# 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 **Students** that will store strings:

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

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

---

# 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> students = new ArrayList<String>();

    students.add("Ali");
    students.add("Zaineb");
    students.add("Mariam");
    students.add("Ahmed");
    System.out.println(students);
  }

}
```

# Access an Item

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

### Example

```
students.get(0);
```

---

# Change an Item

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

### Example

```
students.set(0, "Amena");
```

---

# Remove an Item

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

### Example

```
students.remove(0);
```

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

### Example

```
students.clear();
```

# ArrayList Size

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

### Example

```
students.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> students = new ArrayList<String>();
    students.add("Ali");
    students.add("Ahmed");
    students.add("Noor");
    students.add("Zena");
    for (int i = 0; i < students.size(); i++) {
      System.out.println(students.get(i));
    }
  }
}
```

# Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". 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`):

```
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=0;i< myNumbers.size();i++) {

       System.out.println(myNumbers.get(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:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class
public class Main {
  public static void main(String[] args) {
    ArrayList<String> students = new ArrayList<String>();
    students.add("Ali");
    students.add("Noor");
    students.add("Mohammed");
    students.add("Maha");
    Collections.sort(students);  // Sort students
    for (int i=0;i<students.size();i++) {
      System.out.println(students.get(i));
    }
  }
}
```

## Example

Sort an ArrayList of Integers:

```
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=0;i< myNumbers.size();i++) {
      System.out.println(myNumbers.get(i));
    }
  }
}
```

Java is a programming language.

Java is used to develop mobile apps, web apps, desktop apps, games and much more.

# 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 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 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.

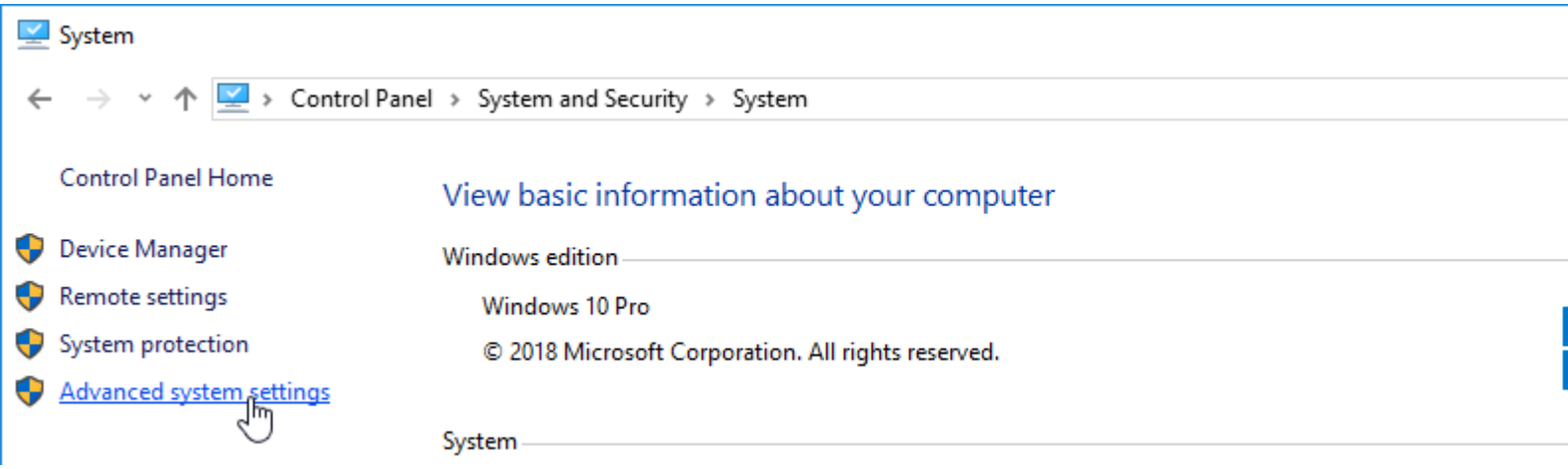
**Note:** In this tutorial, we will write Java code in a text editor. However, it is possible to write Java in an Integrated Development Environment, such as IntelliJ IDEA, Netbeans or Eclipse, which are particularly useful when managing larger collections of Java files.

# Setup for Windows

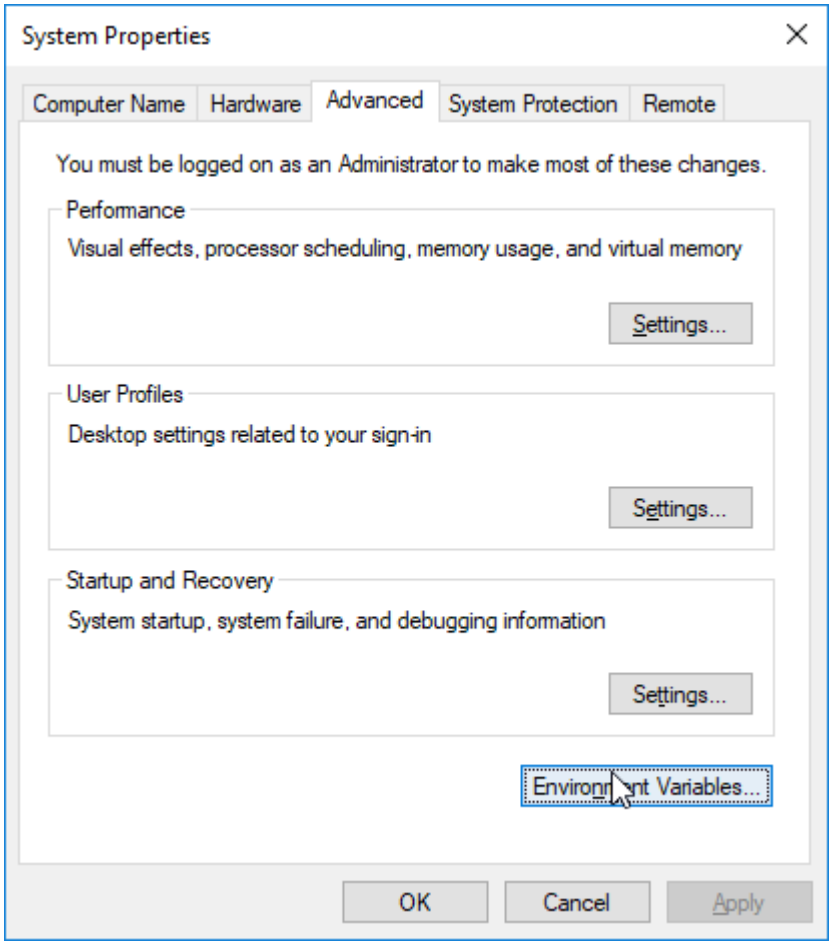
To install Java on Windows:

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

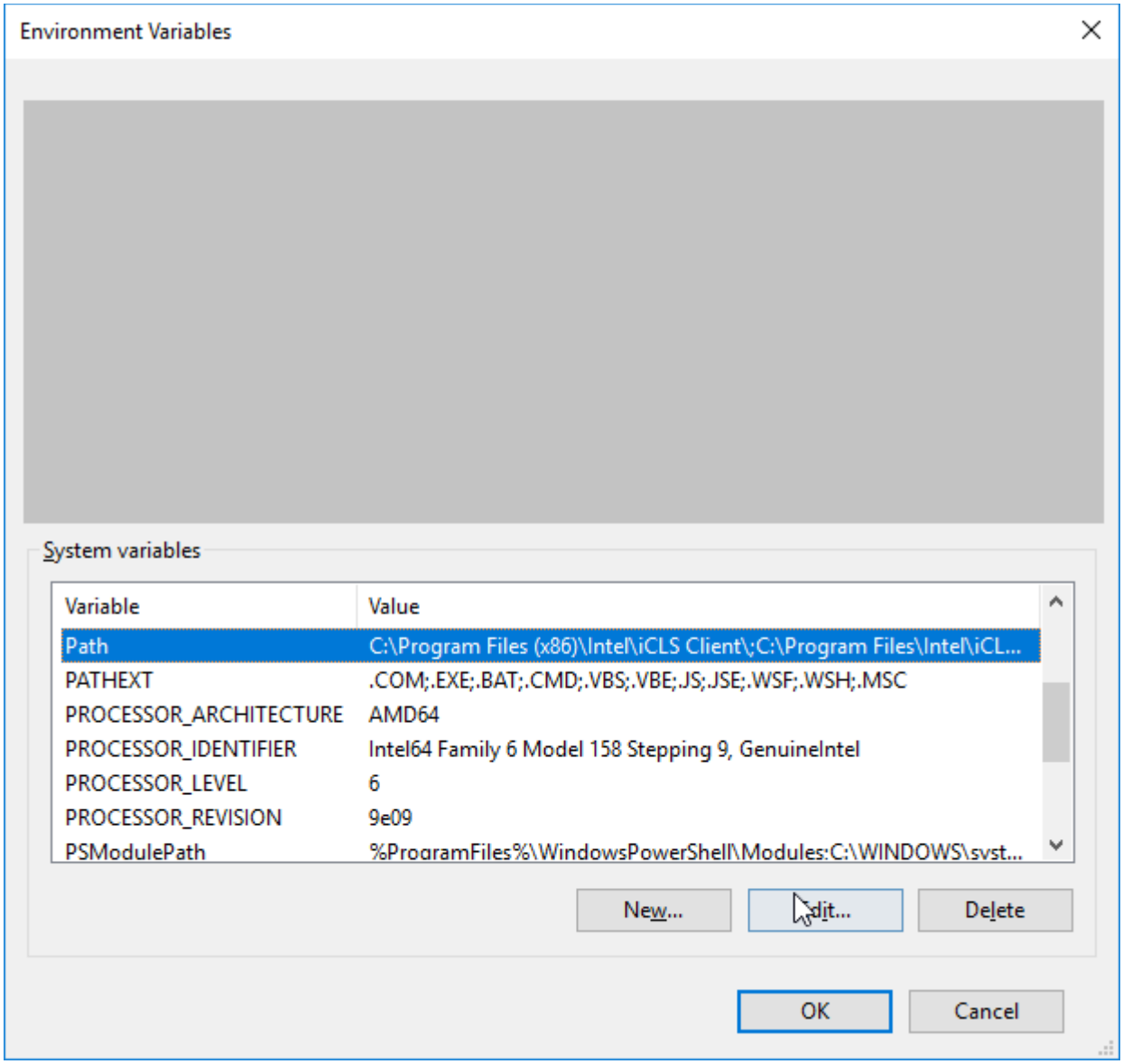
# Step 1


System
Control Panel › System and Security › System
Control Panel Home
Device Manager
Remote settings
System protection
Advanced system settings
View basic information about your computer
Windows edition
Windows 10 Pro
© 2018 Microsoft Corporation. All rights reserved.
System

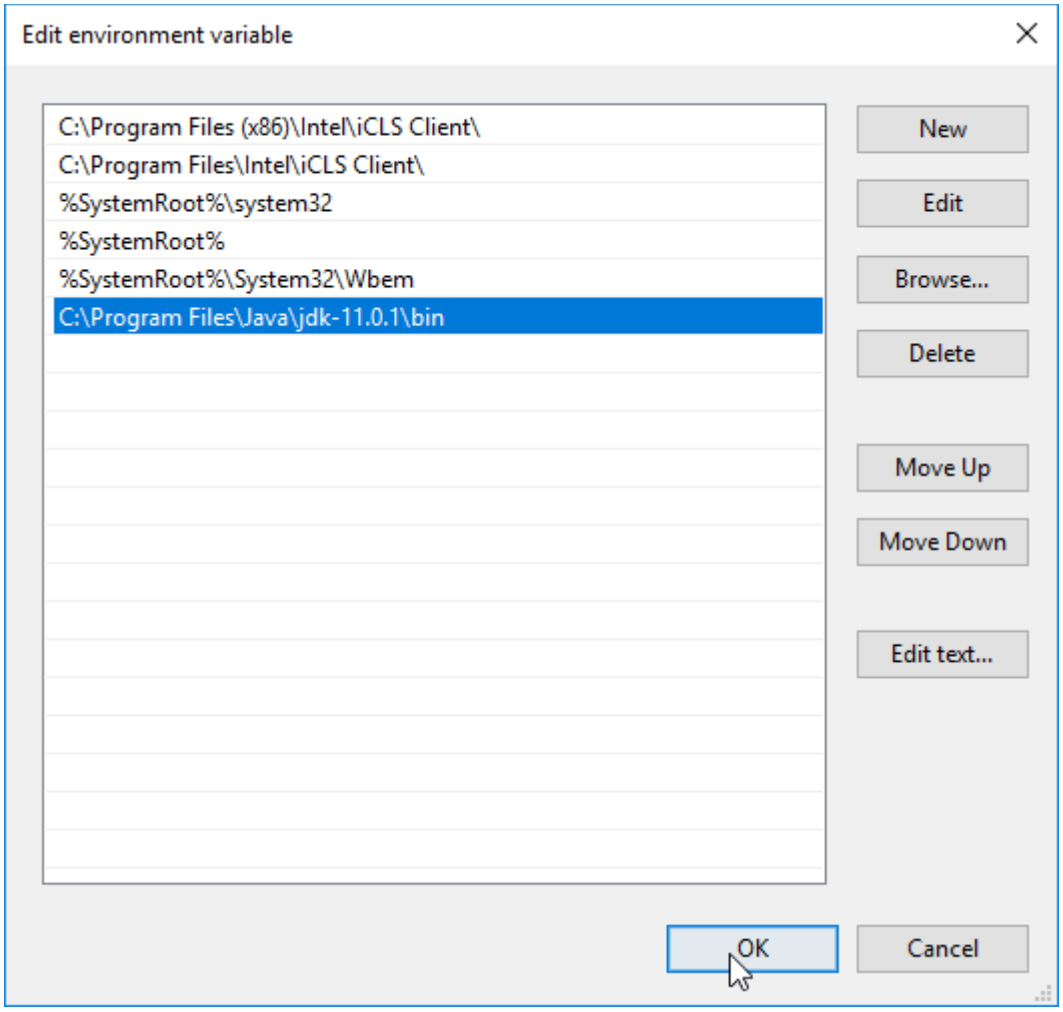
# Step 2


System Properties
Computer Name
Hardware
Advanced
System Protection
Remote
You must be logged on as an Administrator to make most of these changes.
Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...
User Profiles
Desktop settings related to your sign-in
Settings...
Startup and Recovery
System startup, system failure, and debugging information
Settings...
Environment Variables...
OK
Cancel
Apply

# Step 3


Environment Variables
System variables
Variable
Value
Path
C:\Program Files (x86)\Intel\iCLS Client\;C:\Program Files\Intel\iCL...
PATHEXT
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE
AMD64
PROCESSOR_IDENTIFIER
Intel64 Family 6 Model 158 Stepping 9, GenuineIntel
PROCESSOR_LEVEL
6
PROCESSOR_REVISION
9e09
PSModulePath
%ProgramFiles%\WindowsPowerShell\Modules;C:\WINDOWS\syst...
New...
Edit...
Delete
OK
Cancel

# Step 4




# Step 5

Write the following in the command line (cmd.exe):

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

If Java was successfully 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)
```

# 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

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

# Java Syntax

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.

**Note1:** Java is case-sensitive: "MyClass" and "myclass" has different meaning.

**Note2:**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.

## The main Method:

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

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

Any code inside the `main()` method will be executed. You don't have to understand the keywords before and after main. You will get to know them bit by bit while learning java in this course.

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.

# Print command:

**System.out.println()**,inside the `main()` method, we can use the `println()` method to print a line of text to the screen:

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

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

}
```

**Note1:** The curly braces `{}` marks the beginning and the end of a block of code.

**Note2:** Each code statement must end with a semicolon.

# 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 start with two forward slashes (`//`).
- Any text between `//` and the end of the line is ignored by Java (will not be executed).

The following example uses a single-line comment before a line of code:

## Example

```
// This is a comment

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

### Java Multi-line Comments

- Multi-line comments start with `/*` and ends with `*/`.
- Any text between `/*` and `*/` will be ignored by Java.

- The following 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");
```

# 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:

## Syntax

```
type variable = value;
```

Where *type* is one of Java's types (such as `int` or `String`), and *variable* 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:

## 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`:

```
int myNum = 15;

myNum = 20;  // myNum is now 20

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

# Final Variables

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

## Example

```
final int myNum = 15;

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

# Java Data Types

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

## Example

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

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 later)

# 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 |

# Numbers

Primitive number types are divided into two groups:

**Integer types** stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `byte`, `short`, `int` and `long`. Which type you should use, depends on the numeric value.

**Floating point types** represents numbers with a fractional part, containing one or more decimals. There are two types: `float` and `double`.

## Example

```
int myNum = 100000;

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

### Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

## Float

The `float` data type can store fractional numbers from 3.4e−038 to 3.4e+038. Note that you should end the value with an "f":

## Example

```
float myNum = 5.75f;

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

## Double

The `double` data type can store fractional numbers from 1.7e−308 to 1.7e+308. Note that you should end the value with a "d":

## Example

```
double myNum = 19.99d;

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

### Booleans

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

## Example

```
boolean isJavaFun = true;
```

```java
boolean isFishTasty = false;

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

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

**Note**: Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

## Characters

The `char` data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

### Example

```java
char myGrade = 'B';

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

## Strings

The `String` data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

### Example

```java
String greeting = "Hello World";

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

## 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 = 9.78;
```

```
    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:

## Example

```
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:

## Examples

```
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 :

**1- 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 |

**2-Java 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) |

**3-Java Comparison Operators** are used to compare two values:

| 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 |

# String methods:

## 1- **String Length** method

A 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:

**2-**`toLowerCase()`: It used to convert the given string to lowercase letters.

**3-**`toUpperCase()` : It used to convert the given string to uppercase letters.

### Example

```
String txt = "Hello World";

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

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

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

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

Example

```
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

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

**5-String Concatenation** the `+` operator can be used between strings to combine them. This is called **concatenation**:

Example

```
String firstName = "John";
String lastName = "Doe";
System.out.println(firstName + " " + lastName);
```

Note that we have added an empty text (" ") to create a space between firstName and lastName on print.

# Adding Numbers and Strings

WARNING!

Java uses the `+` operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

## Example

```
int x = 10;

int y = 20;

int z = x + y;       // z will be 30 (an integer/number)
```

If you add two strings, the result will be a string concatenation:

## Example

```
String x = "10";

String y = "20";

String z = x + y;    // z will be 1020 (a String)
```

# Java Math

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

## • Math.max(*x,y*)

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

### Example

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

## • Math.min(*x,y*)

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

### Example

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

## • Math.sqrt(*x*)

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

### Example

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

- # Math.abs(*x*)

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

## Example

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

- # Random Numbers

## A:

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

## Example

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

## B:

To get more control over the random number, e.g. you only want a random number between 0 and 100, you can use the following formula:

## Example

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

C: generate random integers within a specific range in Java

R=min + (int)(Math.random() * ((max - min) + 1))

Min :is the minimum value in the range.
Max :is the maximum value in the range.
In order to get the max value in the range we must add 1 to the range(max-min).

# Java Booleans

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 take the values `true` or `false`.

# • 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
```

# • Boolean Expression

A **Boolean expression** is a Java expression that returns a Boolean value: `true` or `false`.

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

## Example

```
int x = 10;

int y = 9;

System.out.println(x > y); // 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
```

# If ... Else

## • Java Conditions and If Statements

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

# • 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:

## Example

```
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".

## • 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.");
}
// Outputs "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".

## • 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 < 20) {

  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."

# Switch Statements

Use the `switch` statement to select 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) {

  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

```
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 Loop

- Loops

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

- Java While Loop

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

## Syntax

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

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

## Example

```
int i = 0;
```

```
while (i < 5) {

  System.out.println(i);

  i++;

}
```

**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:

## Example

```
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!

# For Loop

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

## Syntax

```
for (statement 1; statement 2; statement 3) {

  // code block to be executed

}
```

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

**Statement 2** defines the condition for executing the code block.

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

The example below will print the numbers 0 to 4:

## Example

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

  System.out.println(i);

}
```

### *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

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

# Java Break and Continue

- ## Java Break

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

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

This example jumps out of the loop when i is equal to 4:

## Example

```
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

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  System.out.println(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, we can use an array literal - 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};
```

To create an array of characters , you could write:

```
char[] myNum = {‘a’, ‘b’,’c’,’d’};
```

# Access the Elements of an Array

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

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

## Example

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

System.out.println(cars[0]);// Outputs Volvo

System.out.println(cars[1]);// Outputs BMW
```

# Change an Array Element

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

## Example

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

## Example

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

cars[0] = "Opel";

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

// Now outputs Opel instead of Volvo
```

# Array Length

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

## Example

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

System.out.println(cars.length);

// Outputs 4
```

# Loop Through an Array

You can loop through the array elements with the `for` loop, and use the `length` property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

## Example

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

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

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

}
```

# Multidimensional Arrays

A multidimensional array is an array containing one or more arrays.

To create a two-dimensional array, add each array within its own set of **curly braces**:

## Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

**myNumbers** is now an array with two arrays as its elements.

To access the elements of the **myNumbers** array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

## Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

int x = myNumbers[1][2];

System.out.println(x); // Outputs 7
```

We can also use a `for loop` inside another `for loop` to get the elements of a two-dimensional array (we still have to point to the two indexes):

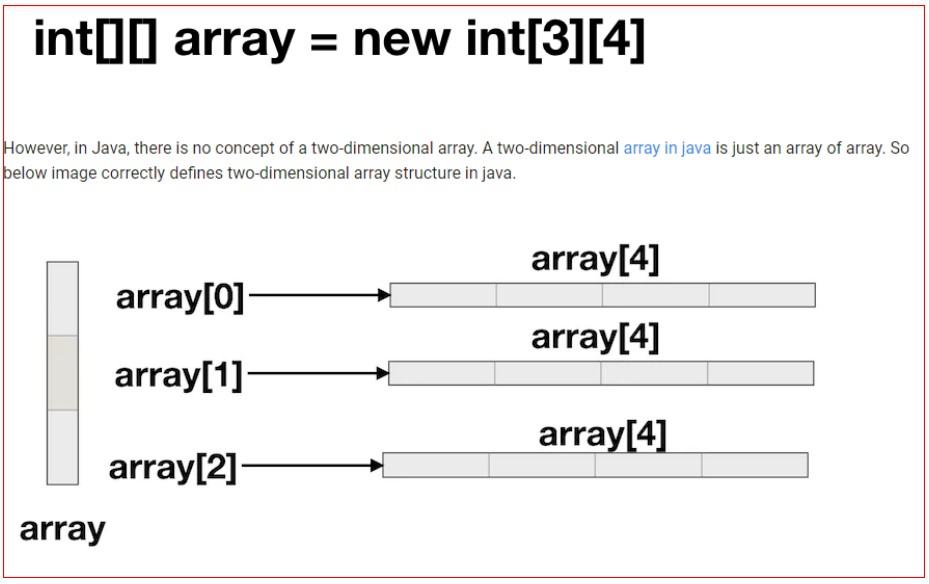
## Example

```
public class Main {

  public static void main(String[] args) {

    int[][] Array = { {1, 2, 3, 4}, {5, 6, 7} };

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

      for(int j = 0; j < array[i].length; ++j) {

        System.out.println(array[i][j]);

      }
    }

  }
}
```

1 2 3 4 a[0][0] a[0][1] a[0][2] a[0][3]

2 3 4 5 a[1][0] a[1][1] a[1][2] a[1][3]

2 2 3 4 a[2][0] a[2][1] a[2][2] a[2][3]



# 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.
- `void` means that this method does not have a return value. You will learn more about return values later.

# Call a Method

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

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

## Example

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

```
public class MyClass {

  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

```
public class MyClass {
  static void myMethod() {
    System.out.println("I just got executed!");
  }
  public static void main(String[] args) {
    myMethod();
    myMethod();
    myMethod();
  }
}
/ I just got executed!
// I just got executed!
// I just got executed!
```

# Java Method Parameters

## Parameters and Arguments

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

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

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

### Example

```
public class Myclass {

  static void myMethod(String name) {

    System.out.println(" the name is "+ name   );

  }


  public static void main(String[] args) {

    myMethod("Mariam");

    myMethod("Zaenab");

    myMethod("Amina");

  }

}
```

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

```
public class MyClass {

  static int myMethod(int x) {

   int s;

   s=(x*x);

    return s;

  }

  public static void main(String[] args) {

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

  }

}

// Outputs 9 (3*3)
```

## Example

```
public class MyClass {
  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

```
public class Main {



  // Create a checkAge() method with an integer variable called age, to
Apply for a job

  static void checkAge(int age) {

    // If age is less than 25, print "You cannot get this job"

    if (age < 25) {

      System.out.println("Access denied - You cannot get this job!");



    // If age is greater than, or equal to 25, print "access granted You
are old enough!"

    } else {

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

    }

  }



  public static void main(String[] args) {

    checkAge(30); // 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 have two methods that add numbers of different type:

### Example

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

  return x + y;

}



static double plusMethodDouble(double x, double y) {

  return x + y;

}

public static void main(String[] args) {

  System.out.println("int: " + plusMethodInt(8, 5));

  System.out.println("double: " + plusMethodDouble(4.3, 6.26));

}
```

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

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

## Example

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

static double plusMethod(double x, double y) {
  return x + y;
}

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

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

# 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

```
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. See example below:

```
public class Myclass {

  public static void main(String[] args) {

  for (int i=0; i<10;i++) {
    // Code here CAN use i
}
// Code here CANNOT use i
}
}
```

# Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

**Syntax:**

```
returntype methodname(){
//code to be executed
methodname();//calling same method
}
```

## Java Recursion Example 1: Infinite times

```
public class RecursionExample1 {
static void p(){
System.out.println("hello");
p();
}

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

Output:

```
hello
hello
...
java.lang.StackOverflowError
```

## Java Recursion Example 2: Finite times

```
public class RecursionExample2 {
static int count=0;
static void p(){
count++;
if(count<=5){
System.out.println("hello "+count);
p();
}
}
public static void main(String[] args) {
p();
}
}
```

Output:

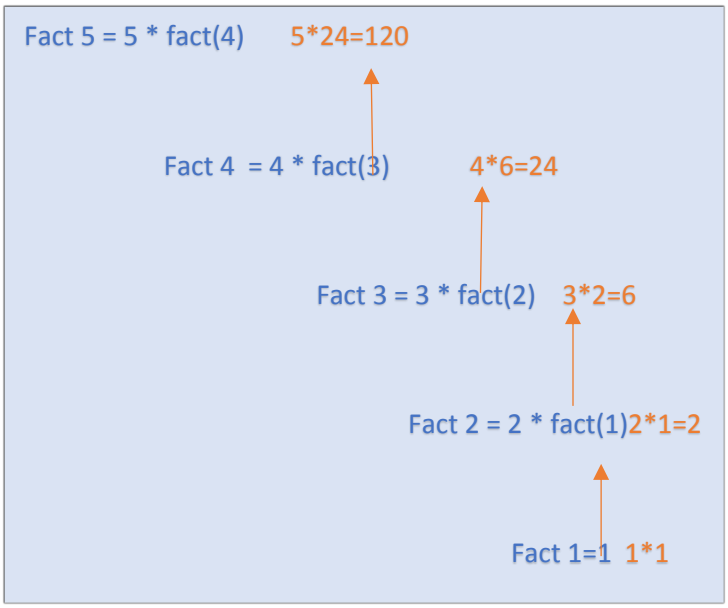
```
hello 1
hello 2
hello 3
hello 4
hello 5
```

## Java Recursion Example 3: Factorial Number

```
public class RecursionExample3 {
   static int factl(int n){
        if (n == 1)
          return 1;
        else
          return(n * factl(n-1));
   }
  public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+fact(5));
}
}
```

Output:

```
Factorial of 5 is: 120
```



# 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 | objects |
|---|---|
| Fruit | Apple |
| | Banana |
| | Mango |

Another example:

| Class | objects |
|---|---|
| Car | 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.

# Create an Object

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

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

## Example

Create an object called "`myObj`" and print the value of x:

```
public class MyClass {
  int x = 5;


  public static void main(String[] args) {
    MyClass myObj = new Myclass(); //myObj has one member is X
    System.out.println(myObj.x); //To call member belong to object write:
      object.member
  }
}
```

# Multiple Objects

You can create multiple objects of one class:

```
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***

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

***Second.java***

```
class Second {

  public static void main(String[] args) {

    Main myObj = new Main();

    System.out.println(myObj.x);

  }

}
```

the output will be: `5`

# Java Class Attributes

In the previous example we used the term "variable" for `x`, in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class:

## Example

Create a class called "`Main`" with two attributes: `x` and `y`:

```
public class Main {

  int x = 5;

  int y = 3;

}
```

# Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax (`.`):

The following example will create an object of the `Main` class, with the name `myObj`. We use the `x` attribute on the object to print its value:

## Example

Create an object called "`myObj`" and print the value of `x`:

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

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

# Modify Attributes

You can also modify attribute values:

## Example

Set the value of `x` to 40:

```
public class Main {
  int x;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

Or override existing values:

## Example

Change the value of `x` to 25:

```
public class Main {

  int x = 10;


  public static void main(String[] args) {

    Main myObj = new Main();

    myObj.x = 25; // x is now 25

    System.out.println(myObj.x);

  }

}
```

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

## Example

```
public class Main {

  final int x = 10;

  public static void main(String[] args) {

    Main myObj = new Main();

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

    System.out.println(myObj.x);

  }
}
```

# Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

## Example

Change the value of `x` to 25 in `myObj2`, and leave `x` in `myObj1` unchanged:

```
public class Main {

  int x = 5;

  public static void main(String[] args) {

    Main myObj1 = new Main();  // Object 1

    Main myObj2 = new Main();  // Object 2

    myObj2.x = 25;

    System.out.println(myObj1.x);  // Outputs 5

    System.out.println(myObj2.x);  // Outputs 25

  }

}
```

# Multiple Attributes

You can specify as many attributes as you want:

**Example**

```
public class Main {
  String fname = "John";
  String lname = "Doe";
  int age = 24;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }
}
```

# Java Class Methods

You learned from the Java Methods lecture that methods are declared within a class, and that they are used to perform certain actions:

## Example

Create a method named `myMethod()` in Main:

```
public class Main {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

`myMethod()` prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

## Example

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

```
public class Main {
  static void myMethod() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "Hello World!"
```

# Static vs. Non-Static

You will often see Java programs that have either `static` or `public` attributes and methods.
In the example above, we created a `static` method, which means that it can be accessed without creating an object of the class, unlike `public`, which can only be accessed by objects:

## Example

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

```
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 compile an error

    Main myObj = new Main(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method on the object
  }
}
```

# Access Methods With an Object

## Example

Create a Car object named `myCar`. Call the `fullThrottle()` and `speed()` methods on the `myCar` object, and run the program:

```
// Create a Main class
public class Main {

  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Main myCar = new Main();   // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
}
// The car is going as fast as it can!
// Max speed is: 200
```

## Example explained

1) We created a custom `Main` class with the `class` keyword.
2) We created the `fullThrottle()` and `speed()` methods in the `Main` class.
3) The `fullThrottle()` method and the `speed()` method will print out some text, when they are called.
4) The `speed()` method accepts an `int` parameter called `maxSpeed` - we will use this in **8)**.
5) In order to use the `Main` class and its methods, we need to create an **object** of the `Main` Class.
6) Then, go to the `main()` method, which you know by now is a built-in Java method that runs your program (any code inside main is executed).
7) By using the `new` keyword we created an object with the name `myCar`.

8) Then, we call the `fullThrottle()` and `speed()` methods on the `myCar` object, and run the program using the name of the object (`myCar`), followed by a dot (`.`), followed by the name of the method (`fullThrottle();` and `speed(200);`). Notice that we add an `int` parameter of **200** inside the `speed()` method.

## Remember that..

The dot (`.`) is used to access the object's attributes and methods.
To call a method in Java, write the method name followed by a set of parentheses **()**, followed by a semicolon (`;`).
A class must have a matching filename (`Main` and **Main.java**).

# Methods with Multiple Classes

Like we specified in the Classes lecture, it is a good practice to create an object of a class and access it in another class. In this example, we have created two files in the same directory:

- Main.java
- Second.java

***Main.java***

```
public class Main {
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
}
```

***Second.java***

```
class Second {
  public static void main(String[] args) {
    Main myCar = new Main();     // Create a myCar object
    myCar.fullThrottle();       // Call the fullThrottle() method
    myCar.speed(200);           // Call the speed() method
  }
}
```

the output will be:

```
The car is going as fast as it can!
Max speed is: 200
```

# 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
```

**Note1** that the constructor name must **match the class name**, and it cannot have a **return type** (like `void,int,….`).
**Note2:** that the constructor is called when the object is created.
**Note3:**All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you.

# 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
```

**Note:**You can have as many parameters as you want:

## Example

```
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, "opel");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}

// Outputs 1969 opel
```

# Java Modifiers

## -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;
  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`:

## Repeated Example

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

```
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
  }
}
```

# 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

```
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

```
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:

- Instead, we use the `getName()` and `setName()` methods to acccess and update the variable:

## Example

```
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 & 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 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:

```
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

```
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

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

**Note1**:Save the file as **MyPackageClass.java**, and compile it:

**Note2:** The package name should be written in lower case to avoid conflict with class names.

# 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

```
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.
- `Protected` access modifier refers to ability to accessing to the attributes from the Parent class and sub class only.

### 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.

# 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 {
  ...
}
```