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

If you have Class other than main with some variables then you need parametrrized constructor to intiallize the values.

Create an Object

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

public class ABC{

int x = 5; //java Attribute

public static void main(String[] args) {

ABC **myObj** = new ABC ();

System.out.println(myObj.x);

}

}

## 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

## Constructor Overloading

Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. Or different data types

## Constructor Chaining

Constructor chaining in Java is a practice where one constructor calls another constructor of the same class or a superclass during object creation.

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

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. |  |

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 |  |
| protected | The code is accessible in the same package and **subclasses**. |  |

Java Inheritance

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);

}

}

Types of Inheritance

1. single 2) Multilevel 3) Hierarchical

Multiple and hybrid not supported & Can only be achived by Interfaces

# Polymorphism

### Method Overloading (compile-time polymorphism or static or early binding)

### Inheritance not required

1. By changing number of arguments
2. By changing the data type

### method overriding (runtime polymorphism)

### Inheritance required

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

1. **class** Bike{
2. **void** run(){System.out.println("running");}
3. }
4. **class** Splendor **extends** Bike{
5. **void** run(){System.out.println("running safely with 60km");}
7. **public** **static** **void** main(String args[]){
8. Bike b = **new** Splendor();//upcasting
9. b.run();
10. }
11. }

running safely with 60km

| **Compile Time Polymorphism** | **Run time Polymorphism** |
| --- | --- |
| It is also known as Static binding, Early binding and overloading as well. | It is also known as Dynamic binding, Late binding and overriding as well. |
| It is achieved by method overloading | It is achieved by virtual functions and pointers. |
| Compile time polymorphism is less flexible as all things execute at compile time. | Run time polymorphism is more flexible as all things execute at run time. |
| Inheritance is not involved. | Inheritance is involved. |

The ****super**** keyword in Java is a reference variable which is used to refer immediate parent class object / variable /method /Constructor

## 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"

# Java Abstraction

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

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");

}

}

* abstract class have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* abstract class have final methods which will force the subclass not to change the body of the method.

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 Main class)
* 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.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can ****have abstract and non-abstract**** methods. | Interface can have ****only abstract**** methods. Since Java 8, it can have ****default and static methods**** also. |
| 2) Abstract class ****doesn't support multiple inheritance****. | Interface ****supports multiple inheritance****. |
| 3) Abstract class ****can have final, non-final, static and non-static variables****. | Interface has ****only static and final variables****. |
| 4) Abstract class ****can provide the implementation of interface****. | Interface ****can't provide the implementation of abstract class****. |
| 5) The ****abstract keyword**** is used to declare abstract class. | The ****interface keyword**** is used to declare interface. |
| 6) An ****abstract class**** can extend another Java class and implement multiple Java interfaces. | An ****interface**** can extend another Java interface only. |
| 7) An ****abstract class**** can be extended using keyword "extends". | An ****interface**** can be implemented using keyword "implements". |
| 8) A Java ****abstract class**** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)****Example:**** public abstract class Shape{ public abstract void draw(); } | ****Example:**** public interface Drawable{ void draw(); } |

## 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 myObj2 = LocalTime.now();

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

}

}

1. What is construtor?
2. What are access modfifers
3. What is encapsulation and Inheritance and abstraction and Polymorphism
4. What is interface and when to use
5. Interface vs abtract class
6. What is enum and dates in Java.