description
abstract	A non-access modifier. Used for classes and methods: An abstract class cannot be used to create objects (to access it, it must be inherited from another class). An abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from)
Assert	For debugging
boolean	A data type that can only store true and false values
break	Breaks out of a loop or a switch block

byte	A data type that can store whole numbers from -128 and 127
case	Marks a block of code in switch statements
catch	Catches exceptions generated by try statements
char	A data type that is used to store a single character
class	Defines a class
continue	Continues to the next iteration of a loop
Const	Defines a constant. Not in use - use final instead
default	Specifies the default block of code in a switch statement
do	Used together with while to create a do-while loop
double	A data type that can store whole numbers from 1.7e−308 to 1.7e+308
else	Used in conditional statements
enum	Declares an enumerated (unchangeable) type

Exports	Exports a package with a module. New in Java 9
extends	Extends a class (indicates that a class is inherited from another class)
final	A non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override)
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
float	A data type that can store whole numbers from 3.4e−038 to 3.4e+038
for	Create a for loop
Goto	Not in use, and has no function
if	Makes a conditional statement
implements	Implements an interface
import	Used to import a package, class or interface
instanceof	Checks whether an object is an instance of a specific class or an interface

int	A data type that can store whole numbers from -2147483648 to 2147483647
interface	Used to declare a special type of class that only contains abstract methods
long	A data type that can store whole numbers from -9223372036854775808 to 9223372036854775808
Module	Declares a module. New in Java 9
Native	Specifies that a method is not implemented in the same Java source file (but in another language)
new	Creates new objects
package	Declares a package
private	An access modifier used for attributes, methods and constructors, making them only accessible within the declared class
protected	An access modifier used for attributes, methods and constructors, making them accessible in the same package and subclasses
public	An access modifier used for classes, attributes, methods and constructors, making them accessible by any other class

Requires	Specifies required libraries inside a module. New in Java 9
return	Finished the execution of a method, and can be used to return a value from a method
short	A data type that can store whole numbers from -32768 to 32767
static	A non-access modifier used for methods and attributes. Static methods/attributes can be accessed without creating an object of a class
Strictfp	Restrict the precision and rounding of floating point calculations
super	Refers to superclass (parent) objects
switch	Selects one of many code blocks to be executed
synchronized	A non-access modifier, which specifies that methods can only be accessed by one thread at a time
this	Refers to the current object in a method or constructor
throw	Creates a custom error
throws	Indicates what exceptions may be thrown by a method

transient	A non-accesss modifier, which specifies that an attribute is not part of an object's persistent state
try	Creates a try...catch statement
Var	Declares a variable. New in Java 10
void	Specifies that a method should not have a return value
Volatile	Indicates that an attribute is not cached thread-locally, and is always read from the "main memory"
while	Creates a while loop

Java String Methods

All String Methods

The String class has a set of built-in methods that you can use on strings.

Method	Description	Return Type
charAt()	Returns the character at the specified index (position)	char
codePointAt()	Returns the Unicode of the character at the specified index	int
codePointBefore()	Returns the Unicode of the character before the specified index	int
codePointCount()	Returns the number of Unicode values found in a string.	int

compareTo()	Compares two strings lexicographically	int
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int
concat()	Appends a string to the end of another string	String
contains()	Checks whether a string contains a sequence of characters	boolean
contentEquals()	Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer	boolean
copyValueOf()	Returns a String that represents the characters of the character array	String
endsWith()	Checks whether a string ends with the specified character(s)	boolean
equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean
equalsIgnoreCase()	Compares two strings, ignoring case considerations	boolean
format()	Returns a formatted string using the specified locale, format string, and arguments	String
getBytes()	Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array	byte[]
getChars()	Copies characters from a string to an array of chars	void
hashCode()	Returns the hash code of a string	int
indexOf()	Returns the position of the first found occurrence of specified characters in a string	int
intern()	Returns the canonical representation for the string object	String

isEmpty()	Checks whether a string is empty or not	boolean
lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string	int
length()	Returns the length of a specified string	int
matches()	Searches a string for a match against a regular expression, and returns the matches	boolean
offsetByCodePoints()	Returns the index within this String that is offset from the given index by codePointOffset code points	int
regionMatches()	Tests if two string regions are equal	boolean
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String
split()	Splits a string into an array of substrings	String[]
startsWith()	Checks whether a string starts with specified characters	boolean
subSequence()	Returns a new character sequence that is a subsequence of this sequence	CharSequence
substring()	Returns a new string which is the substring of a specified string	String
toCharArray()	Converts this string to a new character array	char[]
toLowerCase()	Converts a string to lower case letters	String

toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends of a string	String
valueOf()	Returns the string representation of the specified value	String

Java Math Methods

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

All Math Methods

A list of all Math methods can be found in the table below:

Note: All Math methods are static.

Method	Description	Return Type
abs(x)	Returns the absolute value of x	double|float|int|long
acos(x)	Returns the arccosine of x, in radians	double
asin(x)	Returns the arcsine of x, in radians	double
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians	double

atan2(y,x)	Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).	double
cbrt(x)	Returns the cube root of x	double
ceil(x)	Returns the value of x rounded up to its nearest integer	double
copySign(x, y)	Returns the first floating point x with the sign of the second floating point y	double
cos(x)	Returns the cosine of x (x is in radians)	double
cosh(x)	Returns the hyperbolic cosine of a double value	double
exp(x)	Returns the value of E^x	double
expm1(x)	Returns e^x -1	double
'floor(x)	Returns the value of x rounded down to its nearest integer	double
getExponent(x)	Returns the unbiased exponent used in x	int

hypot(x, y)	Returns sqrt($x^2 + y^2$) without intermediate overflow or underflow	double
IEEEremainder(x, y)	Computes the remainder operation on x and y as prescribed by the IEEE 754 standard	double
log(x)	Returns the natural logarithm (base E) of x	double
log10(x)	Returns the base 10 logarithm of x	double
log1p(x)	Returns the natural logarithm (base E) of the sum of x and 1	double
max(x, y)	Returns the number with the highest value	double|float|int|long
min(x, y)	Returns the number with the lowest value	double|float|int|long
nextAfter(x, y)	Returns the floating point number adjacent to x in the direction of y	double|float
nextUp(x)	Returns the floating point value adjacent to x in the direction of positive infinity	double|float

pow(x, y)	Returns the value of x to the power of y	double
random()	Returns a random number between 0 and 1	double
round(x)	Returns the value of x rounded to its nearest integer	int
rint(x)	Returns the double value that is closest to x and equal to a mathematical integer	double
signum(x)	Returns the sign of x	double
sin(x)	Returns the sine of x (x is in radians)	double
sinh(x)	Returns the hyperbolic sine of a double value	double
sqrt(x)	Returns the square root of x	double
tan(x)	Returns the tangent of an angle	double
tanh(x)	Returns the hyperbolic tangent of a double value	double
toDegrees(x)	Converts an angle measured in radians to an	double

	approx. equivalent angle measured in degrees	
toRadians(x)	Converts an angle measured in degrees to an approx. angle measured in radians	double
ulp(x)	Returns the size of the unit of least precision (ulp) of x	double|float