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

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:

Example

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

public class Main {

int x = 5; //java Attribute

public static void main(String[] args) {

Main **myObj** = new Main();

System.out.println(myObj.x);

}

}

The final keyword is useful when you want a variable to always store the same value

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

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

}

}

## 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

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, "Mustang");

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

}

}

// Outputs 1969 Mustang

Access Modifiers

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

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** | **Try it** |
| public | The class is accessible by any other class | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_mod_public) |
| *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](https://www.w3schools.com/java/java_packages.asp) | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_mod_default) |

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

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** | **Try it** |
| public | The code is accessible for all classes | [Try it »](https://www.w3schools.com/java/tryjava_multi.asp?filename=demo_mod_public2&multi=demo_mod_public2_multi) |
| private | The code is only accessible within the declared class | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_access_mod) |
| *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](https://www.w3schools.com/java/java_packages.asp) | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_mod_default2) |
| protected | The code is accessible in the same package and **subclasses**. You will learn more about subclasses and superclasses in the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp) |  |

## 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

public class Person {

private String name; // private = restricted access

// Getter

public String getName() {

return name;

}

// Setter

public void setName(String newName) {

this.name = newName;

}

}

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"

ava 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)

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

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

for example, the Scanner class, **which is used to get user input**, write the following code:

### Example

import java.util.Scanner;

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!");

}

}

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

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 (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.

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);

}

}

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");

}

}

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

// 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)

}

// 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).

## 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:

enum Level {

LOW,

MEDIUM,

HIGH

}

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

Level myVar = Level.MEDIUM;

## 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

## 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.

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

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

public class Main {

public static void main(String[] args) {

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

LocalTime myObj = LocalTime.now();

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

}

}