

# **Unit 3**

# **Object Oriented Programming**

# **Concepts**

# Object Oriented Programming

- ❖ Object-oriented programming (OOP) is a programming paradigm based on the concept of objects and classes to organize the data

## Objects

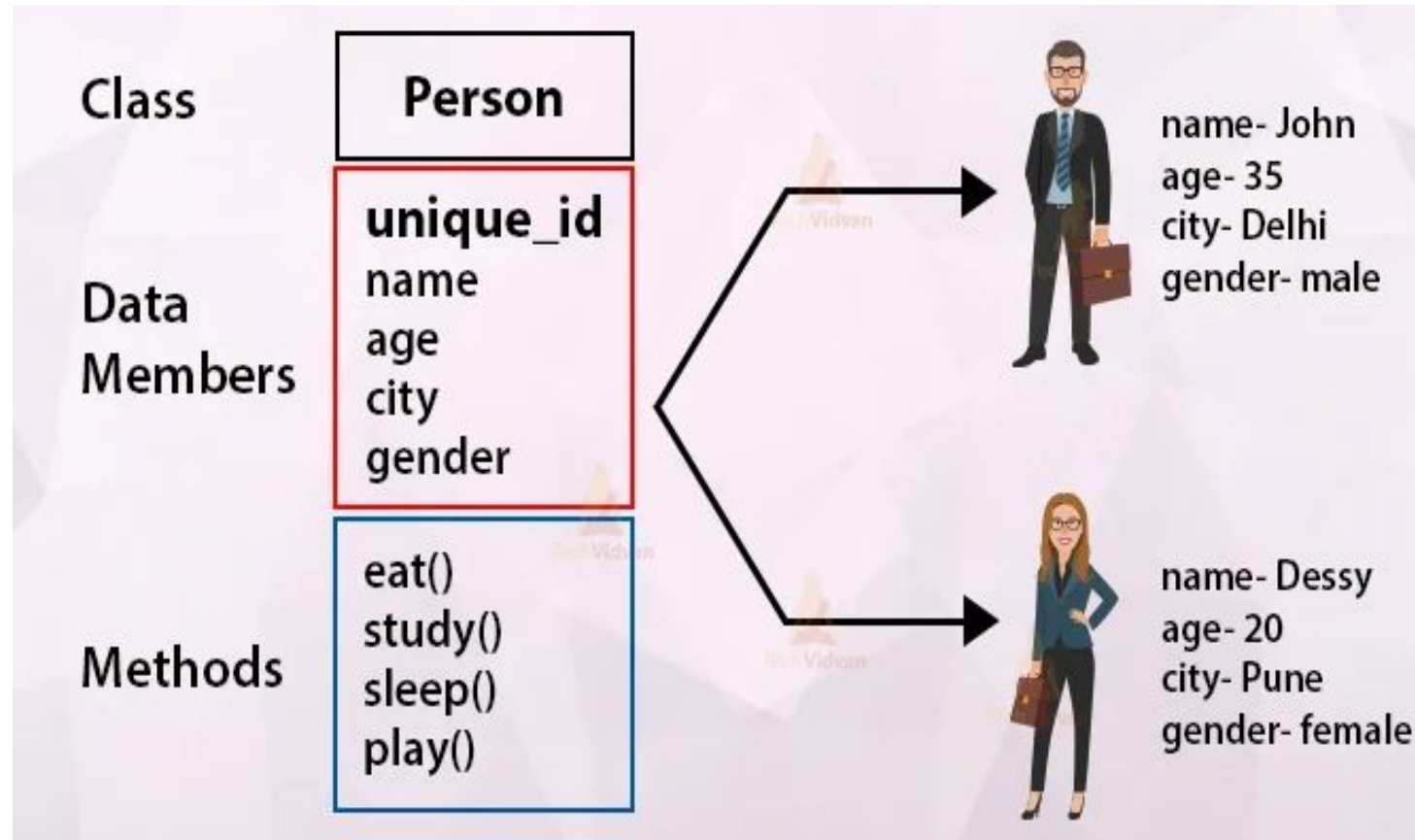
- ❖ Objects are the real world entities having:
  - **State:** It is represented by attributes of an object. It also reflects the properties of an object.
  - **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
  - **Identity:** It gives a unique name to an object and enables one object to interact with other objects.
- ❖ For example: bike is an object. It has:
  - ❖ State: gear, speed
  - ❖ Behavior: braking, accelerating, changing gears, etc.

# Object Oriented Programming

## Class

- ❖ A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.
- ❖ Collection of objects is called class. It is a logical entity.
- ❖ A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.
- ❖ We can think of the class as a sketch(prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these description, we build the house. The house is an object.

# Object Oriented Programming-Class and Object



# Java Class and Objects

- ❖ We can create a class in Java using following syntax:

```
class className{  
    fields or variables or  
    dataMembers or attributes  
    methods  
}
```

- ❖ Here fields (variables) and methods represent the state and behavior of the object respectively
  - Fields are used to store data
  - Methods are used to perform some operations on fields

- ❖ As an example, we can create Bike class as:

```
class Bike{  
    int speed =50;  
    public void brake(){  
        System.out.println("Applying  
        Brakes");  
    }  
}
```

# Java Class and Objects

- ❖ We have created the blueprint for bikes i.e. the Bike class. Now we can use it to create any number of bikes.
- ❖ All bikes made using this blueprint will share the fields and methods of the blueprint.
- ❖ An object is instance of a class.
- ❖ We can create objects of class Bike as follows:  
`Bike dirtBike = new Bike();`
- ❖ Here, dirtBike is an object of class Bike and it will have all the fields and methods described in the class.
- ❖ We can create multiple objects from same class as well.  
`Bike dirtBike = new Bike();`  
`Bike sportsBike = new Bike();`  
`Bike cruiserBike = new Bike();`

# Java Class and Objects-Accessing Members of a Class

- ❖ We can use the name of objects along with the . operator (dot operator) to access the members of a class.

```
class Bike{
    //state or fields
    int speed =50;

    //behavior or method
    public void brake() {
        System.out.println("Applying brakes");
    }

    public static void main(String[] args) {





        //create object
        Bike dirtBike = new Bike();

        //access fields and methods
        System.out.println("Speed:"+dirtBike.speed+"kmph");
        dirtBike.brake();
    }
}
```

# Java Class and Objects-Accessing Members of a Class

```
public class Bike {  
    int speed =40;  
    public void brake(){  
        speed-=10;  
        System.out.println("Current Speed is "+ speed+" kmph");  
    }  
    public void accelerate(){  
        speed+=10;  
        System.out.println("Current Speed is "+speed+" kmph");  
    }  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        Bike myBike = new Bike();  
        System.out.println("Initial Speed is "+ myBike.speed+" kmph");  
        myBike.brake();  
        myBike.brake();  
        myBike.accelerate();  
        myBike.accelerate();  
        myBike.accelerate();  
        System.out.println("Final Speed is "+ myBike.speed+" kmph");  
    }  
}
```

| Output - ClassObjectExample (run) ×  | Exception Reporter |
|--|--------------------|
|  run:<br> Initial Speed is 40 kmph<br> Current Speed is 30 kmph<br> Current Speed is 20 kmph<br>Current Speed is 30 kmph<br>Current Speed is 40 kmph<br>Current Speed is 50 kmph<br>Final Speed is 50 kmph<br>BUILD SUCCESSFUL (total time: 1 second) |                    |



# Java Access Modifier/Specifier or Java Access Control

- ❖ In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter method.
- ❖ There are four types of Java access modifiers:
  - **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
  - **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
  - **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
  - **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

# Java Access Modifier-Example

- ❖ In example aside, burnPetrol() method is made private that means it cannot be accessed by other classes.
- ❖ From MainClass class, we cannot directly call burnPetrol() via object of Bike class.
- ❖ However, brake() and accelerate() are public meaning they can be accessed by other classes.

```
public class Bike {  
    int speed =40;  
    public void brake(){  
        speed-=10;  
        System.out.println("Current Speed is "+ speed+" kmph");  
    }  
    public void accelerate(){  
        burnPetrol();  
        speed+=10;  
        System.out.println("Current Speed is "+speed+" kmph");  
    }  
    private void burnPetrol(){  
        System.out.println("Burning petrol");  
    }  
}
```

## Java Access Modifier-Java Default Access Modifier

- ❖ If we do not explicitly specify any access modifier for classes, methods, variables, etc., then **by default the default access modifier is considered.**

```
package defaultPackage;  
class Logger {  
    void message(){  
        System.out.println("This is a message");  
    }  
}
```

- ❖ Here Logger class has default access modifier.
- ❖ The class is visible to all the classes that belong to *defaultPackage* package.
- ❖ However if we try to use the *Logger* class in any class outside of the *defaultPackage*, we will get a compilation error.

# Java Access Modifier-Private Access Modifier

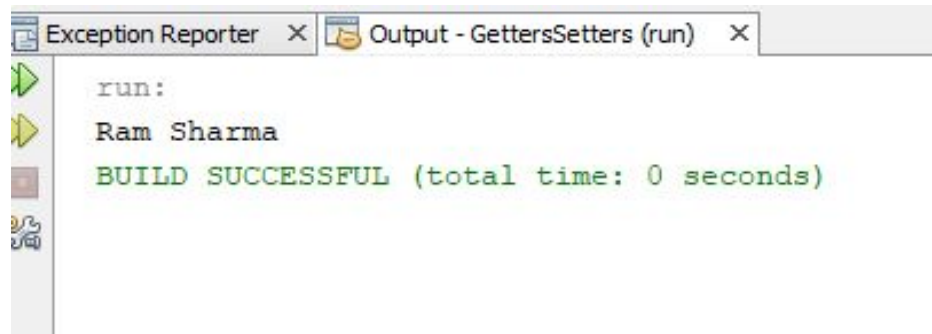
- ❖ When variables and methods are declared private, they cannot be accessed outside of the class.
- ❖ Classes cannot be private
- ❖ When we run the program aside, we will get an error.
- ❖ Note: We cannot declare class as private in Java

```
class Data {  
    // private variable  
    private String name;  
}  
  
public class Main {  
    public static void main(String[] main){  
  
        // create an object of Data  
        Data d = new Data();  
  
        // access private variable and field from another class  
        d.name = "Programiz";  
    }  
}
```

# Java Access Modifier-Private Access Modifier

- ❖ If we need to access those private variables, we can use the getters and setters method.

```
public class GettersSetters {  
  
    public static void main(String[] args) {  
        Student std1 = new Student();  
        std1.setName("Ram Sharma");  
        System.out.println(std1.getName());  
    }  
}
```




```
public class Student {  
  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

## Java Access Modifier-Protected Access Modifier

- ❖ When variables and methods are declared protected, they can be accessed within the same package as well as from its subclasses.
- ❖ Note: We cannot declare class as protected in Java

```
class Animal {  
    // protected method  
    protected void display() {  
        System.out.println("I am an animal");  
    }  
}  
  
class Dog extends Animal {  
    public static void main(String[] args) {  
  
        // create an object of Dog class  
        Dog dog = new Dog();  
        // access protected method  
        dog.display();  
    }  
}
```

Output



```
I am an animal
```

# Java Access Modifier-Public Access Modifier

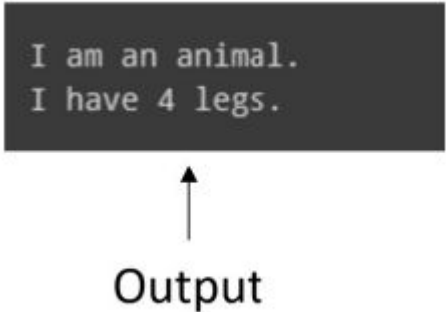
- ❖ When variables, classes, methods are declared public, they can be accessed from anywhere.

```
// Animal.java file
// public class
public class Animal {
    // public variable
    public int legCount;

    // public method
    public void display() {
        System.out.println("I am an animal.");
        System.out.println("I have " + legCount + " legs.");
    }
}

// Main.java
public class Main {
    public static void main( String[] args ) {
        // accessing the public class
        Animal animal = new Animal();

        // accessing the public variable
        animal.legCount = 4;
        // accessing the public method
        animal.display();
    }
}
```



```
I am an animal.
I have 4 legs.
```

Output

# Abstraction in Java

- ❖ Abstraction is the process of hiding the implementation details from the user, only the functionality will be provided to the user.
- ❖ In other words, the user will have the information on what the object does instead of how it does.
- ❖ In Java, abstraction can be achieved using Abstract Classes and Interfaces.
- ❖ **Real World Example:** Making coffee with a coffee machine is a good example of abstraction. You need to know how to use your coffee machine to make coffee. The thing you don't need to know is how the coffee machine is working internally to brew a fresh cup of delicious coffee. You don't need to know the ideal temperature of the water or the amount of ground coffee you need to use.



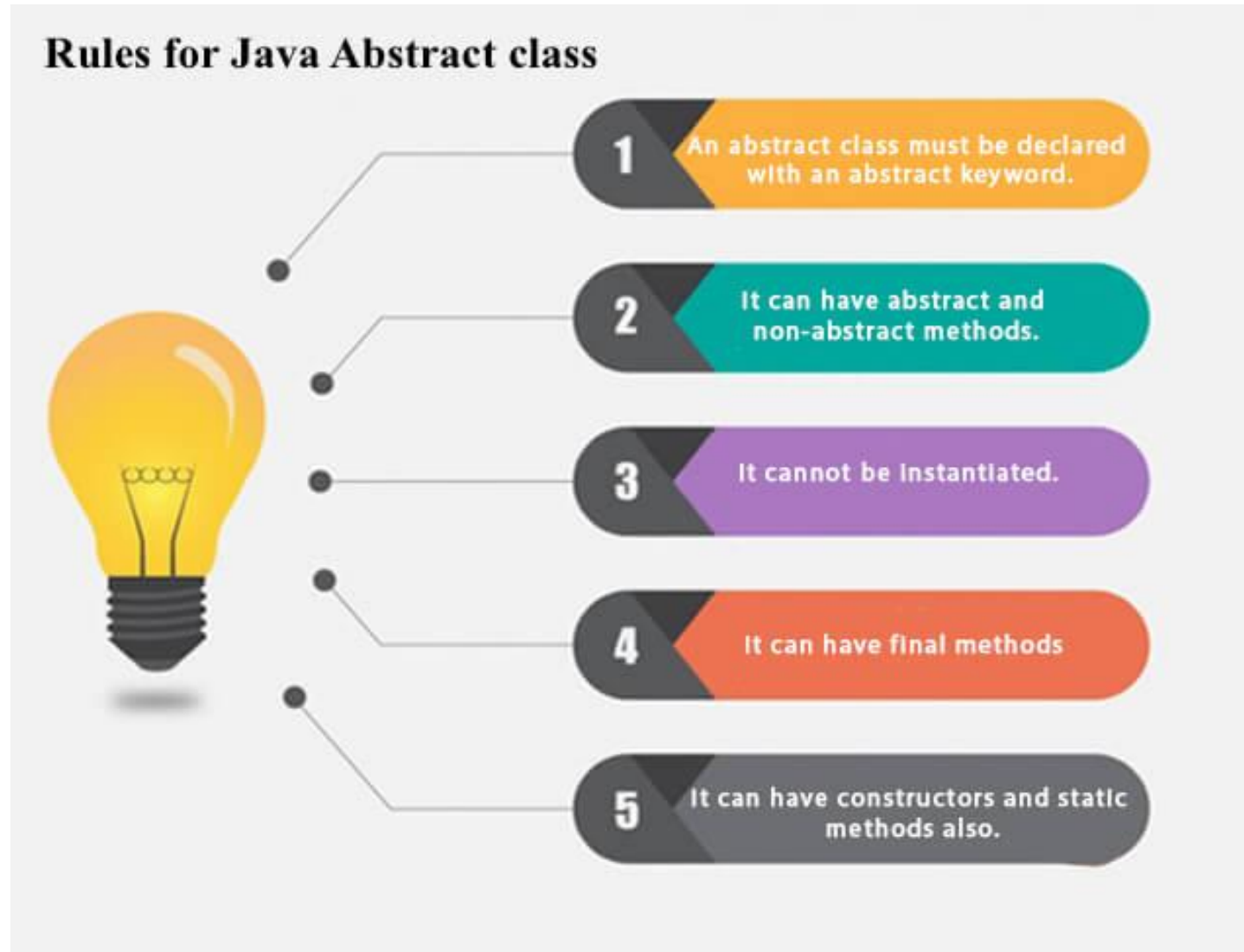
# Abstraction in Java

- ❖ Abstraction is the process of hiding the implementation details from the user, only the functionality will be provided to the user.
- ❖ In other words, the user will have the information on what the object does instead of how it does.
- ❖ In Java, abstraction can be achieved using **Abstract Classes** and **Interfaces**.
- ❖ **Real World Example:** Making coffee with a coffee machine is a good example of abstraction. You need to know how to use your coffee machine to make coffee. The thing you don't need to know is how the coffee machine is working internally to brew a fresh cup of delicious coffee. You don't need to know the ideal temperature of the water or the amount of ground coffee you need to use.

## Abstraction In Java-Abstract Class

- ❖ Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- ❖ A class which contains the abstract keyword in its declaration is known as abstract class.
- ❖ Abstract classes may or may not contain abstract methods
  - An abstract method is a method without body
- ❖ But, if a class has at least one abstract method, then the class **must** be declared abstract.
- ❖ If a class is declared abstract, it cannot be instantiated.
- ❖ To use an abstract class, we have to inherit it from another class, provide implementations to the abstract methods in it.

# Abstraction In Java-Abstract Class-Summary

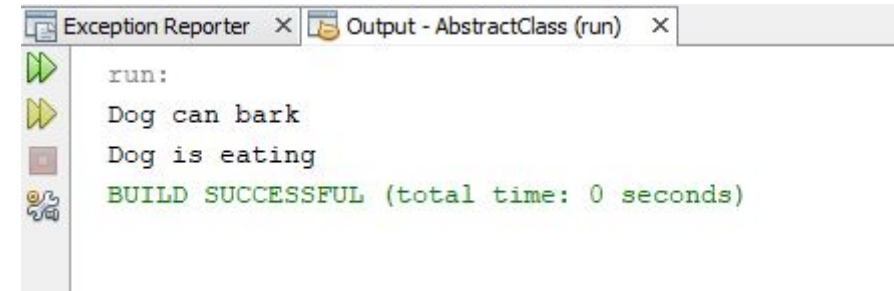


# Abstraction In Java-Abstract Class-Example

```
public abstract class Animal {  
    public abstract void makeSound();    //Abstract Method  
    public void eat(){                  //Non-Abstract Method  
        System.out.println("Dog is eating");  
    }  
}
```

```
public class AbstractClass {  
  
    public static void main(String[] args) {  
        Dog dog = new Dog();  
        dog.makeSound();  
        dog.eat();  
    }  
}
```

```
public class Dog extends Animal{  
  
    @Override  
    public void makeSound() {  
        System.out.println("Dog can bark");  
    }  
  
}
```



The screenshot shows a window titled "Output - AbstractClass (run)" with the following content:

```
run:  
Dog can bark  
Dog is eating  
BUILD SUCCESSFUL (total time: 0 seconds)
```

On the left side of the window, there are icons for running the program (a green play button), viewing the output (a magnifying glass), and other IDE functions.

## Abstraction In Java-Interfaces

- ❖ Another way of achieving abstraction in Java is with Interfaces.
- ❖ An interface in Java is a blueprint of a class.
- ❖ There can be only abstract methods in the Java interface, not the method body.
- ❖ Interfaces specify what a class must do and not how. It is the blueprint of the class.
- ❖ Interface specifies a set of methods that the class has to implement.
- ❖ To implement an interface, **implements** keyword is used.

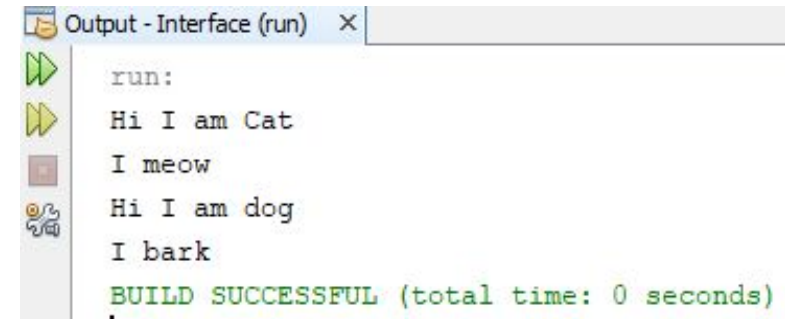
# Abstraction In Java-Interfaces

```
public interface AnimalInterface {  
  
    public void showType();  
    public void makeSound();  
  
}
```

```
public class Cat implements AnimalInterface{  
  
    @Override  
    public void showType() {  
        System.out.println("Hi I am Cat");  
    }  
  
    @Override  
    public void makeSound() {  
        System.out.println("I meow");  
    }  
  
}
```

```
public class Dog implements AnimalInterface{  
  
    @Override  
    public void showType() {  
        System.out.println("Hi I am dog");  
    }  
  
    @Override  
    public void makeSound() {  
        System.out.println("I bark");  
    }  
  
}
```

```
public class Interface {  
  
    public static void main(String[] args) {  
        Cat cat = new Cat();  
        cat.showType();  
        cat.makeSound();  
        Dog dog = new Dog();  
        dog.showType();  
        dog.makeSound();  
    }  
  
}
```



```
Output - Interface (run) x  
run:  
Hi I am Cat  
I meow  
Hi I am dog  
I bark  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Java This Keyword

- ❖ This keyword refers to the current object in a method or constructor.
- ❖ The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).
- ❖ This keyword can also be used to:
  - Invoke current class constructor
  - Invoke current class method
  - Return the current class object
  - Pass an argument in the method call
  - Pass an argument in the constructor call



# Java This Keyword-Example

```
public class ThisKeyword {  
    private int id;  
    private String name;  
    private String address;  
  
    public void setValues(int id, String name, String address){  
        this.id=id;  
        this.name=name;  
        this.address=address;  
    }  
    public void displayValues(){  
        System.out.println("Id is "+id+" Name is "+name+" Address is "+address);  
    }  
    public static void main(String[] args) {  
        ThisKeyword tk = new ThisKeyword();  
        tk.setValues(1,"Ram Bahadur","Kathmandu");  
        tk.displayValues();  
    }  
}
```

Output - ThisKeyword (run) X



run:



Id is 1 Name is Ram Bahadur Address is Kathmandu



BUILD SUCCESSFUL (total time: 0 seconds)





# Assignment

1. Differentiate between:
  - a. Class and Interface
  - b. Class and Abstract Class
  - c. Abstract Class and Interface

# Constructors In Java

- ❖ A constructor in Java is a **special method** that is used to initialize objects but doesn't have a return type.
- ❖ **Constructor's name must match the class name i.e. constructor name should be the name of class**
- ❖ The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.
- ❖ Syntax:

```
class className{  
    className(){  
        }  
}
```
- ❖ There are two types of constructors in Java:
  - a. Default constructor (no-arg constructor)
  - b. Parameterized constructor

## Constructors in Java-How Constructors are Different From Methods in Java?

- ❖ Constructors must have the same name as the class within which it is defined while it is not necessary for the method in Java.
- ❖ Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
- ❖ Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

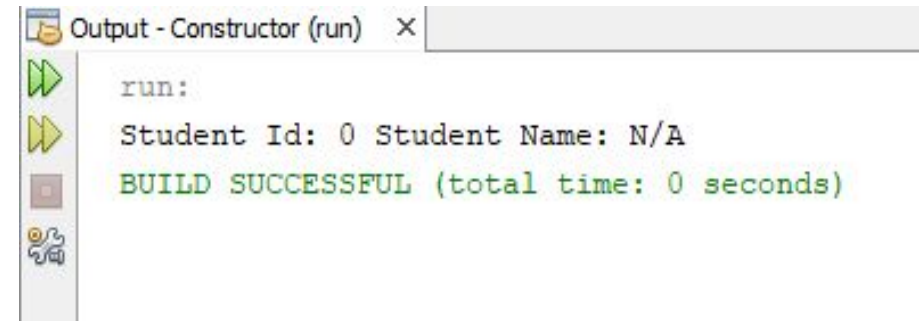
## Constructors In Java-Default Constructor

- ❖ A constructor is called "Default Constructor" when it doesn't have any parameter.
- ❖ The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.
- ❖ Default constructor is called on creating object

# Constructors In Java-Default Constructor

```
public class Constructor {  
  
    public static void main(String[] args) {  
        Student std1 = new Student();    //calls default constructor  
        System.out.println("Student Id: "+ std1.getId()+" Student Name: "+std1.getName());  
    }  
  
}
```

```
public class Student {  
    private int id;  
    private String name;  
    public Student() {    //default constructor  
        id=0;  
        name="N/A";  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```



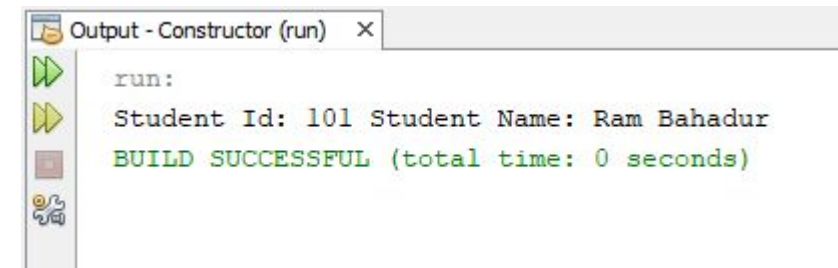
```
Output - Constructor (run) X  
run:  
Student Id: 0 Student Name: N/A  
BUILD SUCCESSFUL (total time: 0 seconds)
```

## **Constructors In Java-Parameterized Constructor**

- ❖ A constructor that has parameters is known as parameterized constructor.
- ❖ If we want to initialize fields of the class with our own values, then use a parameterized constructor.

# Constructors In Java-Parameterized Constructor

```
public class Constructor {  
  
    public static void main(String[] args) {  
        Student std1 = new Student(101, "Ram Bahadur");    //calls parameterized constructor  
        System.out.println("Student Id: " + std1.getId() + " Student Name: " + std1.getName());  
    }  
  
}  
  
public class Student {  
    private int id;  
    private String name;  
    public Student(int id, String name){    //parameterized constructor  
        this.id=id;  
        this.name=name;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
}
```



```
Output - Constructor (run) x  
run:  
Student Id: 101 Student Name: Ram Bahadur  
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Method Overloading in Java

- ❖ In Java, two or more methods may have same name but differ in parameters (different number of parameters, different types of parameters, or both)
- ❖ These methods are said to be overloaded methods and this feature is called method overloading.  
i.e. method overloading □ same method name but different parameters



# Method Overloading in Java-Example

```
package method.overloading;

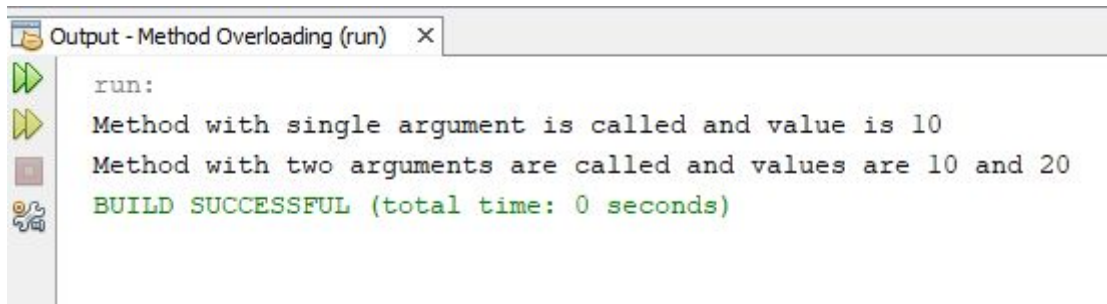
public class MethodOverloading {

    public static void main(String[] args) {
        MethodOverloading overLoading = new MethodOverloading();
        overLoading.myMethod(10);
        overLoading.myMethod(10, 20);
    }

    public void myMethod(int x) {
        System.out.println("Method with single argument is called and value is "+ x);
    }

    public void myMethod(int x, int y) {
        System.out.println("Method with two arguments are called and values are "+x+" and "+y );
    }

}
```



Output - Method Overloading (run) X

```
run:
Method with single argument is called and value is 10
Method with two arguments are called and values are 10 and 20
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Method Overloading in Java-Example

```
package method.overloading;

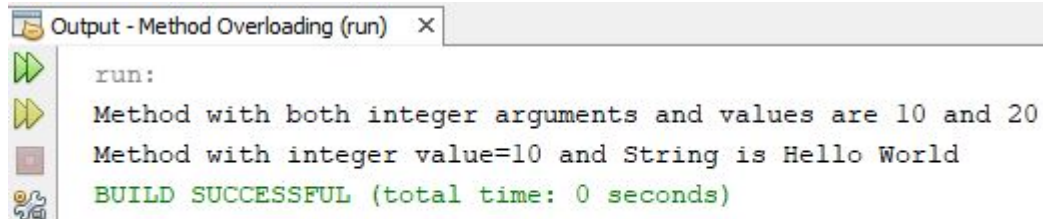
public class MethodOverloading {

    public static void main(String[] args) {
        MethodOverloading overLoading = new MethodOverloading();
        overLoading.myMethod(10,20);
        overLoading.myMethod(10,"Hello World");
    }

    public void myMethod(int x, int y){
        System.out.println("Method with both integer arguments and values are "+ x+" and "+y);
    }

    public void myMethod(int x, String y){
        System.out.println("Method with integer value="+x+" and String is "+y );
    }

}
```

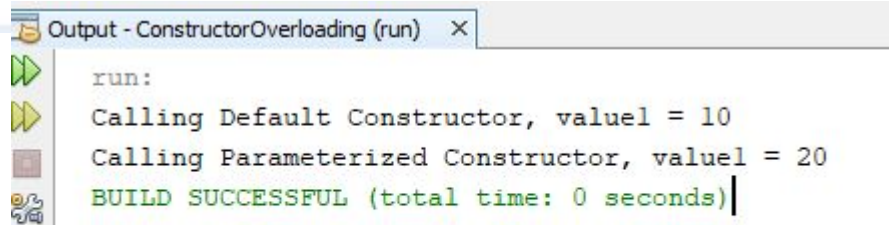


```
Output - Method Overloading (run) X
run:
Method with both integer arguments and values are 10 and 20
Method with integer value=10 and String is Hello World
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Constructor Overloading in Java

- ❖ Similar to methods overloading in Java, we can also create two or more constructors with different parameters. This is called constructor overloading.

```
public class ConstructorOverloading {  
    int value1;  
  
    public ConstructorOverloading() {  
        this.value1=10;  
    }  
    public ConstructorOverloading(int value1) {  
        this.value1=value1;  
    }  
    public static void main(String[] args) {  
        ConstructorOverloading co1 = new ConstructorOverloading();  
        ConstructorOverloading co2 = new ConstructorOverloading(20);  
        System.out.println("Calling Default Constructor, value1 = "+ co1.value1);  
        System.out.println("Calling Parameterized Constructor, value1 = "+ co2.value1);  
    }  
}
```

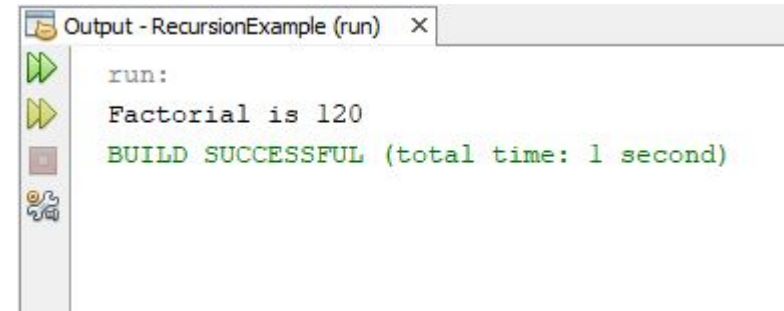


```
run:  
Calling Default Constructor, value1 = 10  
Calling Parameterized Constructor, value1 = 20  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Java Recursion

- ❖ A method that calls itself is known as recursive method and this process is called recursion.

```
public class RecursionExample {  
    static int fact(int n) {  
        if (n!=0) {  
            return n* fact(n-1);  
        }else{  
            return 1;  
        }  
    }  
  
    public static void main(String[] args) {  
        int n=5;  
        System.out.println("Factorial is "+fact(n));  
    }  
}
```



# Java Static Keyword

❖ In Java, if we want to access class members without creating an instance of class, we need to declare the class members static.

## ❖ Static Methods

- Static methods are also called as class methods because those methods belong to class rather than the object of a class.
- **We can invoke static methods directly using the class name.**

```
class Test {  
    // static method inside the Test class  
    public static void method() {...}  
}  
  
class Main {  
    // invoking the static method  
    Test.method();  
}
```



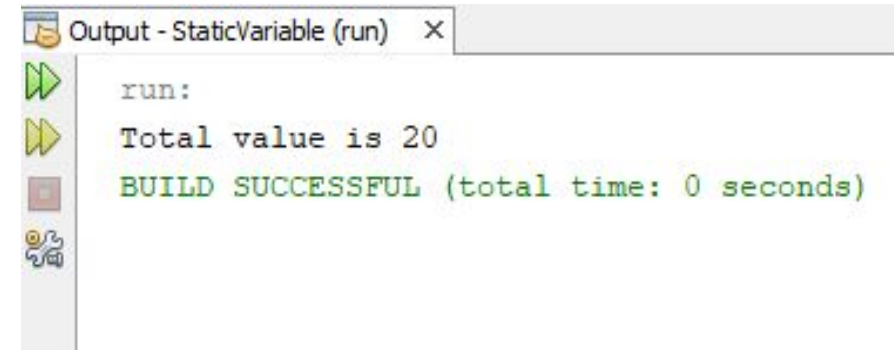
# Java Static Keyword

## ❖ Static Variables

- If we declare a variable static, all objects of the class share the same static variable.
- It is because like static methods, static variables are also associated with the class.
- We do not need to create objects of the class to access the static variables.

```
package staticvariable;
```

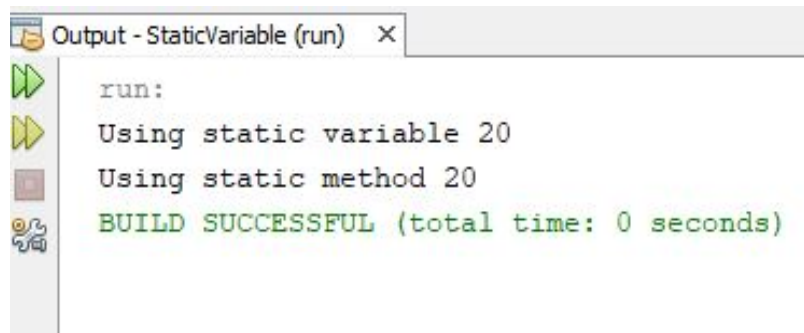
```
public class StaticVariable {  
  
    static int total=20;  
  
    public static void main(String[] args) {  
        System.out.println("Total value is "+ total);  
    }  
  
}
```



# Java Static Keyword-Accessing Static Variable and Methods

```
package staticvariable;

public class StaticVariable {
    static int total=20;
    static int getTotal(){
        return total;
    }
    public static void main(String[] args) {
        System.out.println("Using static variable "+ total);
        System.out.println("Using static method "+ getTotal());
    }
}
```



The screenshot shows the output of a Java program. The title bar of the window is "Output - StaticVariable (run)". The output text is as follows:

```
run:
Using static variable 20
Using static method 20
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Pass by Value and Pass by Reference in Java

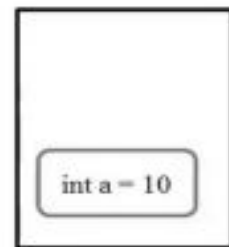
- ❖ Pass by Value and Pass by reference is the two ways by which we can pass a value to the variable in a function.
  - a. **Pass by Value:** It is a process in which the function parameter values are copied to another variable and instead this object copied is passed. This is known as call by Value.
  - b. **Pass by Reference:** It is a process in which the actual copy of reference is passed to the function. This is called by Reference.



# Java is Strictly Pass by Value

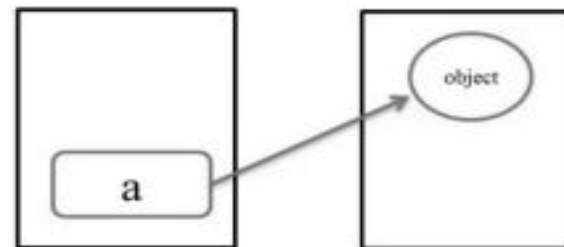
- ❖ Java is strictly pass by value. It means during method call, values are passed not addresses.
- ❖ Java follows the following rules in storing variables:
  - Local variables like primitives and object references are created on Stack memory.
  - Objects are created on Heap memory.

`int a = 10; // local variable`



Stack

`Test a = new Test( );`



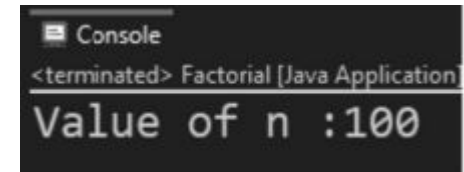
Stack

Heap

# Java is Strictly Pass by Value

❖ Pass by value is clear while using primitive types as shown below:

```
class Main {  
  
    static void change( int n ) {  
        n=50;  
    }  
  
    public static void main(String[] args) {  
        int n = 100;  
        change(n);  
        System.out.println("Value of n :"+n);  
    }  
}
```



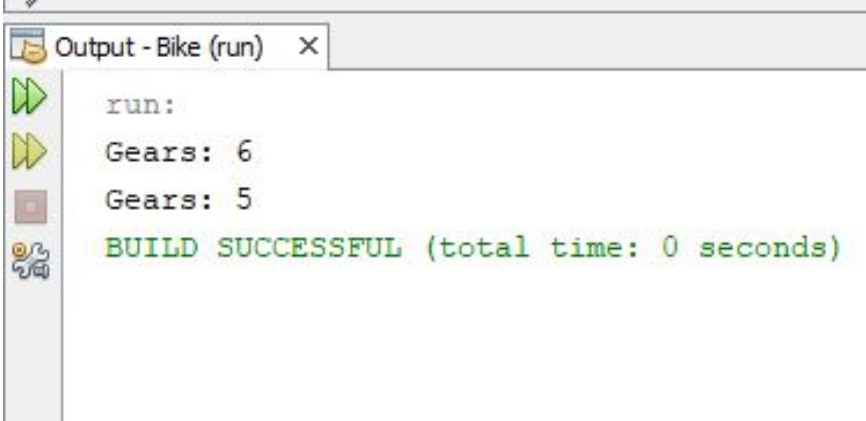
Console  
<terminated> Factorial [Java Application]  
Value of n :100

## **Java is Strictly Pass by Value**

- ❖ In case of objects, which are created in heap and have a reference in stack, they are also passed by value.
- ❖ We pass the reference to the object and the called method copies the reference to create another reference on stack which is pointing the same object.

# Java is Strictly Pass by Value

```
public class Bike {  
    private int gears;  
    public Bike(int gears) {  
        this.gears = gears;  
    }  
    public void setGears(int gears) {  
        this.gears = gears;  
    }  
    public int getGears() {  
        return gears;  
    }  
    public static void change(Bike bike) {  
        bike.setGears(5);  
    }  
    public static void main(String[] args) {  
        Bike myBike = new Bike(6);  
        System.out.println("Gears: " + myBike.getGears());  
        change(myBike);  
        System.out.println("Gears: " + myBike.getGears());  
    }  
}
```



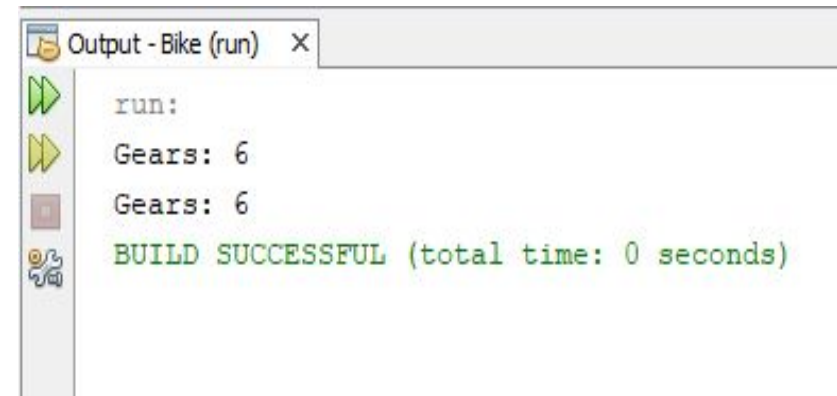
```
Output - Bike (run) X  
run:  
Gears: 6  
Gears: 5  
BUILD SUCCESSFUL (total time: 0 seconds)
```

- ❖ Here since both the copies are pointing the same object, changing the object in the called function changes the common object.

# Java is Strictly Pass by Value

- ❖ However if we use the new operator to change the reference in the called method then it points to a new object and changes to this new object is not reflected in the original object.

```
public class Bike {  
    private int gears;  
    public Bike(int gears) {  
        this.gears = gears;  
    }  
    public void setGears(int gears) {  
        this.gears = gears;  
    }  
    public int getGears() {  
        return gears;  
    }  
    public static void change(Bike bike) {  
        bike= new Bike(10);  
    }  
    public static void main(String[] args) {  
        Bike myBike = new Bike(6);  
        System.out.println("Gears: " + myBike.getGears());  
        change(myBike);  
        System.out.println("Gears: " + myBike.getGears());  
    }  
}
```



```
Output - Bike (run) X  
run:  
Gears: 6  
Gears: 6  
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Java Nested and Inner Class

- ❖ In Java, it is also possible to nest classes (a class within a class).
- ❖ The purpose of nested classes is to group classes that belong together, which makes our code more readable and maintainable.

```
class OuterClass {  
    // ...  
    class NestedClass {  
        // ...  
    }  
}
```

- ❖ There are two types of nested classes we can create in Java:
  - **Non-static nested class (Inner class)**
  - Static nested class:

## **Why Use Nested Classes?**

- ❖ It is a way of logically grouping classes that are only used in one place.
- ❖ It increases encapsulation
- ❖ It can lead to more readable and maintainable code.

## **Difference between nested class and inner class in Java**

- ❖ An inner class is a part of a nested class.
- ❖ Non-static nested classes are known as inner classes.



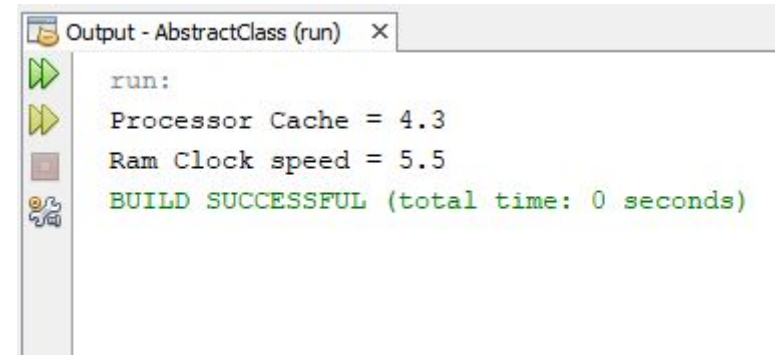
## **Non-Static Nested Class (Inner Class)**

- ❖ A non-static nested class is a class within another class. It has access to members of enclosing class(outer class).
- ❖ It is commonly known as inner class.
- ❖ Since the inner class exists within the outer class, we must instantiate the outer class first, in order to instantiate the inner class.
- ❖ We can access the members of the outer class from the inner class by using this keyword.

# Non-Static Nested Class (Inner Class)-Example

```
class CPU {  
    double price;  
    // nested class  
    class Processor{  
        // members of nested class  
        double cores;  
        String manufacturer;  
  
        double getCache(){  
            return 4.3;  
        }  
    }  
    // nested class  
    class RAM{  
        // members of nested class  
        double memory;  
        String manufacturer;  
        double getClockSpeed(){  
            return 5.5;  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
  
        // create object of Outer class CPU  
        CPU cpu = new CPU();  
  
        // create an object of inner class Processor using outer class  
        CPU.Processor processor = cpu.new Processor();  
  
        // create an object of inner class RAM using outer class CPU  
        CPU.RAM ram = cpu.new RAM();  
        System.out.println("Processor Cache = " + processor.getCache());  
        System.out.println("Ram Clock speed = " + ram.getClockSpeed());  
    }  
}
```



Output - AbstractClass (run) X

```
run:  
Processor Cache = 4.3  
Ram Clock speed = 5.5  
BUILD SUCCESSFUL (total time: 0 seconds)
```

❖ Note: We can declare the inner class as protected.

# Non-Static Nested Class (Inner Class)-Example-Accessing

```
class Car {
    String carName;
    String carType;
    // assign values using constructor
    public Car(String name, String type) {
        this.carName = name;
        this.carType = type;
    }
    // private method
    private String getCarName() {
        return this.carName;
    }
    class Engine {
        String engineType;
        void setEngine() {
            // Accessing the carType property of Car
            if(Car.this.carType.equals("4WD")){
                // Invoking method getCarName() of Car
                if(Car.this.getCarName().equals("Crysler")) {
                    this.engineType = "Smaller";
                } else {
                    this.engineType = "Bigger";
                }
            }else{
                this.engineType = "Bigger";
            }
        }
        String getEngineType(){
            return this.engineType;
        }
    }
}
```

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

        // create an object of the outer class Car
        Car car1 = new Car("Mazda", "8WD");

        // create an object of inner class using the outer class
        Car.Engine engine = car1.new Engine();
        engine.setEngine();
        System.out.println("Engine Type for 8WD= " + engine.getEngineType());

        Car car2 = new Car("Crysler", "4WD");
        Car.Engine c2engine = car2.new Engine();
        c2engine.setEngine();
        System.out.println("Engine Type for 4WD = " + c2engine.getEngineType());
    }
}
```

```
Console
<terminated> Factorial [Java Application] C:\Program Files\Java\jdk-16\bin
Engine Type for 8WD = Bigger
Engine Type for 4WD = Smaller
```

## Static Nested Class

- ❖ In Java, we can also define a static class inside another class.
- ❖ Such class is known as static nested class.
- ❖ Static nested classes are not called static inner classes.
- ❖ Unlike inner class, a static nested class cannot access the member variables of the outer class.
- ❖ It is because the **static nested class** doesn't require us to create an instance of the outer class.

# Static Nested Class

```
class MotherBoard {
    String model;
    public MotherBoard(String model) {
        this.model = model;
    }

    // static nested class
    static class USB{
        int usb2 = 2;
        int usb3 = 1;
        int getTotalPorts(){
            // accessing the variable model of the outer class
            if(MotherBoard.this.model.equals("MSI")) {
                return 4;
            }
            else {
                return usb2 + usb3;
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {

        // create an object of the static nested class
        MotherBoard.USB usb = new MotherBoard.USB();
        System.out.println("Total Ports = " + usb.getTotalPorts());
    }
}
```

error: non-static variable this cannot be referenced from a static context

- This is because we are not using the object of the outer class to create an object of the inner class.
- Hence, there is no reference to the outer class Motherboard stored in *Motherboard.this*

## Key Points to Remember on Nested an Inner Class

- ❖ Java treats the inner class as a regular member of a class. They are just like methods and variables declared inside a class.
- ❖ Since inner classes are members of the outer class, you can apply any access modifiers like private, protected to your inner class which is not possible on normal class.
- ❖ Since the nested class is a member of its enclosing outer class, you can use the dot (.) notation to access the nested class and its members.
- ❖ Using the nested class will make your code more readable and provide better encapsulation.
- ❖ Non-static nested classes (inner classes) have access to other members of the outer/enclosing class, even if they are declared private.