Java OOP

Introduction to object oriented programming
Objects and classes
Constructors
this keyword
Access Modifiers
Method Overloading
String representation of an object

OOP Concepts : Knowledge Check

What are the different principle of OOPS?

Different principles of OOPS are Abstraction, Encapsulation, Polymorphism
and Inheritance.

1.Abstraction:– It is a thought process which show only the necessary properties.
2.Encapsulation:– Hiding complexity of a class.
3.Inheritance:- Defining a Parent-Child Relationship.
4.Polymorphism:– Object Changes it’s behaviour according to the situation.

Structure of a Java class

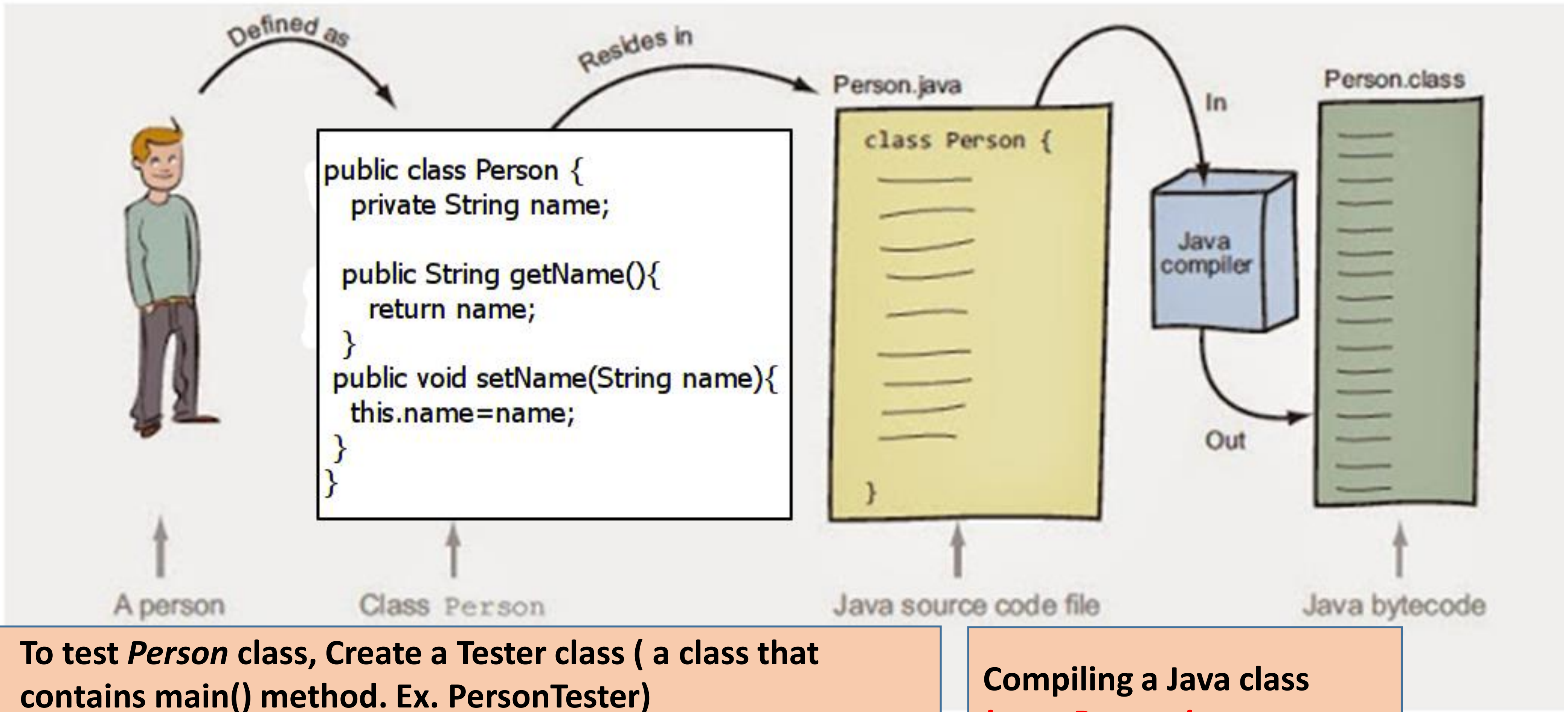
```
[package packagename; ]
import packagename.classname;
……
public class ClassName [extends SuperClassName [ implements InterfaceName,… ] ]
{
    instance variables (instance fields)
    class variables (static fields)

    constructor methods
    getter and setter methods
    business-logic methods


}
```

Can be public or package-private(default)

Note: Methods are of two types:

- Instance methods
- Static methods

Java Class Example



```
public class Person {
   private String name;

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

- To test *Person* class, Create a Tester class (a class that contains main() method. Ex. PersonTester)
- Instantiate Person class within the main method of Tester class

Instantiating Person class
```
Person person = new Person();
```

- Compile *PersonTester* class

  Javac PersonTester.java
- Run Java Tester class : java PersonTest

Compiling a Java class
javac Person.java

CONSTITUENTS OF A CLASS

Data members/fields

- instance fields/variables
- static fields (also called as class variables)

Methods

- instance methods
- class/static methods

Note : The main method may or may not be present depending on whether the class is a starter/tester class or not.

Instance variables

- Instance variables are declared in a class, but outside a method, constructor or any block.

- Instance variables are part of an object that occupies memory in the heap when it is created with the use of the keyword 'new' and destroyed by garbage collector when the object has no reference.

- Access modifiers can be applied for instance variables.

- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level)

- Instance variables have default values. For numbers the default value is 0, for booleans it is *false* and for object references it is null.

Accessing Instance Variables and Methods:

- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and in a different class (when instance variables are given accessibility) should be called using the fully qualified name ***objectName.variableName*****.**

- ***ObjectName* is also called as *ObjectReference.***

- Instance variables and methods are accessed via created objects.

- To access an instance member, the fully qualified path should be as follows:

/ First create an object */*
ClassName ***objectReference*** = new ClassName();

/ Now call a variable as follows */*
objectReference.variableName;

Ex.

```
Circle circle = new Circle();

double area;

a = circle.areaOfCircle();
```

this keyword

```
class Counter{
        private int count; // instance variable


        public void increment(){
         count= count + 1;
        }

        public int getCount(){
         return count;
        }
}
```

```
class CounterTest{
      public static void main(String args[]){
            Counter a = new Counter();
            Counter b = new Counter();
            a.increment();
            b.increment();
            System.out.println(a.getCount());
            System.out.println(b.getCount());
      }
}
```

- In the Counter example, when we invoke a.increment(), it contains count=count+1 statement.
- Whose count is it?
- When we say *a.increment()* , how does the method know that it has to increment the instance field of object a ?

this keyword

The answer is:

- ***The invoking object is implicitly received within the instance method via this*** reference.
- ***this*** is a built-in reference to the invoking object within the instance method

```
public void increment(){
    this.count = this.count + 1;
}
```

```
public int getCount(){
    return this.count;
}
```

Java OOP

- **Constructor methods**
  - **default constructor**
- **static/class Variables & Methods**
- **Static block**

CONSTRUCTOR METHOD

- A constructor method is an instance method that is invoked during the creation of the object.

- Memory allocation of objects will be in the heap area

- Constructor Methods are primarily used to initialize the data members of an object.

- When writing a constructor, remember that:
    - **it has the same name as the class name**
    - **it does not return a value not even *void.***
    - **It may or may not have parameters (arguments) i.e. constructor methods can be overloaded.**

- **By default, a class is provided with a default constructor(no-arg constructor).**

Default Constructor Method

- The default constructor provided by java has empty body but the we can explicitly define the default constructor method and initialize data members.

- *Syntax of default constructor:*

```
[access_modifier] classname(){
        //code
    }
```

```
public class Student{
    private int admissionCode;
    private String studentName;
    private int marks;

    //default constructor method
    public Student(){
            this.admissionCode=1;
            this. studentName="Unknown";
             this.marks=0;
       }
 // getter & setter methods
}
```

static variable

static variable or class variable

- It is a variable which belongs to the class
- A single copy is shared by all instances of the class
- static variables are placed in method area
- static variables can be accessed without creating of instance of the class to which it belongs to

Syntax:

```
ClassName.staticVariable
```

static method

static method or class method

- Accessed using ***ClassName.methodName***
- For using static methods, creation of instance is not necessary
- A static method can directly access the static members of the class. Instance members of the class cannot ***be directly*** accessed since static method do not have access to this reference.
- To access instance members of the class within the static method , we can achieve indirectly i.e. by creating instance/object of the class and invoking instance members through the object.

Note: static methods can be accessed with an object reference:

objectName.staticMethodName(args) but this is discouraged because it does not make it clear that they are class methods.

static data member : DEMO

```
Class Duck {
    private int size;
    private static int duckCount;
    public Duck(){
                duckCount++;
    }
    public void setSize (int size){
                this.size = size;
    }
    public int getSize (){
                return this.size;
    }
    public static void main(String args[]){
                System.out.println("Size of the duck is;" + size);
                Duck duck1 = new Duck();
                System.out.println(duck1.getSize()); // duck1.size
                }}
```

The static duckCount variable is initialized to 0, ONLY when the class is first loaded, NOT each time a new instance is created.

Each time the constructor is invoked ie an object gets created, the static variable duckCount will be incremented thus keeping a count of the total no of Duck objects created

Which duck? Whose size? A static method cannot directly access non-static.

Re-writing the class, Duck

```
public class Duck {
private int size;
private static int duckCount;

public Duck(){
        duckCount++;
}

public int getSize() {
        return size;
}

public void setSize(int size) {
        this.size = size;
}

public static int getDuckCount() {
        return duckCount;
}
}
```

```
public class DuckDemo {
public static void main(String[] args) {
        System.out.println("Duck size : "+ size);
        Duck duck1 = new Duck();
        System.out.println("Duck size : "+ duck1.getSize());
        System.out.println("Total ducks : "+ Duck.getDuckCount());
        Duck duck2 = new Duck();
        System.out.println("Duck size : "+ duck2.getSize());
        System.out.println("Total ducks : "+ Duck.getDuckCount());
}
}
```

Java Project : DuckProject

Packages:

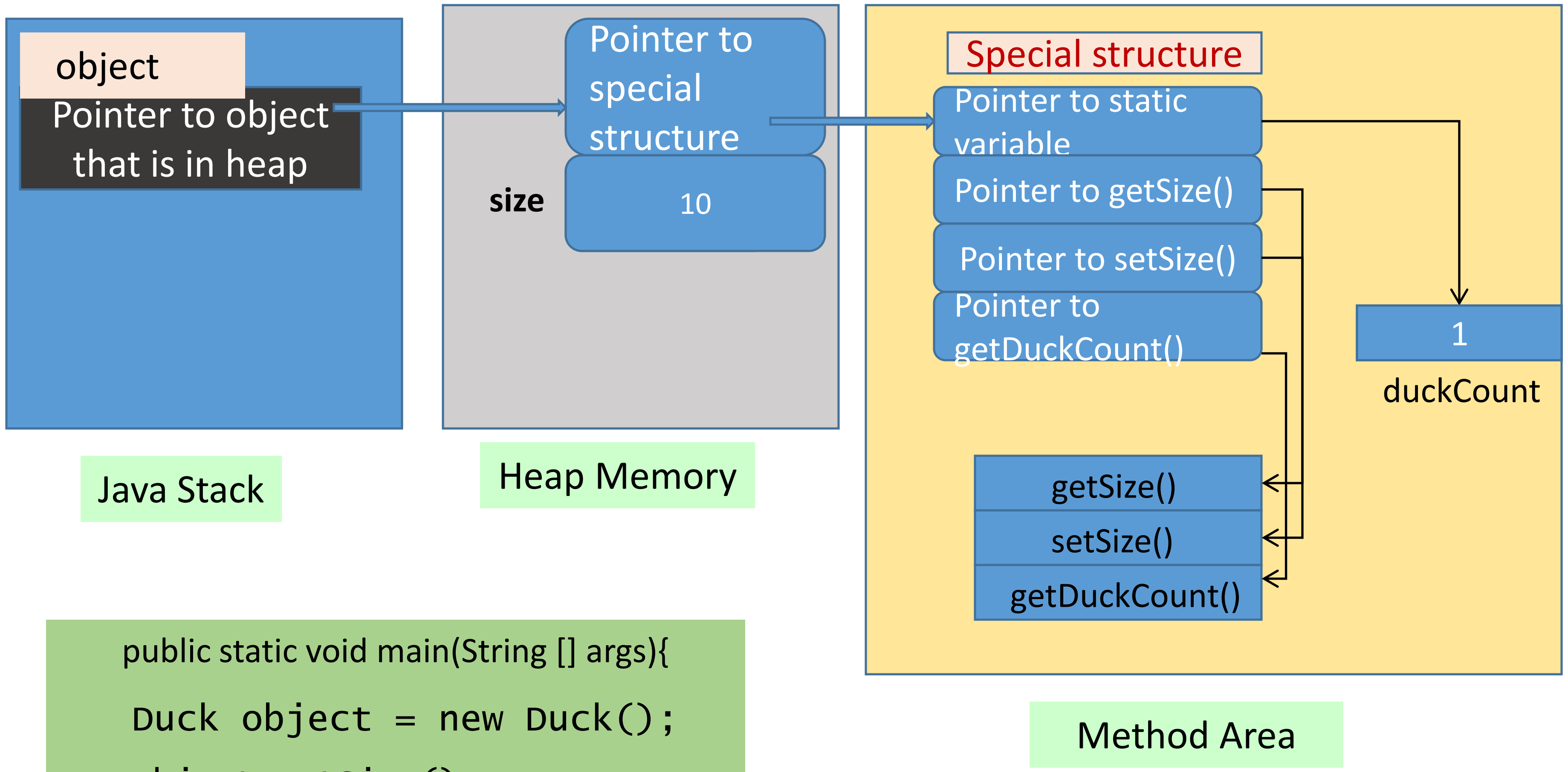
        com.deloitte.trg.service

                Class Name : Duck

        com.deloitte.trg.ui

                Class Name: DuckDemo

Memory Organization for members of a class



```
public static void main(String [] args){
    Duck object = new Duck();
    object.getSize();
    object.setSize(10);
    Duck.getDuckCount();
    object.getSize();
}
```

static block

static block

- **The static block is a block of statements inside a Java class that will be executed when a class is first loaded and initialized**
- A class is loaded typically after the JVM starts
- A static block helps to initialize the static variables just like constructors help to initialize instance variables.

```
class Test{
        static {
                //Code goes here
        }
}
```

```
public class StaticBlockDemo {
        private static int a;
        private static int b
        static{
                a=100;b=200;
                 System.out.println("I'm static block");
          }
        public static void main(String [] args){
                 System.out.println(a+", "+b);
        }
}
```

Java OOP

Access modifiers

Controlling Access to Members of a Class

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
- There are two levels of access control:
  - At the top level—public, or *package-private* (no explicit modifier).
  - At the member level—public, private, protected, or *package-private* (no explicit modifier).
- A class may be declared with the modifier public, in which case that class is visible to all classes everywhere.
- If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes)

Controlling Access to Members of a Class

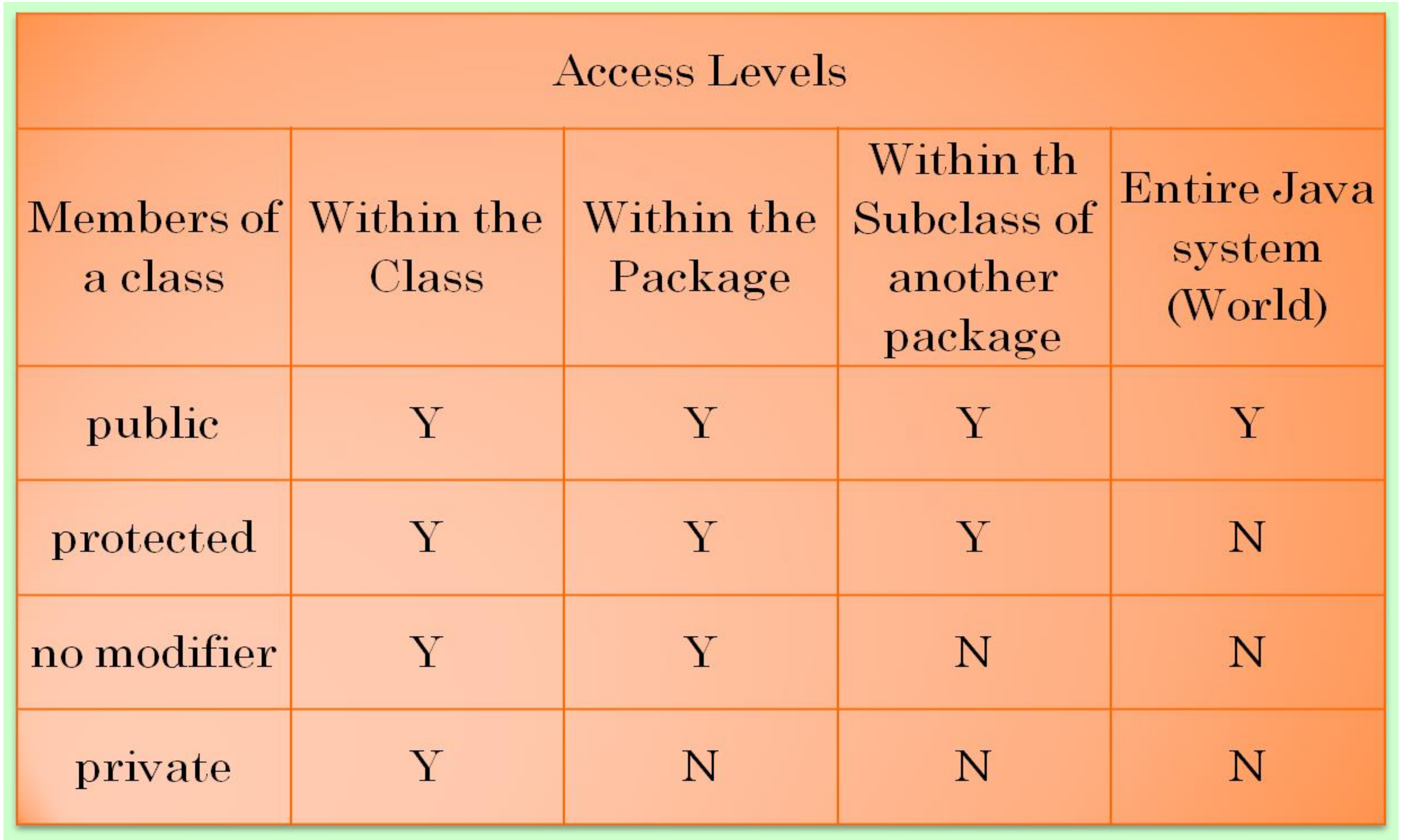
- At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning.
- For members, there are two additional access modifiers: *private* and *protected*.
- The private modifier specifies that the member can only be accessed in its own class.
- The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

Controlling Access to Members of a Class

Keyword	Applicable To	Who can Access
private	Data members and methods Inner classes	All members within the same Class only
(No keyword, usually we call it default or package-private)	Data members, methods, classes and interfaces	All classes in the same package
protected	Data members and methods	All classes in the same package as well as all its sub classes , even if sub classes reside in a different package
public	Data members, methods, classes and interfaces	Any class

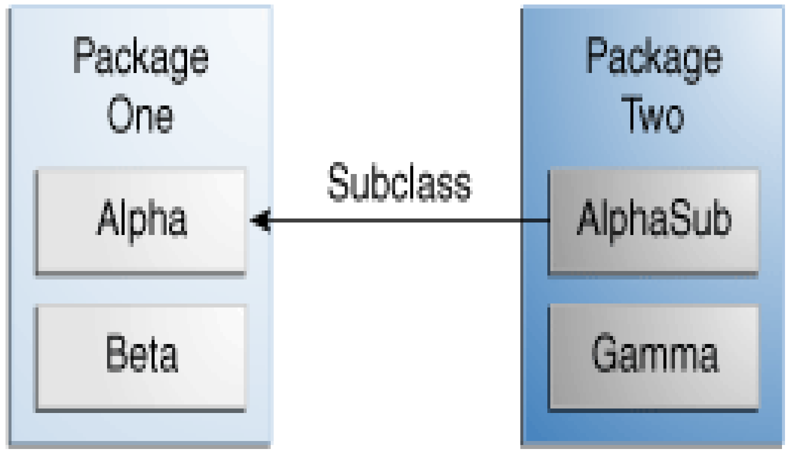
Controlling Access to Members of a Class

The following table shows the access to members permitted by each modifier.



Access Levels				
Members of a class	Within the Class	Within the Package	Within th Subclass of another package	Entire Java system (World)
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Controlling Access to Members of a Class



Visibility				
Modifiers for the members of Alpha class	**Alpha**	**Beta**	**AlphaSub**	**Gamma**
public				
protected				
no modifier				
private				

GENERALIZATION

In the spirit of encapsulation ,

- Data members are always kept ***private***
  - It is accessible only within the class
- The methods which expose the behavior of the object are kept ***public***
  - However, we can have helper methods which are private

Key features of object oriented programs

- encapsulation (binding of code and data together)
- State (data) is hidden and Behavior (methods) is exposed to external world

Java - OOP

Variables Scope & Lifetime
Object State
Object Reference
Call by value & Call by reference

Variables scope & Lifetime

Java comes with three kinds of scope and we name variables according to the scope they reside in as detailed in the table below.

Variable	Scope	Lifetime
static	Static variables apply to the class as a whole and are declared within the class but outside a method.	Exists for as long as the class it belongs to is loaded in the JVM.
instance	Instance variables apply to an instance of the class and are declared within the class but outside a method.	Exists for as long as the instance of the class it belongs to.
local	Local variables apply to the method they appear in.	Exists until the method it is declared in finishes executing.

Object State

Each time we create a new instance of the class, that instance gets its own set of instance variables. The values held within each instance are known as *object state*.

Objects live on ***t****he Heap* and so does the instance variables which represents an *Object State.*

Object Reference

What is Object Reference variable in java?
Simply, it is a variable whose type is an object type; i.e. some type that is either java.lang.Object or a subtype of java.lang.Object.

Object Reference

- **You can not do pointer arithmetic with references.**
- **References are strongly typed.** Another difference is that the type of a reference is *much* more strictly controlled in Java than the type of a pointer is in C.

    In C you can have an int* and cast it to a char* and just re-interpret the memory at that location.

That re-interpretation doesn't work in Java: you can only interpret the object at the other end of the reference as something that it *already is* (i.e. you can cast a Object reference to String reference *only if* the object pointed to is actually a String).

```
Object object1="Hello";
String string1=(String) object1;
System.out.println(string1);

Object object2=100;
String string2=(String) object2;
System.out.println(string1);
```

```
Runtime exception
java.lang.ClassCastException:
java.lang.Integer cannot be cast to
java.lang.String
```

```
Integer intObject = (Integer) object2;//Valid
System.out.println(intObject);
```

Call by value & Call by reference

1. Some Definitions
   - Call by value: pass contents of the actual parameter.
   - Call by reference: pass address of the actual parameter.

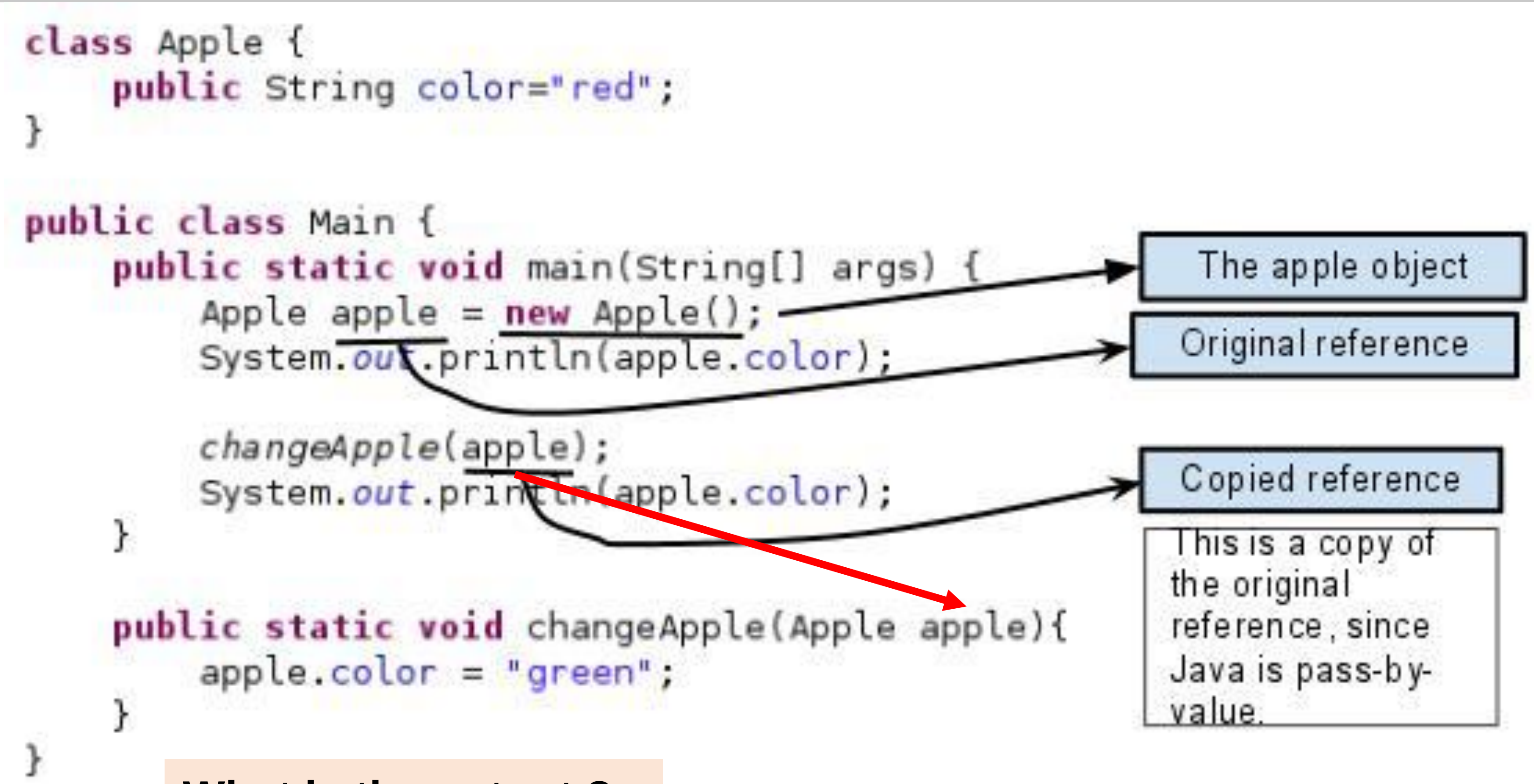
Java is always call-by-value. Primitive data types and object reference are just values.

2. Passing Primitive Type Variable

Since Java is pass-by-value, it's not hard to understand the following code will not swap anything.

```
public void swap(Type arg1, Type arg2) {
      Type temp = arg1;
      arg1 = arg2;
      arg2 = temp;
}
```

Passing object reference

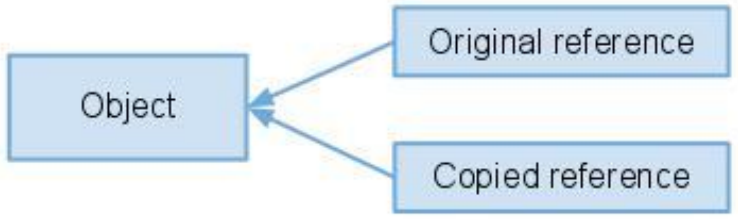


```
class Apple {
    public String color="red";
}

public class Main {
    public static void main(String[] args) {
        Apple apple = new Apple();
        System.out.println(apple.color);

        changeApple(apple);
        System.out.println(apple.color);
    }

    public static void changeApple(Apple apple){
        apple.color = "green";
    }
}
```

The apple object
Original reference
Copied reference
This is a copy of the original reference, since Java is pass-by-value.

What is the output ?

Since the original and copied reference refer the same object, the member value gets changed.



Output :
red
green

Java-OOP

Constructor overloading
this() method

CONSTRUCTOR OVERLOADING

- Defining more than one constructor method within a class is called as ***constructor overloading.***

- The constructor methods defined in a class should differ in number of arguments or their types.

- Since the constructor methods receive parameters, they are also called as ***parameterized constructor methods.***

- Recall that default constructor does not receive any parameters and also remember that constructor methods are instance methods that get implicitly executed during the creation of objects.

CONSTRUCTOR OVERLOADING Example

```
public class Circle {
private int x,y;
private double radius;
public Circle() {
      x = 0; y = 0; radius = 0.0;
}
public Circle(int x, int y, double radius) {
   this.x = x; this.y = y;
   this.radius = radius;
  }
public Circle(double radius) {
      this.x = 0;
      this.y = 0;
      this.radius = radius;
}
```

```
public Circle(Circle circle) {
      this.x = circle.x;
      this.y = circle.y;
      this.radius = circle.radius;
 }
```

```
Circle obj1= new Circle();
Circle obj2= new Circle(10,20,5.4);
Circle obj3= new Circle(5.4);
Circle obj4 = new Circle(obj2);
```

this() method

Purpose:

To invoke one constructor method within another constructor method of a *same class.*

Usage:

To avoid duplication of code within parameterized constructors of a class.

Note: this() method can be called only within the constructor method and it has to be the first statement of the constructor method.

Constructor Overloading Example Re-written

```
public class Circle {
private int x,y;
private double radius;
public Circle(int x, int y, double radius) {
        this.x=x;
        this.y=y;
        this.radius = radius;
  }
public Circle() {
      this(0,0,0);
}
public Circle(double radius) {
      this(0,0,radius);
      }
public Circle(Circle circle) {
      this(circle.x, circle.y, circle.radius);
 }

 }
```

```
Circle obj1= new Circle();
Circle obj2= new Circle(10,20,5.4);
Circle obj3= new Circle(5.4);
Circle obj4 = new Circle(obj2);
```

Java - OOP

Method Overloading

final variable

METHOD OVERLOADING

- Defining more than one method within a class having ***same name but differing in method signature*** are termed overloaded methods
- The method signature refers only to parameter-list
- Calls to overloaded methods will be resolved during ***compile time***
    - Also known as ***static polymorphism or static binding***

- Argument(Parameter) list could differ via:
    - Number of parameters
    - Datatype of parameters
    - Sequence of parameters

- If two or more method have same name and same parameter list **but differs in return type are not** said to be overloaded method
- Access modifiers are also **not** part of method signature

METHOD OVERLOADING

Ex.

```
void add (int a, int b)

int add (int a, float b)

void add (int a, float b)

void add (int a, int b, float c)
```

Overloaded methods

~~int add (int a, float b)~~

~~void add (int a, float b)~~

Not overloading.
Compiler error.

METHOD OVERLOADING Example

```
public class MethodOverload {
void test(int a) {
System.out.println("a: "+a);
}

void test(int a,int b) {
System.out.println("a and b: "+ a+","+ b);
}
void test(float a) {
System.out.println("double a: "+a);
}
}
```

```
public class MethodOverloadTest {

public static void main(String[] args) {
MethodOverload overload=new
MethodOverload();
overload.test(10);
overload.test(10,2);
//overload.test(5.5);
overload.test((float) 5.5);
}
}
```

Output:

```
a:10
a and b:10,20
double a:5.5
```

METHOD OVERLOADING : implicit data conversion example

```
class Overload {
void test(int a, int b) {
    System.out.println("a and b: " + a + "," + b);
}
void test(double a) {
    System.out.println("double a: " + a);
}
}
```

```
class MethodOverloading {
public static void main(String args[]) {
Overload overload = new Overload();

overload.test(10);
overload.test(10, 20);
overload.test(5.5);
}
}
```

Output:

```
double a:10.0

a and b:10,20

double a:5.5
```

Implicit data conversion takes place.

int is converted to double

METHOD OVERLOADING

```
class Overload {
void test(int a) {
System.out.println("a: " + a);
}
void test(int a, int b) {
System.out.println("a and b: " + a + "," + b);
}

}
```

```
class MethodOverloading {
public static void main(String args[]) {
Overload overload = new Overload();

overload.test(10);
overload.test(10, 20);
overload.test(5.5);

}
}
```

COMPILATION ERROR

Solution:
overload.test((int)5.5);

final variables

final is a keyword in java and can be applied to member variables, methods, classes and local variables in Java.

What is final variable in Java?

- Any variable either member variable or local variable (declared inside method or block) modified by final keyword is called final variable.
- final variables are often declared with static keyword in java and treated as constants.

Note: Final variables are equivalent to constants in other programming languages. So they have to be initialized during declaration & cannot be modified later.

final variables

```
public class FinalVariableExample {
public static void main(String[] args) {
	//final int HOURS_IN_DAY=24;
	final int HOURS_IN_DAY;
	//Valid: One time assignment is allowed
	HOURS_IN_DAY=24;
	//Compile Error: Cannot assign again
	//HOURS_IN_DAY=12;
	System.out.println("Hours in 5 days = "+ HOURS_IN_DAY*5);
}
}
```

Compilation error
public static final int HOURS_IN_DAY;
Conversion specifiers and static keyword cannot be applied for local variables.

Note: The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_").

Java - OOP

String representation of an object

String representation of an object

- The toString() method of an Object class returns a string representation of the object.
- In general, the toString method returns a string that "textually represents" this object.
- We can override the *toString()* method of Object class to provide string representation of the object.
- Except for ***wrapper class objects and String objects***, when an object is placed in println() method, following is displayed:

  packagename.classname@ unsigned hexadecimal representation of the hash code of the object.

- In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

String Representation of an object

```
public class Demo {
private String message;

public Demo(){
this.message="Hi! Welcome";
}

public String getMessage() {
return message;
}
public void setMessage(String message) {
this.message = message;
}
```

```
public class DemoTester {
public static void main(String[] args) {
        Demo demo = new Demo();
        System.out.println(demo);
}
}
```

```
@Override
public String toString() {
  return "Demo [message=" + message + "]";
}
```

If a method marked with @Override fails to correctly override a method in one of its super classes, the compiler generates an error.

```
Demo [message=Hi! Welcome]
```

Problem-statement : Student class

Java Project: StudentAdmissionProject

Create a class, Student in the service layer with the following details :

private Integer admissionCode;
private String studentName;
private Date birthdate;
private Integer [] marks; *// array to hold marks in 3 subjects*
private String grade;
private static Integer admissionCounter;
private static String schoolName;

- create a static block to store “Hyderabad Public School” in static field *schoolName* .
- default constructor – a*dmissionCode* to be auto generated with the help of *admissionCounter* variable.
- 2-arg constructor that takes s*tudentName and birthdate* as arguments- a*dmissionCode* to be auto generated using *admissionCounter* variable.
- 3-arg constructor that takes s*tudentName, birthdate* and *marks* as arguments- a*dmissionCode* to be auto generated using *admissionCounter* variable.

Note: Invoke 2-arg constructor within 3-arg constructor using this() method

- Getter for all the data members.
- Setter methods for studentName, birthdate and marks
- Setter method for grade has to assign grade based on total marks(subject1+subject2+subject3) as follows:
  - **Excellent** : if marks between 240 and 300, **Good** **:** if marks between 239 and 180,
  - **Average** : if marks between 150 and 179 and **Fail** : if marks are less than 150.
- Getter and setter methods for *schoolName*
- Override *toString()* method to provide String representation of the Student class

Problem-statement : StudentValidator class

The following class to be created in service layer

•Class : **StudentValidator**

•An instance method, *public boolean isValidStudentName(Student student)* returns *true* if the student name contains only alphabets and spaces else returns *false.*

•An instance method, *public boolean isValidStudentBirthdate(Student student)* returns *true* if student's birthdate least greater than current date by 5 years else returns *false*

•An instance method *public boolean isValidStudentMarks(int [] marks)* that returns *true* if marks in all the 3 subject are between 0 to 100, else returns *false.*

•An instance method, *public boolean isValidStudent(Student student)* returns *true* if the Student object is valid based invoking the above 3 methods int this method else returns false.

Problem-statement : StudentManager class

The following class to be created in service layer

Class : **StudentService**

- A instance method,
- A instance method*, public void showStudentDetails(Student [] student)*
    - *Displays students details along with their school name.*

Problem-statement : StudentTester class

- Create Tester class, **StudentTester in ui layer.**
- **C**reate at least 5 Student objects, place them in an array .

  Note: Before placing the Student object in the array, validate the Student object by invoking isValidStudent() of **StudentValidator** class.
- Display all the students details, along with their school name by invoking *showStudentDetails()* method of **StudentManager** class.

Thank You!