Object Oriented Programing 1

Lecture 10: Java Constructors, Java Modifiers & Java Encapsulation

By

Elubu Joseph

MSci.IS

Email: josebulinda@gmail.com

or

jose@kumiuniversity.ac.ug

Agenda

1. Java Constructors,
2. Java Modifiers &
3. Java Encapsulation

Constructors in Java

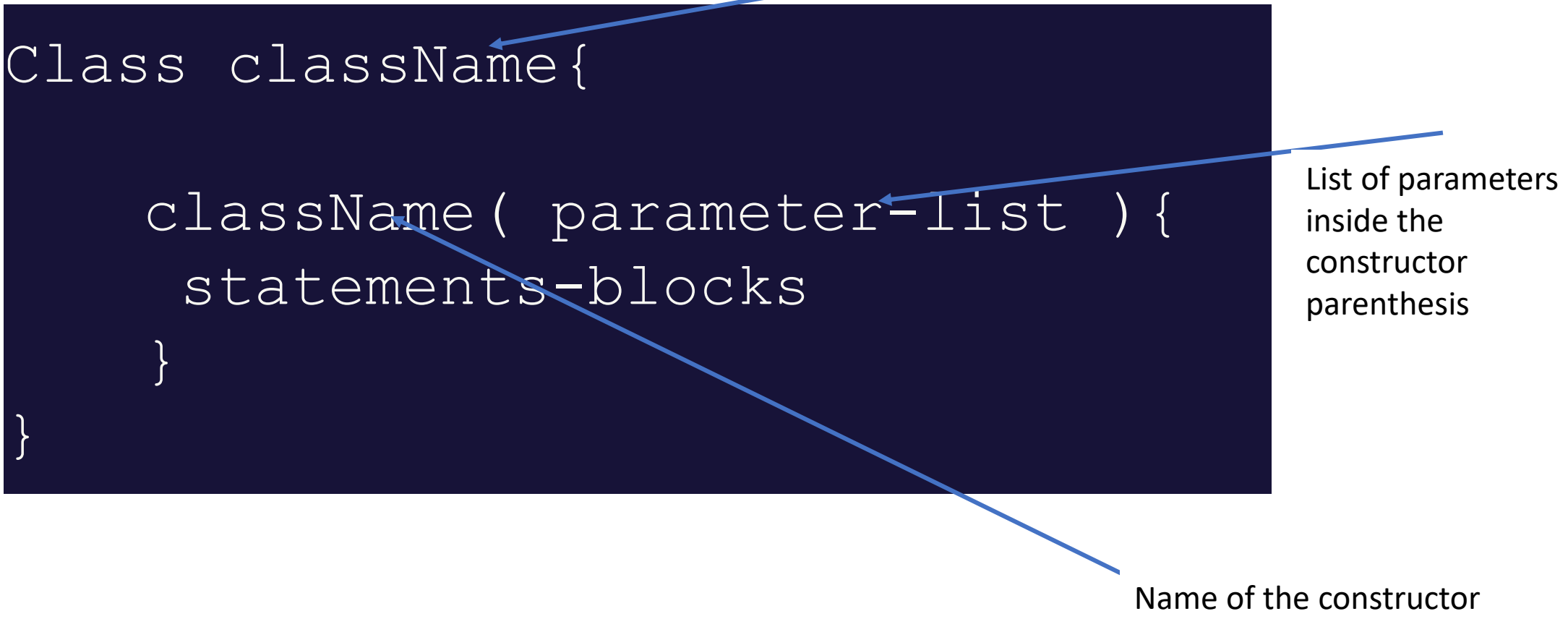
A constructor is a special method that is used to initialize an object. A constructor has **same name as the class name** in which it is declared. Every class has a constructor either implicitly or explicitly.

If we don't declare a constructor in the class then JVM builds a default constructor for that class.

Constructor must have no explicit return type. Constructor in Java **can not** be **abstract, static, final or synchronized**. These modifiers are not allowed for constructor.

Constructors in Java +

Syntax to declare constructor



Constructors in Java ++

Note that class name is the same as that of the method inside it. **className** is the name of class, as **constructor name** is same as class name.

parameter-list is the list of parameter which could be set within the constructor parenthesis. This is optional, because constructors can be parameterize and non-parameterize as well. This example show parmeterize constructor.

Example of a constructor

```
class MotorCar {
    String name;
    String model;    int year;

    MotorCar( ){
        name =“Pajero"; model="Great SUV";
        year =2021;
    }
}
```

Types of Constructor

Java Supports two types of constructors:

1. Default Constructor
2. Parameterized constructor

Each time a new object is created at least one constructor will be invoked.

```
MotorCar c = new MotorCar()//Default constructor invoked
MotorCar c = new MotorCar(name);//Parameterized
constructor invoked
```

Default Constructor

A default constructor is constructor which does not have any parameter. Default constructor can be either user defined or provided by JVM.

If a class does not contain any user defined constructor then during runtime JVM generates a default constructor also known as system defined default constructor.

If a class contain user defined constructor with no parameter then the JVM will not create default constructor.

Default Constructor+

Why create a constructor?

A constructor is created in order to initialize states of an object.

Example. See how JVM creates/adds a constructor at runtime.

```
public class MotorCar {
    String name;

    public static void main(String as[]){
    MotorCar ob=new MotorCar(){
                ob.name ="Pajero";
} }
```

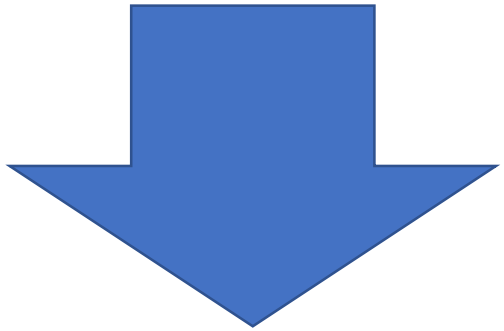
```
public class MotorCar {
    String name;
        MotorCar(){}
    public static void main(String as[]){
    Motor ob=new MotorCar();{
                ob.name ="Pajero";
} }
```

User Define Default Constructor

Constructor which is defined in the class by the programmer is known as user-defined default constructor.

Example:

In this example, we are creating a constructor note that it has same name as the class name.



User Define Default Constructor

```
class userConstruct{
     userConstruct(){
          int a=20; int b=30; int sum;
          sum=a+b;
     System.out.println("User Defined Constructor");
     System.out.println("Total of "+a+" + "+b+" = "+sum);
     }
    public static void main(String args[]){
      userConstruct ob=new userConstruct();
     }
}
```

OUTPUT

```
User Defined Constructor
Total of 20 + 30 = 50
```

Constructor Overloading

Like methods, a constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters.

Constructor overloading is not much different from method overloading.

Method overloading is where you have **multiple methods** with **same name** but **different signature**, whereas in Constructor overloading is where you have **multiple constructors** with **different signature** but only difference is that **constructor doesn't have return type**.

Example of constructor overloading

```
class Cricketer {
      String name; String team; int age;
      Cricketer () //default constructor. {
       name =""; team =""; age = 0;
      }
      Cricketer(String n, String t, int a) //constructor overloaded {
      name = n; team = t; age = a;
      }
      Cricketer (Cricketer ckt) //constructor similar to copy
constructor of c++ {
      name = ckt.name; team = ckt.team; age = ckt.age;
      }
      public String toString() {
      return "this is " + name + " of "+team+""+age;
      }

}
```

Example of constructor overloading

```
class Ctest{
      public static void main (String[] args) {
      Cricketer c1 = new Cricketer();
      Cricketer c2 = new Cricketer("lugogo", "Ug", 32);
      Cricketer c3 = new Cricketer(c2 );

      System.out.println(c2);
      System.out.println(c3);

      c1.name = "Vampas";
      c1.team= "Ug";
      c1.age = 32;

      System.out.println(c1);
}}
```

Output

```
this is lugogo of Ug 32
this is lugogo of Ug 32
this is Vampas of Ug 32
```

Constructor Chaining

Constructor chaining is a process of **calling one constructor from another constructor** within the same class.

Since constructor can only be called from another constructor, constructor chaining is used for this purpose.

To call constructor from another constructor **this keyword** is used. This keyword is used to refer to current object.

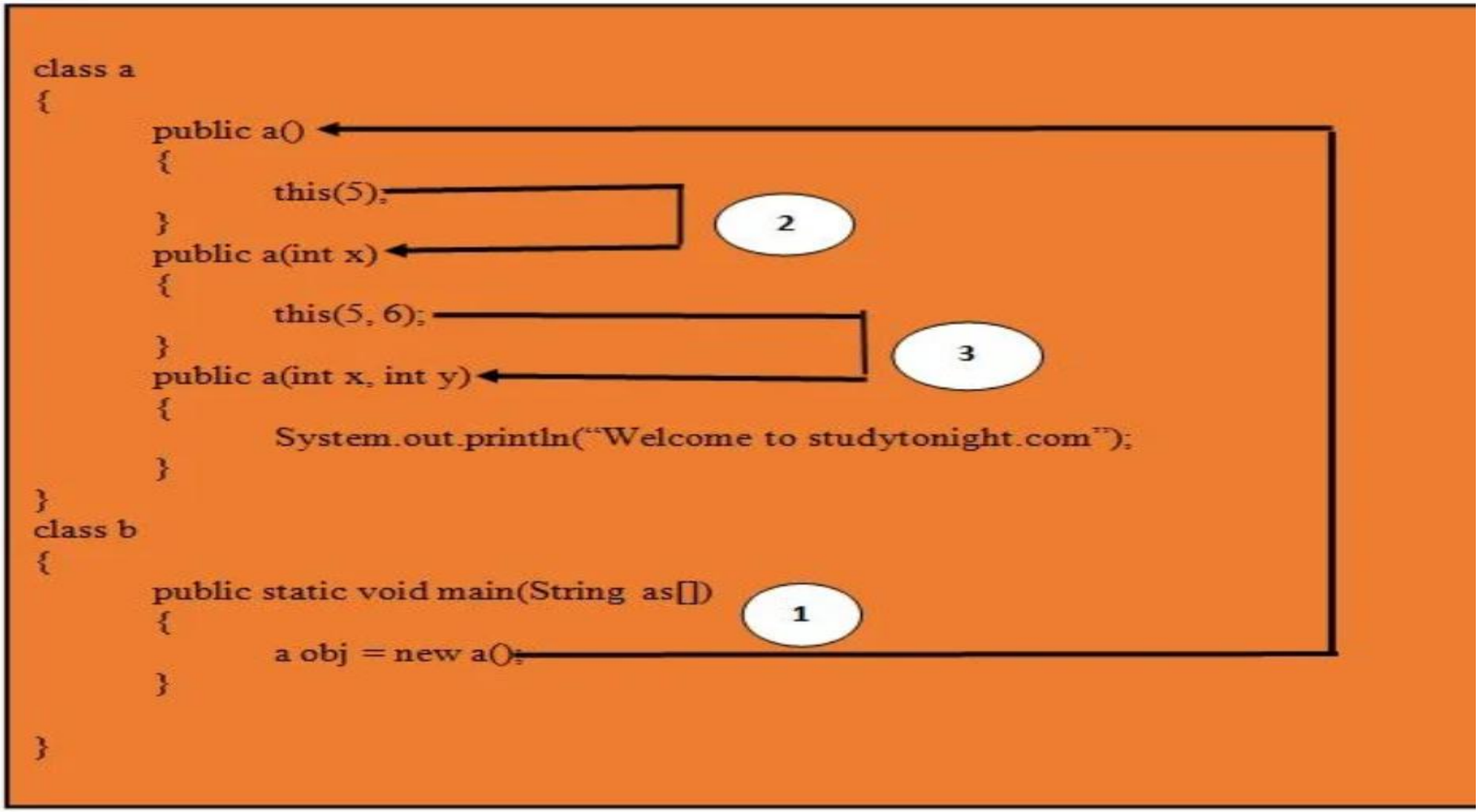
```
class Test {
    Test() {
        this(10);
    }
    Test(int x) {
    System.out.println("x="+x);
    }
    public static void main(String arg[]) {
    Test object = new Test();
}
}
```

Constructor Chaining+

Constructor chaining is used when we want to perform multiple tasks by creating a single object of the class.

In the image below, we have described the flow of constructor calling in the same class.

Constructor call in the same class



Constructor Chaining Example.

```
class abcPro{
      public abcPro() {
            this(15);
      System.out.println("Default Constructor");
      }
      public abcPro(int x) {
            this(8, 6);
      System.out.println("Constructor with one Parameter");
      System.out.println("Value of x ==> "+x);
      }
      public abcPro(int x, int y) {
      System.out.println("Constructor with two Parameters");
       System.out.println("Value of x and y ==> "+x+" "+y);
      }
}
```

Constructor Chaining Example

```
class ChainingPro{
    public static void main(String as[]) {
    abcPro obj = new abcPro();
    }
}
```

Output

```
Constructor with two Parameter
Value of x and y ==> 8 6
Constructor with one Parameter
Value of x ==> 15
Default Constructor
```

Private Constructors

In Java, we can create private constructor to **prevent class being instantiate**. It means by declaring a private constructor, it restricts to create object of that class.

Private constructors are used to create **singleton class**. A class which can have only single object is known as singleton class.

In private constructor, only one object can be created and the object is created within the class and also all the methods are static.

An object can not be created if a private constructor is present inside a class. A class which has a private constructor and all the methods are static is called **Utility class**.

Private constructor Example

```
final class abcPro1{
    private abcPro1() {}
    public static void add(int a, int b){
    int z = a+b;
    System.out.println("Addition: "+z);
    }
    public static void sub(int x, int y){
    int z = x-y;
    System.out.println("Subtraction: "+z);
    }
}
```

Private constructor Example

```
class PrivateConst{
    public static void main(String as[]){
    abcPro1.add(30, 5);
    abcPro1.sub(10, 3);
    }
}
```

Accessing the methods using fully qualified names not object of abcPro1 class.

Output

```
Addition: 35
Subtraction: 7
```

Java Access Modifiers

Java Access Modifiers

Access modifiers are keywords in Java that are used to set accessibility. An access modifier restricts the access of a class, constructor, data member and method in another class.

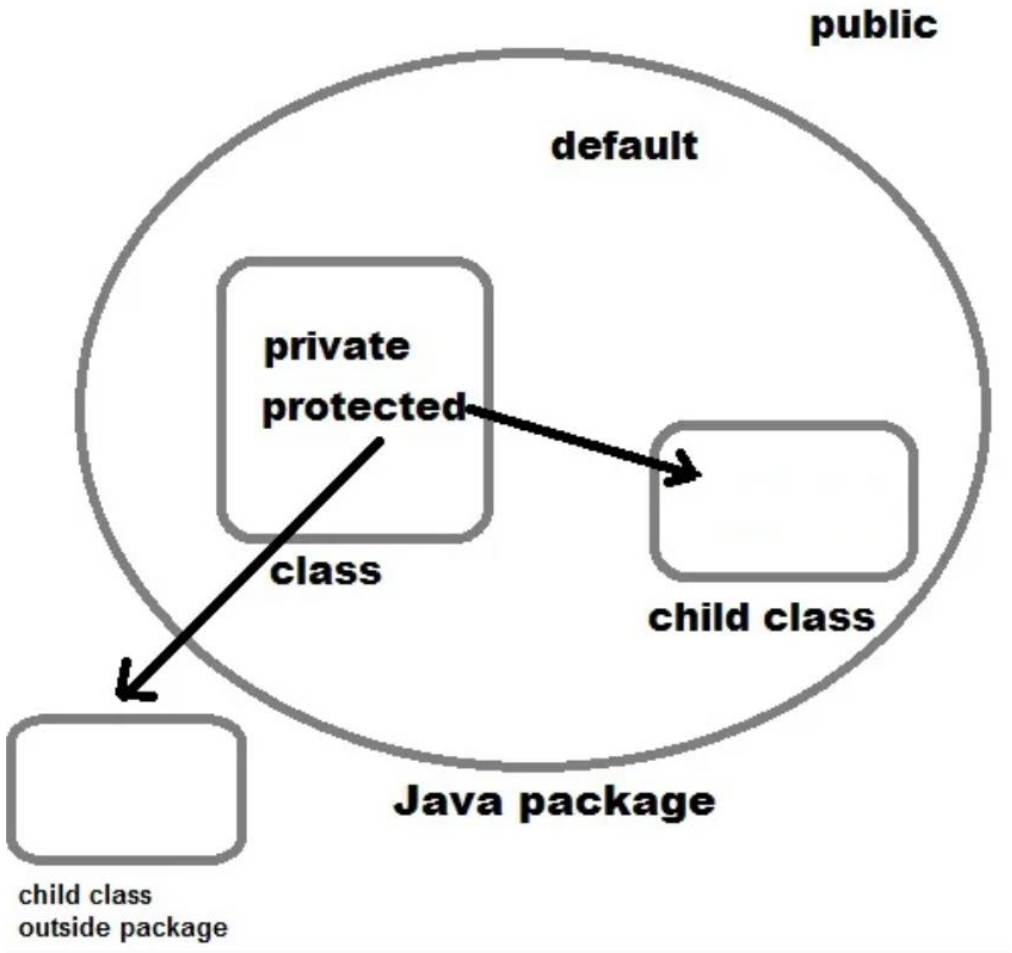
Java language has four access modifier to control access level for classes and its members

Java Access Modifiers+

1. **Default:** Default has scope only inside the same package
2. **Public:** Public has scope that is visible everywhere
3. **Protected:** Protected has scope within the package and all sub classes
4. **Private:** Private has scope only within the classes

Java also supports many non-access modifiers, such as **static, abstract, synchronized, native, volatile, transient** etc.

Java Access Modifiers+



Default Access Modifier

If we don't specify any access modifier then it is **treated as default** modifier. It is used to set accessibility within the package. It means we can not access its method or class from outside the package. It is also known as package accessibility modifier.

Example

In this example, we created a **DAM class** inside the **lecture10 package** and another class **DAMTest** by which we are accessing show() method of DAM class. We did not mentioned access modifier for the show() method that's why it is not accessible and reports an error during compile time.

Default Access Modifier sample program-DAM class

```
package lecture10;

public class DAM{
     int a = 10;
     void show(){//default access modifier for show()
          System.out.println(a);
     }
}
```

Default Access Modifier sample program-DAMTest class

```
import lecture10.DAM;

public class DAMTest{
	public static void main(String[] args){
	DAM ob = new DAM();
//compile error show() is not a public method
		ob.show();
	}
}
```

NOTE! This program will generate compilation error: show() is not a public method

Public Access Modifier

public access modifier is used to set public accessibility to a variable, method or a class. Any variable or method which is declared as public can be accessible from anywhere in the application.

Example

Here, we have two class DAM1 and DAMTest1 located in two different packages. Now we want to access show method of DAM1 class from DAMTest1 class. The method has public accessibility so it works fine. See the example below.

Public Access Modifier-DAM1 class

```
package Lecture10;
public class DAM1{
    int a = 60;
    // public access modifier
    public void show() {
        System.out.println(a);
    }
}
```

Public Access Modifier-DAMTest1 class

```
package Lecture10_2;
import lecture10.DAM1;
public class DAMTest1{
    public static void main(String[] args){
    DAM1 ob = new DAM1();
    ob.show();
    }
}
```

OUTPUT: 60

Protected Access Modifier

Protected modifier protects the variable, method from being accessible outside the package, but allow access to sub classes.

Example.

In this example, DAMTest2 class from lecture10_2 package extendeds DAM2 class from lecture10 package and called a protected method show() which should be accessible through **inheritance**.

Protected Access Modifier-DAM2 class

```
package lecture10;
public class DAM2{
    int a = 20;
    // protected access modifier
    protected void show() {
    System.out.println(a);
    }
}
```

Protected Access Modifier- DAMTest2 class

```
package lecture10_2;
import lecture10.DAM2;
public class DAMTest2 extends DAM2{
    public static void main(String[] args){
    DAMTest2 ob = new DAMTest2();
    ob.show();
    }
}
```

OUTPUT: 20

Note. this a child class.

Private Access Modifier

Private modifier is most restricted modifier which allows accessibility within same class only. We can set this modifier to any variable, method or even constructor as well.

Example

In this example, we set private modifier to show() method and try to access that method from outside the class. Java does not allow it to be accessed from outside the class.

Private Access Modifier-DAM3 class

```
class DAM3 {
    int a = 10;
    private void show() {
        System.out.println(a);
    }
}
```

Private Access Modifier-DAMTest3 class

```
public class DAMTest3 {

    public static void main(String[] args){
    DAM3 demo = new DAM3();

    demo.show(); // compile error
    }
}
```

Output

The method show() from the type DAM3 is not visible

Non-access Modifier

Non-access Modifier

Along with access modifiers, Java provides non-access modifiers as well. These modifier are **used to set special properties** to the variable or method.

Non-access modifiers do not change the accessibility of variable or method, but they provide special properties to them. Java provides following non-access modifiers.

1. Final
2. Static
3. Transient
4. Synchronized
5. Volatile

Final Modifier

Final modifier can be used with **variable, method** and class. if **variable** is declared **final** then we **cannot change** its value. If **method** is declared **final** then it **can not be overridden** and if a **class** is declared **final** then we **can not inherit it**.

Static modifier

static modifier is used to make field static. We can use it to declare variable, methods, class etc. static can be used to declare class level variable. If a method is declared static then we don't need to have object to access that. We can use static to create nested class.

Transient modifier

When an instance variable is declared as transient, then its **value doesn't persist** when an **object is serialized**

Synchronized modifier

When a method is synchronized it can be accessed by only one thread at a time. We will discuss it in detail in Thread.

Volatile modifier

Volatile modifier tells the compiler that the volatile variable can be changed unexpectedly by other parts of a program.

Volatile variables are used in case of multi-threading program.

Note! volatile keyword cannot be used with a method or a class. It can be only used with a variable.

Encapsulation in Java

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Advantage of Encapsulation in Java

1. By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.
2. It provides you the control over the data. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.
3. It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.
4. The encapsulated class is easy to test. So, it is better for unit testing.

Encapsulation sample program

Let's see the simple example of encapsulation that has only one field with its setter and getter methods. In this example we created two classes i.e Student.java and StudentTest calss.

```
//A Java class which is a fully encapsulated class.
//It has a private data member and getter and setter methods
package lecture10;
public class Student{
	//private data member
	private String name;
	//getter method for name
	public String getName(){
	return name;
	}
	//setter method for name
	public void setName(String name){
	this.name=name;
	}
}
```

Encapsulation sample program-

StudentTest calss

```
//A Java class to test the encapsulated class.
package lecture10;
class StudentTest{
      public static void main(String[] args){
      //creating instance of the encapsulated class
      Student s=new Student();
      //setting value in the name member
      s.setName("Tandema");
      //getting value of the name member
      System.out.println(s.getName());
      }
}
```

OUtput

```
run:
Tandema
```

Read-Only Class

```
//A Java class which has only getter methods.
package lecture10;
public class StudentRO{
      //private data member
      private String college="KUMU";
      //getter method for college
      public String getCollege(){
      return college;
      }
}
```

Now, you can't change the value of the college data member which is KUMU

Write-Only class

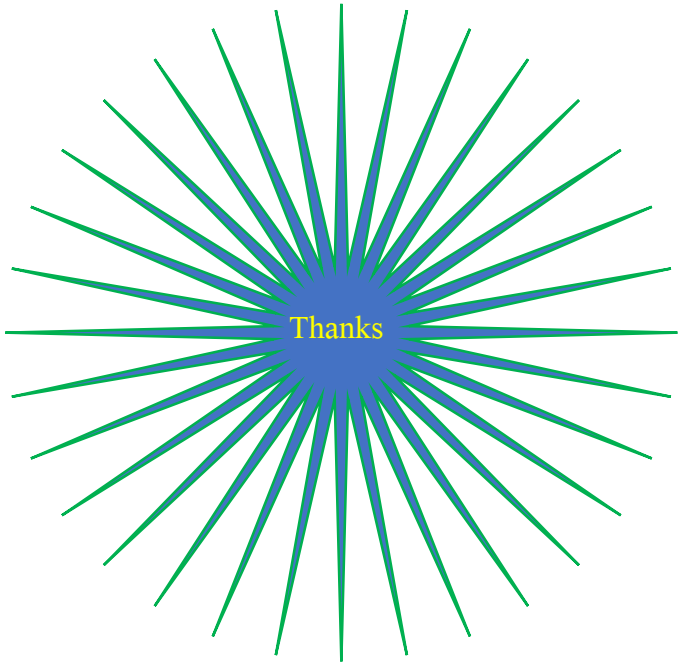
```
//A Java class which has only setter methods.
package lecture10;

public class StudentWO{
      //private data member
      private String college;
      //getter method for college
      public void setCollege(String college){
      this.college=college;
      }
}
```

Now, you can't get the value of the college, you can only change the value of college data member.

Summary

1. Java Constructors(Types and examples)
2. Java Modifiers –various types of access modifiers
3. Java Encapsulation – made the best use of setName() and getName(). And finally wrote Read-Only and Write-only programs.

Thank you for Listening



Reference

Access modifiers in Java. Studytonight.com. (n.d.). Retrieved June 2, 2022, from https://www.studytonight.com/java/modifier-in-java.php

Encapsulation in Java - Javatpoint. www.javatpoint.com. (n.d.). Retrieved June 2, 2022, from https://www.javatpoint.com/encapsulation

Constructors in Java. Studytonight.com. (n.d.). Retrieved June 2, 2022, from https://www.studytonight.com/java/constructor-in-java.php