**Week 1 – Day 1**

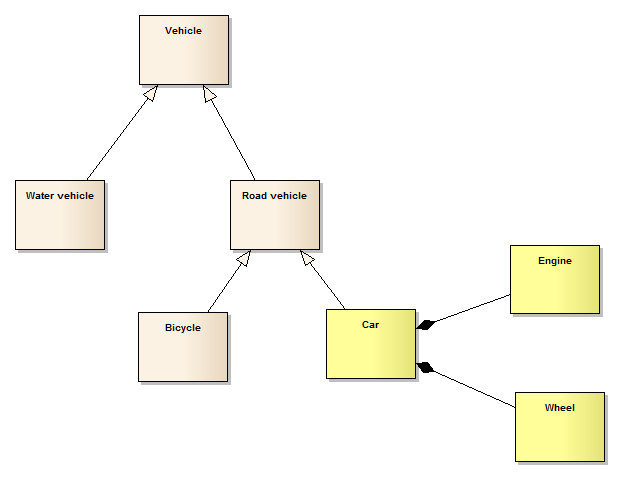
**Assignment 1 :-**

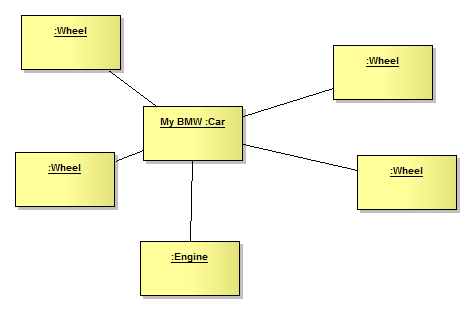
**Q. Design class hierarchy to show understanding of basic OOPs concepts listed above.**

There are not the same thing, they are in fact **very different things**. Confusingly enough, class hierarchy is informally often called object hierarchy.

In order to understand the difference, you should understand the difference between **class and object**. This is the heart of OO theory and you can read about it in any OO book or simply google it.

As a quick help, here is a simple example of both hierarchies and both (class and object):

A class hierarchy refers to a taxonomy of **classes** based on inheritance relationships between them - in the following example all classes under the Vehicle (excluding Engine and Wheel), directly or indirectly inherited from it are forming a **class hierarchy**:

**Objects** are instances of classes, so they "live" on another level of abstraction (e.g. in run-time). Classes are templates for creating their concrete instances - objects. Objects connect themselves and this network of interconnected object form **object hierarchy**. Here, the objects Car, Engine and 4 Wheels are inteconnected in a object hierarchy:

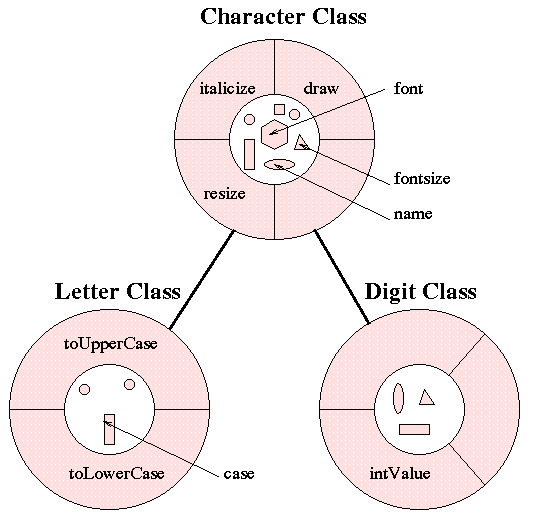
In object-oriented programming, a class is a template that defines the state and behavior common to objects of a certain kind. A class can be defined in terms of other classes. For example, a truck and a racing car are both examples of a car. Another example is a letter and a digit being both a single character that can be drawn on the screen. In the latter example, the following terminology is used:

* The letter class is a *subclass* of the character class; (alternative names: *child class* and *derived class*)
* The character class is immediate *superclass* (or *parent class*) of the letter class;
* The letter class *extends* the character class.

The third formulation expresses that a subclass *inherits* state (instance variables) and behavior (methods) from its superclass(es). Letters and digits share the state (name, font, size, position) and behavior (draw, resize, ...) defined for single characters.

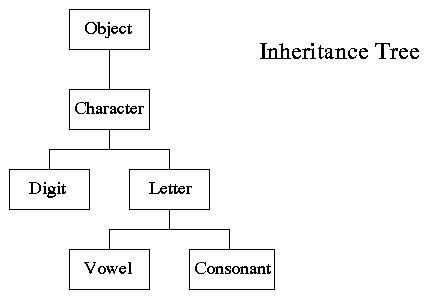
The purpose of a subclass is to extend existing state and behavior: a letter has a case (upper and lower case, say stored in the instance variable *letterCase*) and methods for changing the case (*toUpperCase*, *toLowerCase*) in addition to the state that it already has as a character.

However, a digit does not have a case, so the methods *toUpperCase*, *toLowerCase* do not belong on the common level of the Character class. There are methods that are special to the digit class. For instance, a digit may be constructed from an integer value between 0 and 9, and conversely, the integer value of a digit may be the result of say the *intValue* method.

Subclasses can also override inherited behavior: if you had a colored character as subclass of the character class, you would override the definition of the *draw* method of the character class so that color is taken into account when drawing the character on the screen. This leads to what is called in OOP jargon *polymorphism*: the same message sent to different objects results in behavior that is dependent on the nature of the object receiving the message.

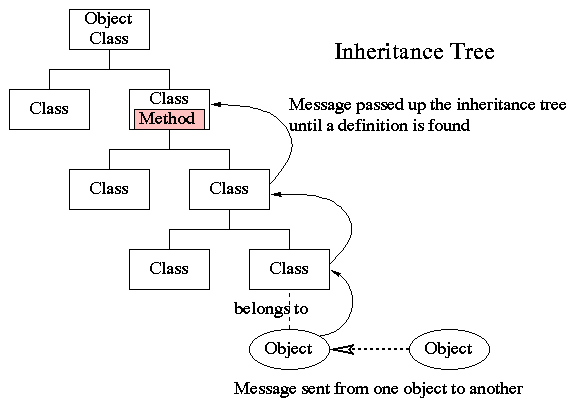
In graphical terms, the above character example may look as follows:

You are not limited to just one layer of inheritance: for example, the letter class can have on its turn the subclasses vowel and consonant.

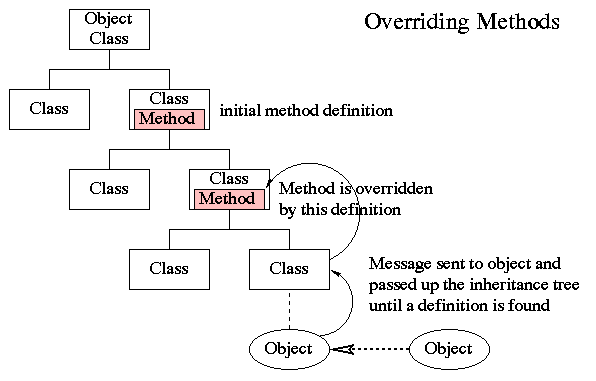


The classes form a *class hierarchy*, or *inheritance tree*, which can be as deep as needed. The hierarchy of classes in Java has one *root class*, called Object, which is superclass of any class. Instance variable and methods are inherited down through the levels. In general, the further down in the hierarchy a class appears, the more specialized its behavior. When a message is sent to an object, it is passed up the inheritance tree starting from the class of the receiving object until a definition is found for the method. This process is called *upcasting*. For instance, the method toString() is defined in the Object class. So every class automatically has this method. If you want that your particular toString() method looks differently, you can reimplement it in your class. In this way you can override a method in a given class by redefining it in a subclass.

In graphical terms, the inheritance tree and the message handling may look as follows:



The picture showing the overriding of methods, may look as follows:



Source: [OOP: Class Hierarchy](https://staff.fnwi.uva.nl/a.j.p.heck/Courses/JAVAcourse/ch3/s1.html)

**Object Oriented Programming (OOPs) Concept**

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

## **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;

}

## **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;

public static void main(String[] args) {

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

System.out.println(myObj.x);

}

}

## **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;

}

}

## **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

# Data Hiding

Data Abstraction and Data Hiding concepts in OOP are used to show only the required information to the end-user and hide the unnecessary details but for distinct purposes such as reducing the system's complexity to make it more user-friendly.

## **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);

}

}

## **Java Polymorphism**

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; [**Inheritance**](https://www.w3schools.com/java/java_inheritance.asp) lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

### **Example**

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}

}

class Pig extends Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

class Dog extends Animal {

public void animalSound() {

System.out.println("The dog says: bow wow");

}

}

Now we can create Pig and Dog objects and call the animalSound() method on both of them:

### **Example**

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}

}

class Pig extends Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

class Dog extends Animal {

public void animalSound() {

System.out.println("The dog says: bow wow");

}

}

class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

## **Interfaces**

Another way to achieve [abstraction](https://www.w3schools.com/java/java_abstract.asp) 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)

}

// 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();

}

}

# Methods

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

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

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

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

## **Create a Method**

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

### **Example**

Create a method inside Main:

public class Main {

static void myMethod() {

// code to be executed

}

}

#### **Example Explained**

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

## **Call a Method**

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

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

### **Example**

Inside main, call the myMethod() method:

public class Main {

static void myMethod() {

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

}

public static void main(String[] args) {

myMethod();

}

}

// Outputs "I just got executed!"

## **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**](https://www.w3schools.com/java/java_interface.asp) (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:

### **Example**

// 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();

}

}