College of Computer and Information Systems Department of Computer Science

CS 1704: Object Oriented Programming

2nd Trimester, 2026

Lecture 1

OOP – Part1

Prepared By:

Dr. Azhar Hasan Alhindi

Textbook/References

- java How to Program, 10th Edition. Download Java How to Program, 10th Edition free PDF by Harvey Deitel, Paul Deitel - OiiPDF.COM

- Murach's Beginning Java with NetBeans. Murach's Beginning Java with NetBeans - PDF Drive

- Java Programming from Problem analysis to program design by D. S. Malik 5th Edition. Java Programming from Problem analysis to program design by D. S. Malik 5th Edition - Book Free (booksfree.org)

. Java Tutorials
http://docs.oracle.com/javase/tutorial/

. API library
http://docs.oracle.com/javase/7/docs/api/

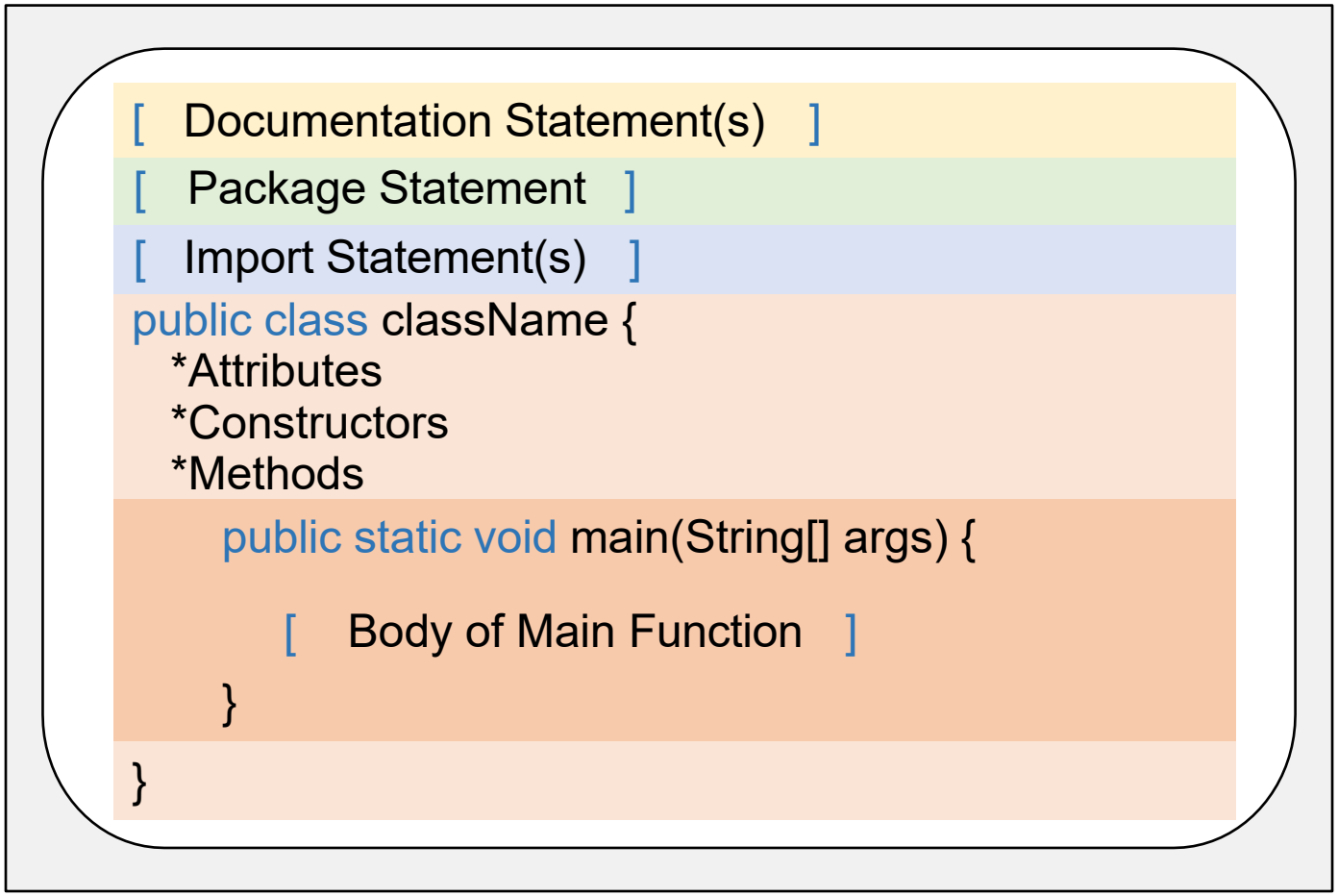
Java Revision

Programming 1:

- Java overview
- Data Types
- Variables
- Arithmetic operations
- Input and Output
- Strings
- Control flow statements
- Methods
- Program Component
- Arrays

Revision: Basic Java Program Structure

- There are 5 components that we usually include in a Java program, which are:

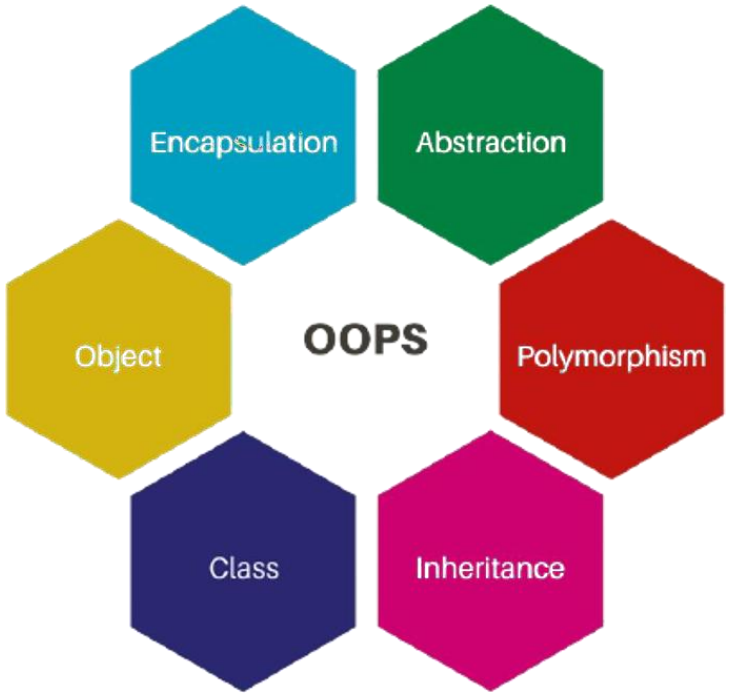


JAVA: An Introduction to Problem Solving & Programming, 8th Ed. By Walter Savitch
ISBN 0134462033

Object Oriented Programming (OOP)

- Java is an Object-Oriented Language.

- OOP Concepts:
  1. Class
  2. Object
  3. Instantiation
  4. Encapsulation
  5. Inheritance
  6. Polymorphism
  7. Abstraction

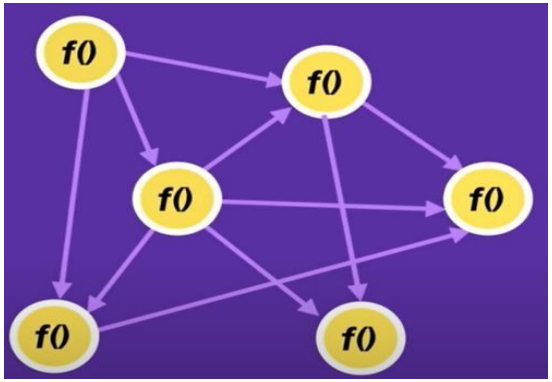


Motivation

- **Programming Paradigms** refer to the different **ways or styles** used to write and organize the code.

- Before *OOP,* we had *Procedural Programming* that divided a program into a set of functions. So, we have data stored in a bunch of variables and functions that operate on the data.

- But as your program grows, you will find yourself coping and pasting lines of code over and over, you will make a change to one function and then several other functions break, that will call ***"Spaghetti code"***.
  - There is so much interdependency between all these functions, It becomes problematic..



Motivation (Example)

First: what is an Object??

- In **primitive data types** we store just a single value.
- In **arrays** we store multiple values but all with the same data type and belong to one variable.
- **We need a way to collect number of variables that related to each other into a unit** in order to **describe a specific thing**.

- **For example:** if we have three cars and each one has a name and color and so on. So, we will end up with:

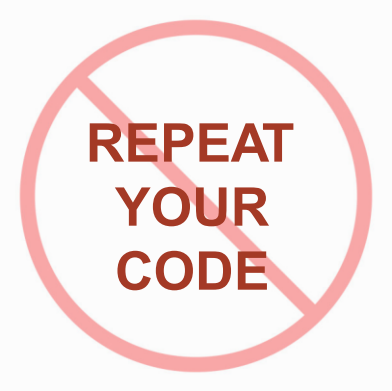
car1Name = "";
car1color = "";

car2Name = "";
car2color = "";

car3Name = "";
car3color= "";
:
:

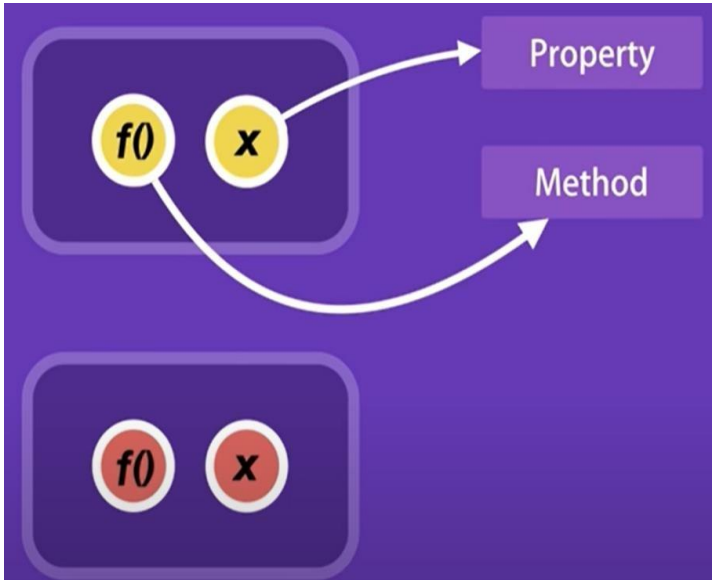
Imagine if we have hundred of cars !!!!!

Motivation (Solution)



- **OOP** came to solve this problem..

- We can deal with each car as a unit and grouping all the variables that describe that car together in one place.
- In OOP, we combined a group of related variables and functions that operate on them into a **unit**, we call that unit "**an object**".
- We refer to these variables as property and functions as methods.
- An object is a **black box** some part of which is hidden from the user (information hiding)

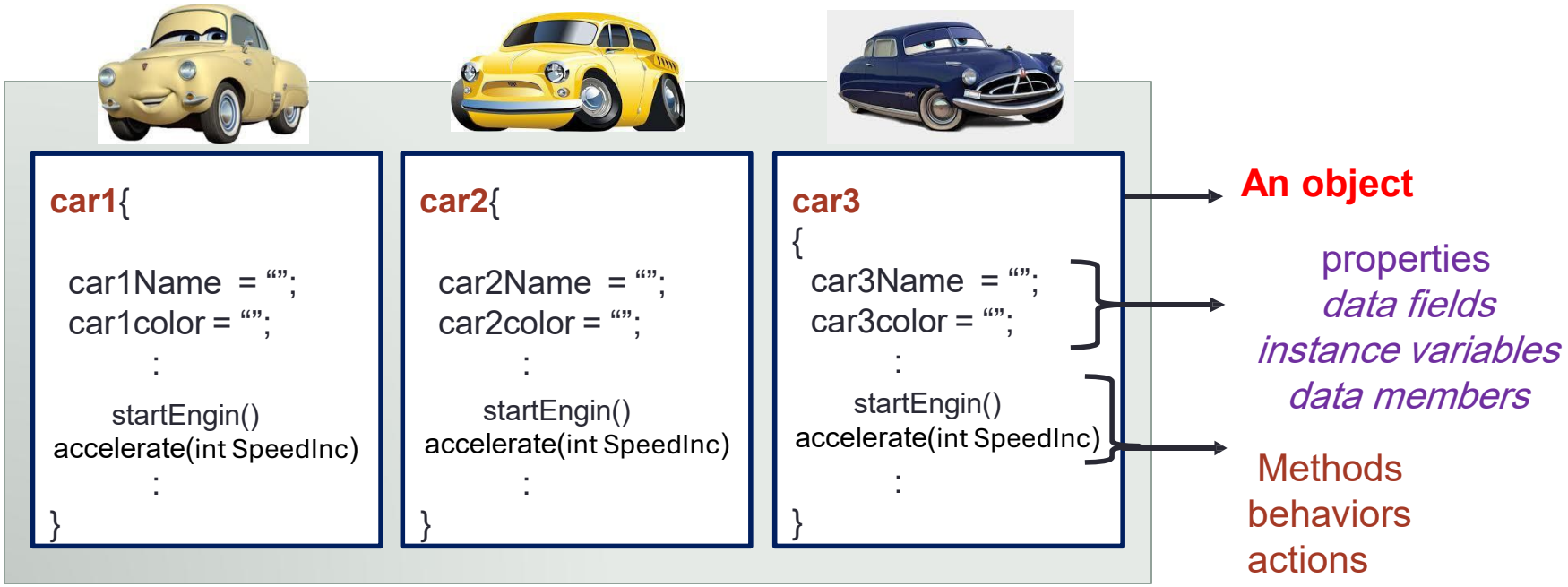


https://www.youtube.com/watch?v=pTB0EiLXUC8

Object

- An object represents an entity in the real world that can be distinctly identified. **For example**, a student, a desk, a circle, a button, and a car, all be viewed as objects.

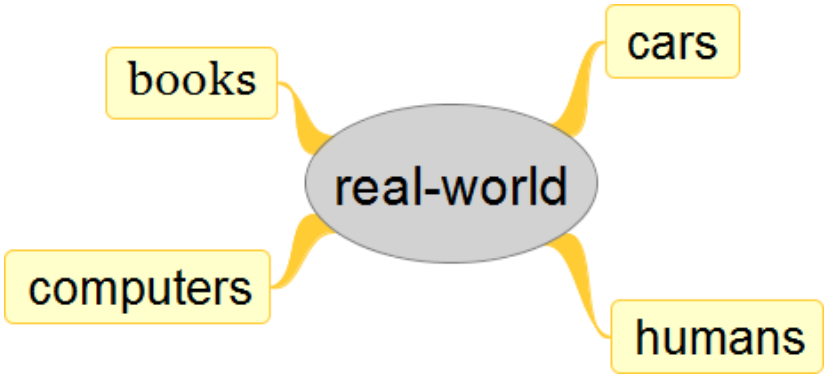
- An object has a unique identity, state, and behavior.



OOP RULES

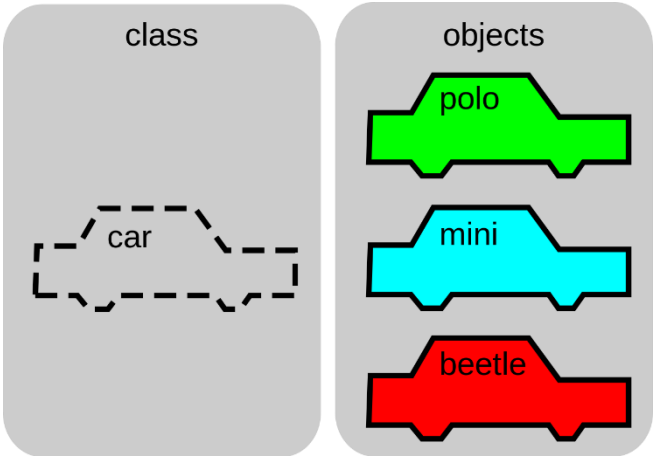
- **Rule1:** Everything is an OBJECT, e.g. car, book, etc.

- **Rule2:** Every OBJECT consists of:
- 1- properties (attributes)
- 2- actions (methods)



Class

- A **class** is a **User Defined Data Type**, You can use it to declare object reference variables.
- A **class** is blueprint/template for creating different objects which defines its properties and behaviours. It is used to create instances of itself.
- The **class contains** *data fields* or *instance variables* called *data members* and *methods* that represents the behaviour of this class or how it interact with world.



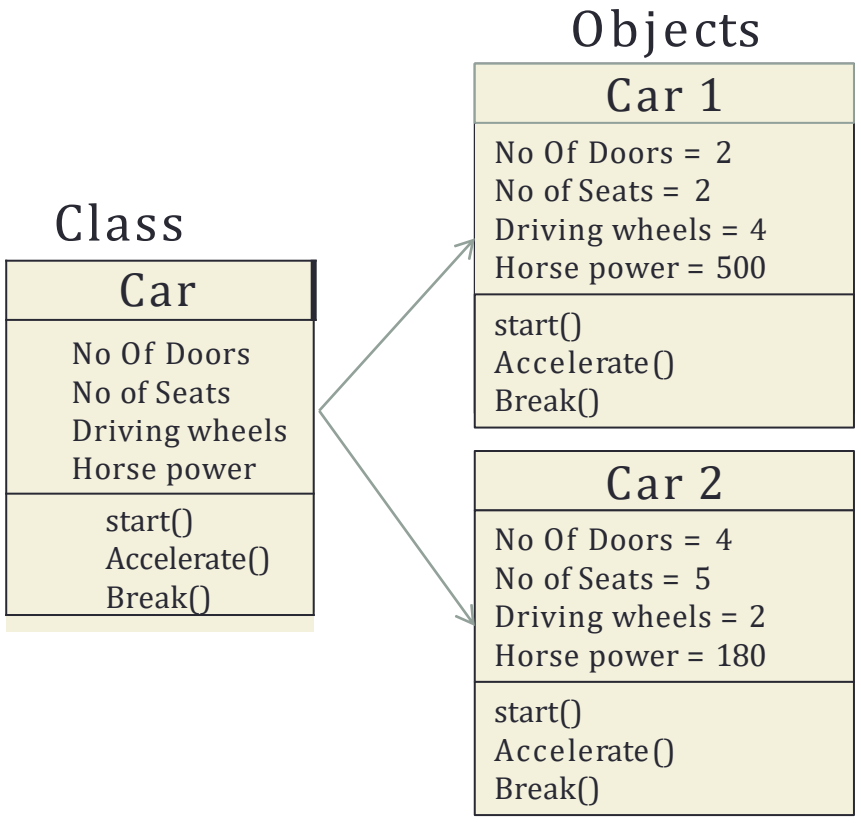
Classes and Objects

- **Objects** are considered instantiations of ***classes.*** We instantiate a class to create an object, which is a **concrete** instance of the class.

- **Different objects** of the same class have the same fields and methods, but the values of the fields will in general differ.
- **For example,** all people have eye color; but the color of each person's eyes can be different from others.

- **Java** is an OOP (deals with classes and objects)

- We need class's methods to deal with the values of class's variables. Its is safer to use these methods to do some **validations before we change any values of these variables**. Direct change of the values is not preferred.

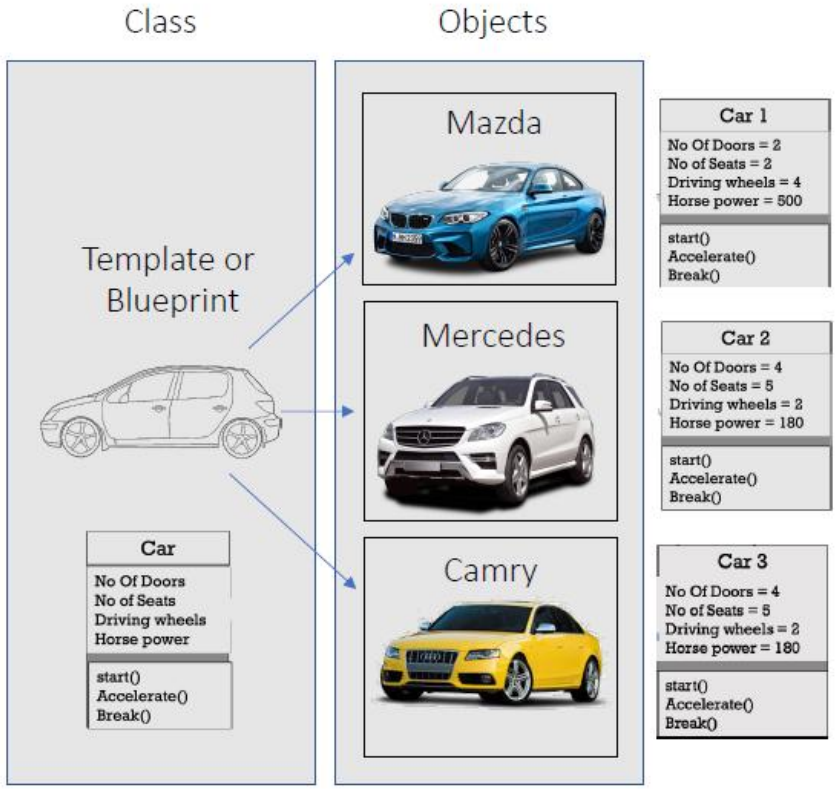


Class vs. Object

- The class has no values.
- Every object is an instance of a class.
- The objects have values.
- The class creates once, and then we have many objects of that class.
- There is no object without a class.

• Example:

- Class: Human -> Objects: Man, woman, …
- Class: Fruit -> Objects: Apple, Mango, …
- Class: Mobile -> Objects: Iphone, , Samsung, …



https://www.youtube.com/watch?v=yy3yLGkuXPk

Class Structure

```
public class class_name {
        list of attributes
        member methods
}
```

Class name:

- The class can be given any name by the programmer as long as :
  - It does not have spaces.
  - Start with a letter (no numbers or special characters are allowed at the beginning of the name)
  - Usually, but not necessary, the class start with a **capital** letter.

Attributes Structure

Modifiers Data_type Attributes_name;

```
public int x;
private double y;
public string z;
```

Modifiers Data_type Attributes_name = intial_value;

```
public int x=0;
privet double y=1.0;
public string z =‘’;
```

Method Structure

```
Modifiers Retrun_value_data_type method_name (list of parameters)
{
        / /  method body
}
```

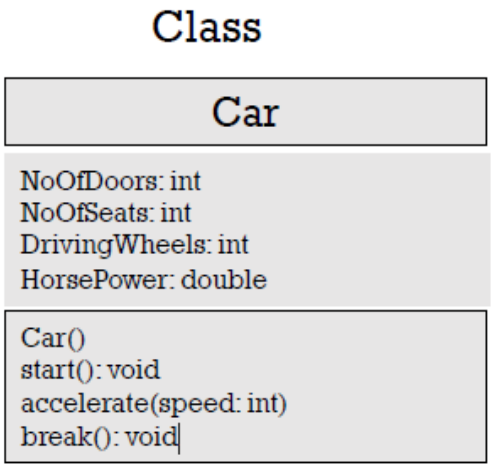
Example:-

```
public void setX()
{
/ /  method body
}
```

```
privet int getX()
{
/ /  method body
}
```

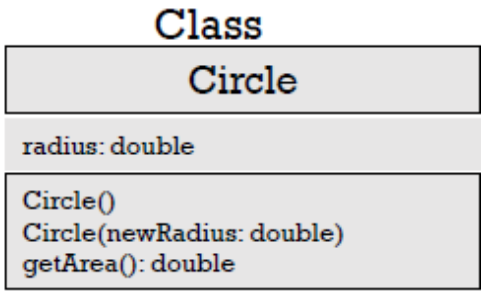
Class Example : Car

```
Public class Car{

    /*1- Properties (also called attributes, fields,
    instance variables, data fields, data members or
    states) */
    int NoOfDoors;
    int NoOfSeat;
    int DrivingWheels;
    Double HorseBower;

    /*2- Constructors are a special kind of methods that
    are invoked to construct objects*/
    Car() {}

    /*3-  Methods (also called actions, behaviors or
    functions)*/
    void accelerate (int speed){}
    void start() {}
    void break () {}

}
```

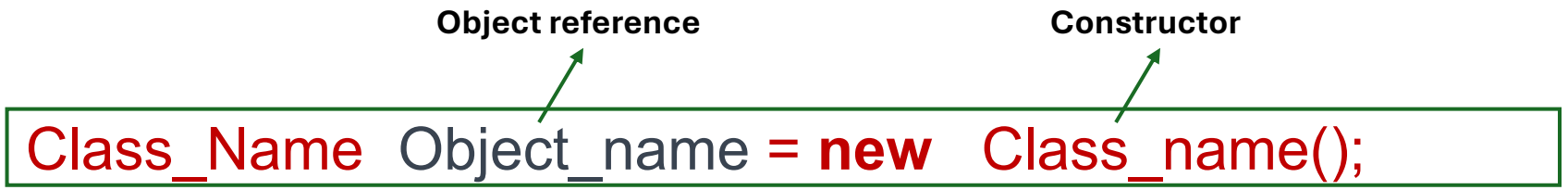


Class Example : Circle

```
Public class Circle{

    /*1- Properties (also called attributes, fields,
    instance variables, data fields, data members or
    states,  ) */
    double radius = 1.0;

    /*2- Constructors are a special kind of methods that
    are invoked to construct objects.*/
    Circle() {
    }
    Circle(double newRadius) {
      radius = newRadius;
    }

    /*3-  Methods (also called actions, behaviors or
    functions)*/
    double getArea() {
      return radius * radius * 3.14159;
    }

}
```



How to create an object



- Examples:
  - Car corola= new Car();
  - A a = new A();

The ***new*** keywords instantiates a class (creating an "instance" of a class that represents an object).

The ***new*** is one of the java's reserved keywords.

Handling an object

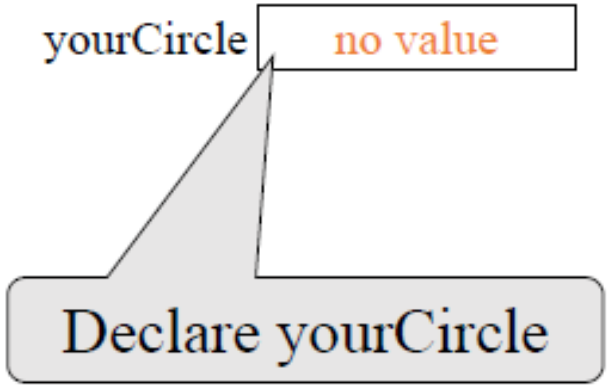
- An object is an instance of a class associated with all the values of its attributes.
- As an instance of the class all the visible attributes and methods are included within the object.
- **They can be reached using '.' operator.**

❑ **Referencing the object's data:**
`objectRefVar.data`
e.g., `myCircle.radius`

❑ **Invoking the object's method:**
`objectRefVar.methodName(arguments)`
e.g., `myCircle.getArea()`

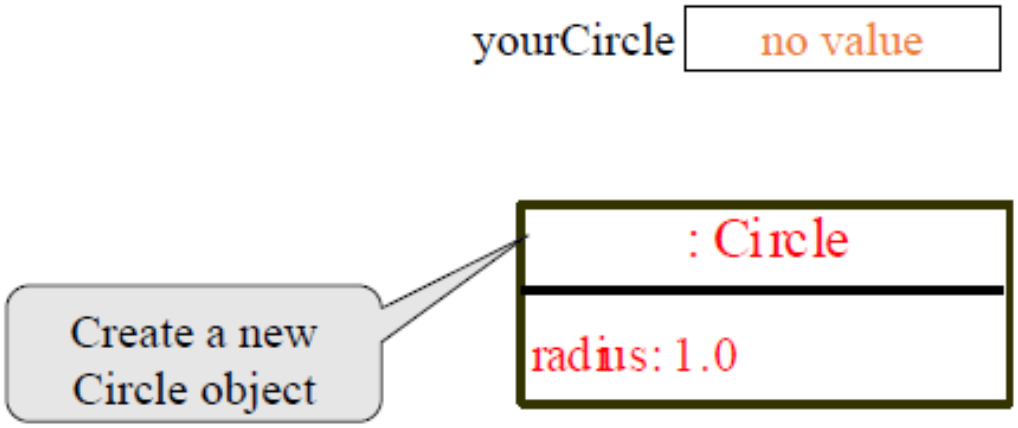
Trace code

```
Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



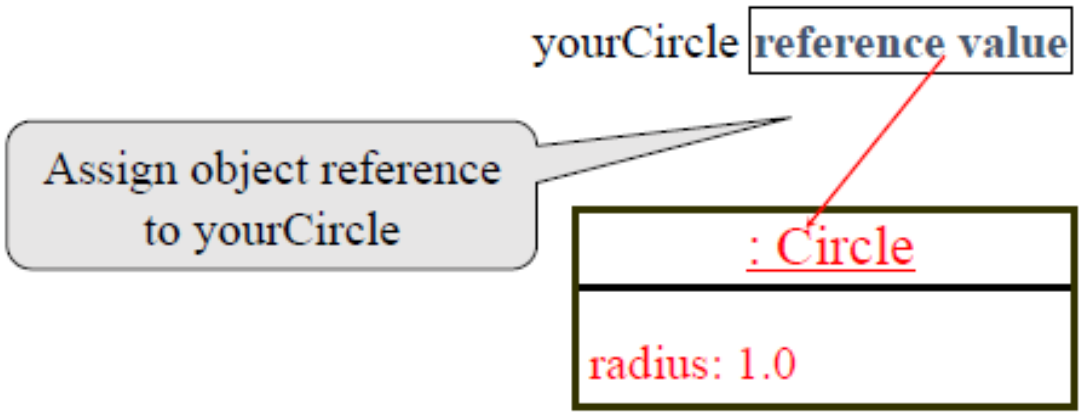
Trace code

```
Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Trace code

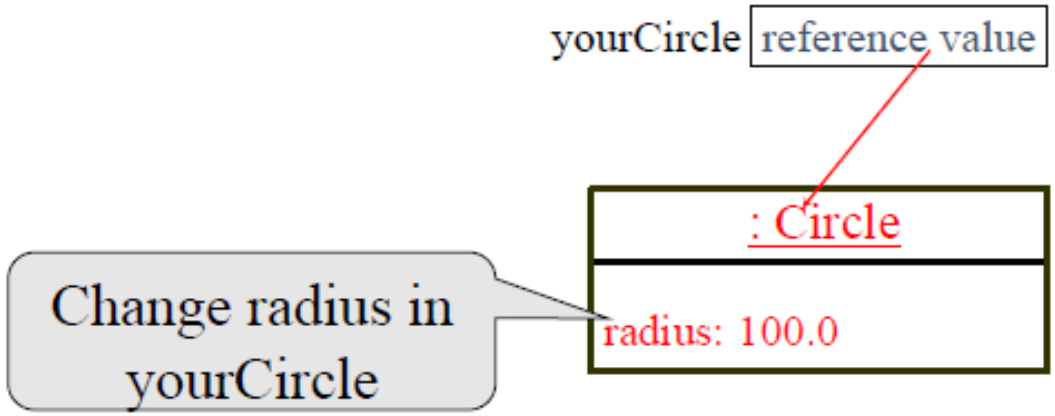
```
Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Trace code

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```



Handling an object: Example

1 Write your class

```
class Dog {

  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Ruff! Ruff!");
  }
}
```

instance variables

a method

2 Write a tester (TestDrive) class

just a main method (we're gonna put code in it in the next step)

```
class DogTestDrive {
  public static void main (String[] args) {
    // Dog test code goes  here
  }
}
```

Handling an object: Example

3 In your tester, make an object and access the object's variables and methods

```
class DogTestDrive {
   public static void main (String[] args) {
      Dog d = new Dog();
      d.size = 40;
      d.bark();
   }
}
```

make a Dog object

use the dot operator (.) to set the size of the Dog

and to call its bark() method

dot operator

The Dot Operator (.)

The dot operator (.) gives you access to an object's state and behavior (instance variables and methods).

```
// make a new object
Dog d = new Dog();

// tell it to bark by using the
// dot operator on the
// variable d to call bark()
d.bark();

// set its size using the
// dot operator
d.size = 40;
```

Handling an object: Example

```java
public class Student {
    public String name;
    public String ID;

    public void setName(String n)
    {   name=n;     }

    public String getName()
    {   return name;     }

    public void print()
    {System.out.println("Name:  "+name);}

}
```

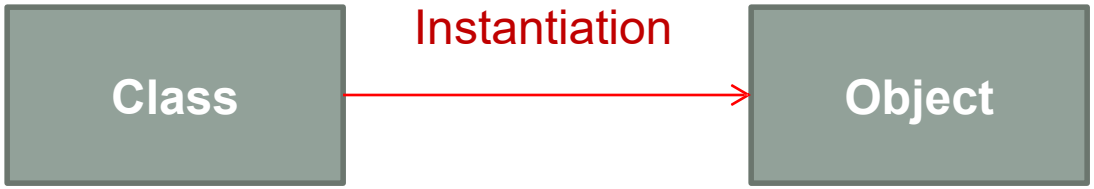
```java
public class TestStudent {

    public static void main(String[] args) {

        Student s1 = new Student();
        System.out.println(s1.name);
        s1.setName("Ahmad");
        s1.print();
    }

}
```

```
run:
null
Name: Ahmad
BUILD SUCCESSFUL (total time: 0 seconds)
```

Approximate Terminology

- instance = object
- field = instance variable=attributes= proprieties = data field
- method = function = Behavior
  - sending a message to an object = calling a method = invoke a method

Instantiating a Class



- The phrase "***instantiating a class***" means the same thing as "***creating an object***." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

- **Instantiation**: The *new* keyword is a Java operator that creates the object.

Initialisation

- Default values for **fields(**instance variables**)** that are not initialized explicitly:-

`null` for a reference type, 0 for a numeric type, `false` for a Boolean type, and '\u0000' for a **`char`** type.

- However, Java **assigns no default value to a local variable** inside a method.

```
public class Test {
  public static void main(String[] args) {
    int x;    // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

Compile error: variable not initialized

- Can destroy existing objects by assigning the keyword ‘`null`’
  - `NewCircle circle = new NewCircle(2.0);`
  - `circle = null;`

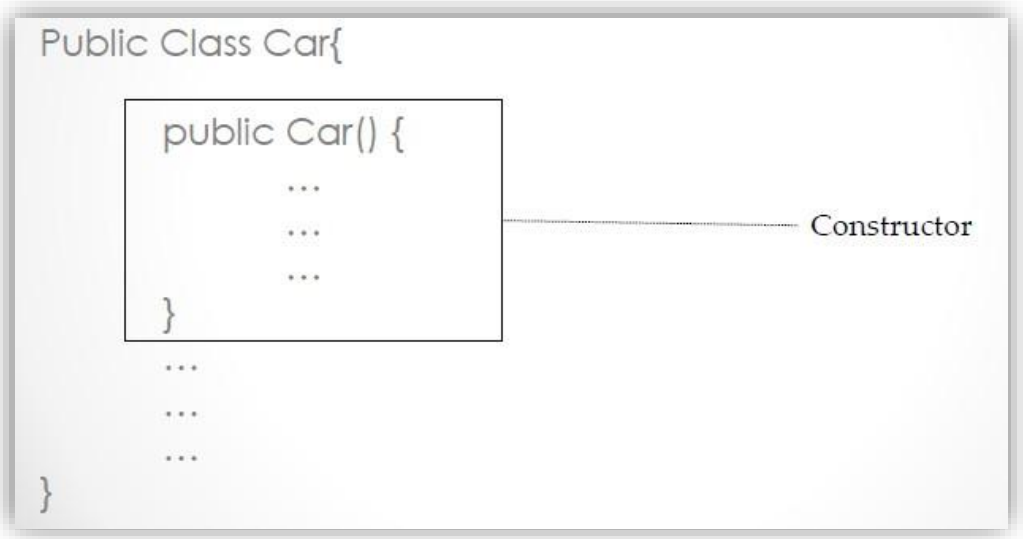
- The memory space that was occupied by the object ‘`circle`’ is then garbage collected by JVM

Constructor

- It is a **special kind of methods** that are invoked to **construct Objects** (set initial values for field variables when the object is created).

- You don't need to make any special calls to a constructor method - they happen automatically when you create a new object.

- It prepares the new object for use.



Constructor

- The **constructor** is a special method that creates an object that:
  - Does not have a return value or return value type.
  - Must have the exact same name as the class
  - May or may not have arguments.

- If the class does not have a constructor, then a **default empty constructor** will be added to it by java compiler. In this case, a no-argument constructor with an empty body is implicitly defined in the class which is provided automatically only if no constructors are explicitly defined in the class

- The default constructor initializes any uninitialized instance variables with default values.
- The constructor will be called each time a new object is created which is every time the keyword *new* is used. The complier to know which constructor to call by matching the parameters.
- Overloading--the same class can have more than one constructor all with the same name but with different arguments, making it easy to construct objects with different initial data values.

Constructor

Feature	Default Constructor	No-Arg Constructor	Parameterized Constructor
Created By	Java Compiler	Programmer	Programmer
Arguments	None	None	One or more
Purpose	Provide default values (0, null)	Provide custom fixed values	Provide specific values per object
Control	No control over logic	Full control over logic	Full control over logic

Constructor: Example 1

```
Public class Example1{
        int x;
        int y;

        public Example1(){
        }
}
```

No-argument constructor

```
Example1 ex1 = new Example1();
```

Constructor: Example 2

```
Public class Example1{
        int x;
        int y;

        public Example1(){
                x = 0;
                y = 1;
        }
}
```

Initial values

```
Example1 ex1 = new Example1();
```

Constructor: Example 3

```
Public class Example1{
        int x;
        int y;

        public Example1(int a, int b){
                x = a;
                y = b;
        }
}
```

Parameterized constructor

```
Example1 ex1 = new Example1(0,1);
```

Constructor: Example 4

```
Public class Example1{
        int x;
        int y;

        public Example1(){
                x = 0;
                y = 1;
        }

        public Example1(int a, int b){
                x = a;
                y = b;
        }
}
```

Constructor overloading

Having more than one constructor with different parameter lists. like Java methods.

```
Example1 ex1 = new Example1();

Example1 ex2 = new Example1(0,1);
```

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Test it Now

Output:

```
Bike is created
```

https://www.javatpoint.com/java-constructor

```
//Let us see another example of default constructor
//which displays the default values
class Student3{
int id;
String name;
//method to display the value of id and name
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
}
}
```

Explanation: In this class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Test it Now

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

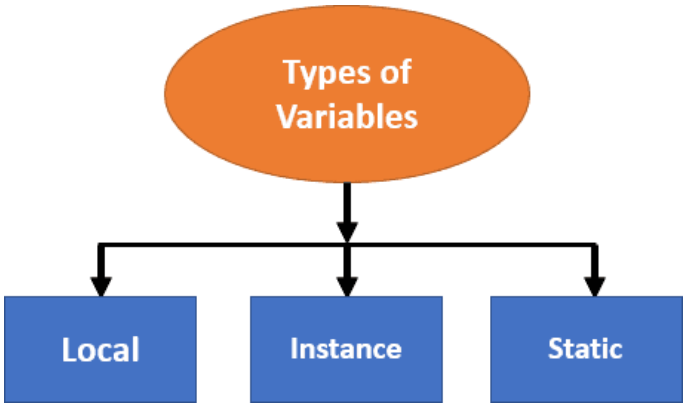
Output:

```
0 null
0 null
```

https://www.javatpoint.com/java-constructor

Types Of Variables

- **There are three types of variables in java:**
  - Local Variables (non–static variables)
    Declared within the body of a method.
  - Global variables
    - Instance Variables (non–static variables)
    - class Variables (static variables)




```
public class Test {

    static int a=100; //static variable
    int b=50; //instance variable

    void method() {
        int n=90; //local variable
    }

   }//end of class
```

The keyword "this"

```
public class Student {
    long stdID;
    int noOfCourses = 0;

    public void setstdID(long stdID){
        this.stdID = stdID;
    }

    public void setstdID(int noOfCourses){
        this.noOfCourses = noOfCourses;
    }
}
```

Global variables

Local variables

- "This" makes it possible **to distinguish between two variables with the same name** by their position as global variables and local variables.
- "This" is a **keyword** that is used in instance methods as a pointer to refer to the current object.
- In the above example the word **this** referred to a virtual object from the type Student and hence **any variable that is reached using "this." is a global variable.**

this keyword with Field (Instance Variable)

- `this` keyword can be very useful in the handling of **Variable Hiding**.
- We can not create two instance/local variables with the same name.
- It is legal to create one instance variable & one local variable or Method parameter with the same name.In this scenario the local variable will hide the instance variable this is called **Variable Hiding**.

Other Uses of “this” Keyword

- Within an instance method or a constructor:

  you can also use the ”this” keyword to call another constructor in the same class (calling overloaded constructor) , this is called “*explicit constructor invocation*”.

```
public class Rectangle {

private int x, y;
private int width, height;

public Rectangle()
 {{this(0, 0, 1, 1); }

public Rectangle(int width, int height)
{ this(0, 0, width, height); }

public Rectangle(int x, int y, int width, int height)
{ this.x = x;
  this.y = y;
  this.width = width;
  this.height = height; }

   ... }
```

https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html

Classes Members vs. Instance Members

Instance variables	Static (class) variables
Declared without the static modifier	Declared with the static modifier
declared in a class, but outside a method, constructor or any block.	declared in a class, but outside a method, constructor or a block.
Associated with objects not to the class, every individual object created from the class has its own copy of these variables.	Belong to a class, rather than to an instance and shared by all instances of the class. So, there would only be one copy of each class variable per class, regardless of how many objects are created from it.
Called with an object *ObjectReference.VariableName*.	Accessed by calling with the classname *ClassName.VariableName*.
Created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.	Static variables are created when the program starts and destroyed when the program stops.

Static (Class) Variables Example

```
public class StaticExample {
     static int count =0; // Static variable

    public StaticExample() {
        count++;
    }

    public static void displayCount() {
        System.out.println("Count: " + count);
    }

    public static void main(String[] args) {
        StaticExample obj1 = new StaticExample();
        StaticExample obj2 = new StaticExample();
        StaticExample.displayCount(); // Output: Count: 2
    }
}
```

Instance Variables Example

```
// INSTANCE VARIABLE
public class Main {

    public static void main(String[] args) {

        Product prod1 = new Product();
        prod1.Barcode = 123456;

        Product prod2 = new Product();
        prod2.Barcode = 987654;

        System.out.println(prod1.Barcode);
        System.out.println(prod2.Barcode);
    }
}

public class Product {
    public int Barcode;
}
```

The output will be:

```
123456

987654
```

Static (Class) Methods vs. Instance Methods

Instance Method

- Instance methods is called when an object of its class is created.
- Instance methods is not created with static keyword.
- instance methods exist as multiple copies depending on the number of instances created for that class.
- Instance method can access static variables and static methods directly.

Static Method

- Static methods are the methods in Java that can be called without creating an object of class.
- Static method is actually declared with static keyword.
- Static method means which will exist as a single copy for a class.
- Static methods can't access instance methods and instance variables directly.

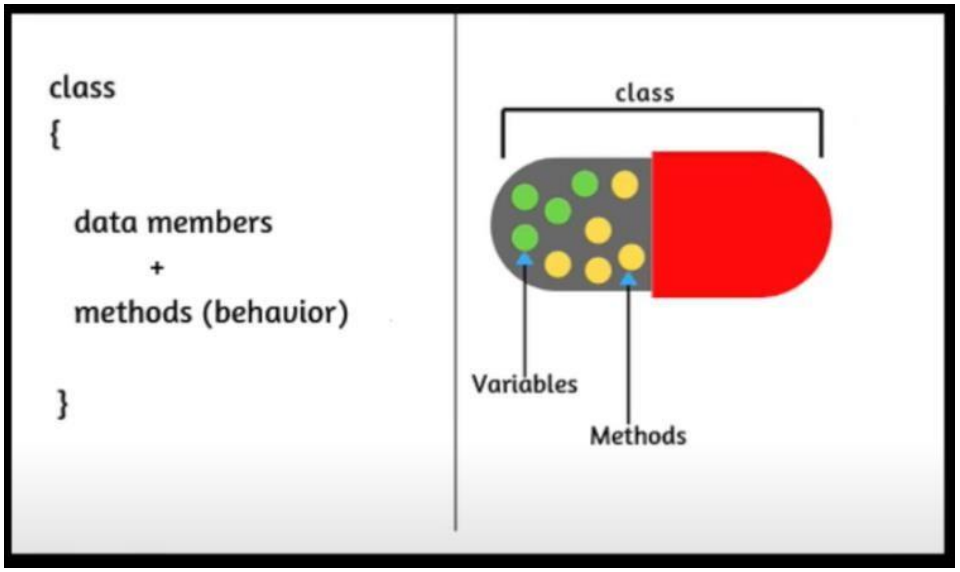
Static (Class) Methods vs. Instance Methods

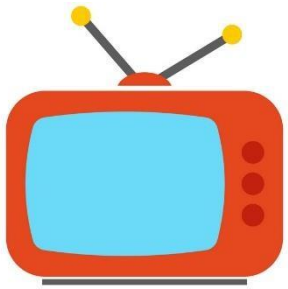
Not all combinations of instance and class variables and methods are allowed:

- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the `this` keyword as there is no instance for `this` to refer to.

Encapsulation

- Encapsulation is a process of **wrapping** data (variables) and methods together into a single unit.
- Encapsulation is also known as "**Data hiding**" because they are protecting data which is prone to change.
- Encapsulation is the technique of **making the fields** (variables) **in a class private and providing access to the fields via public methods.**
- In encapsulation, the **variables** of a class will **be hidden from other classes**, and **can be accessed only through the methods of their current class.**



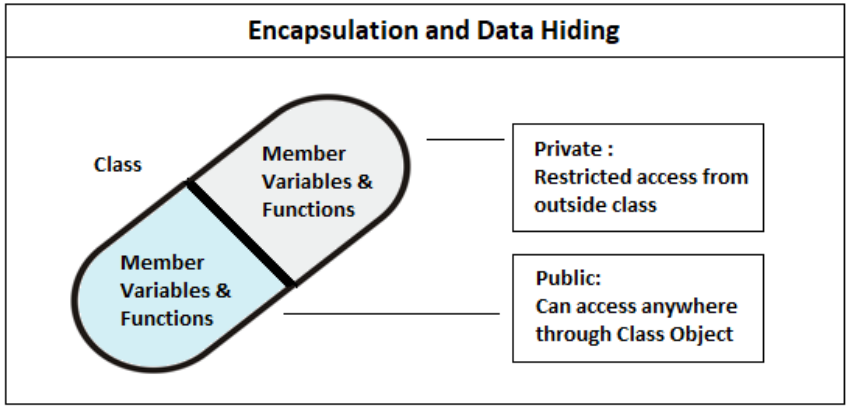


Encapsulation

- **To achieve encapsulation in Java :–**
  - Declare the variables of a class as private.
  - Provide public setter and getter methods to **modify** and **view** the variables values.

- **Benefits of Encapsulation:-**
  - The fields of a class can be made **read-only** or **write-only**.
  - A class can have **total control** over what is stored in its fields.

Data Hiding VS. Encapsulation



Encapsulation	Data Hiding
It means binding data members and methods in a single unit (class)	It means declaring data variables as private to secure it from unauthorized access
It focuses on hiding the complexity of the system	It focuses on data security

Special Methods: Set and Get Methods

- ***Set*** and ***Get*** methods are a **pattern of data encapsulation** that allow controlled access to class data.
- Instead of accessing class variables directly, you define *getters* methods, and *setters* methods.
- We can use *get* and *set* methods of the current class to access its variables from another class.

- Setters/Mutators are methods that (only) **modify** ("mutate") a class's fields
  - Use setters to validate data before changing the object state

- Getters/Accessors are methods that (only) access fields but may not modify a class's fields to return information about the state of an object
  - Use getters to view/ format data before returning the object's state

Special Methods: Set and Get Methods

Commonly, a field has two associated methods: a mutator for setting the value, and an accessor for getting the value and typically with names starting with set or get.

Set methods — return type `void`

Get methods — specified return type

```
public class Person{
private int age;

public void setAge(int a) {
  // validate here
  if(a>0 && a<100)
   this.age = a;
}

public int getAge() {
  // format here
  return this.age;
} }
```

References

- Slides are modified from: Dr. Atif Alhejali's lectures & Dr.Manal's lectures and other references
- https://www.youtube.com/watch?v=qP9-3LnMZsE
- https://www.youtube.com/watch?v=pTB0EiLXUC8&t=75s
- https://learn.zybooks.com/