

Programming I

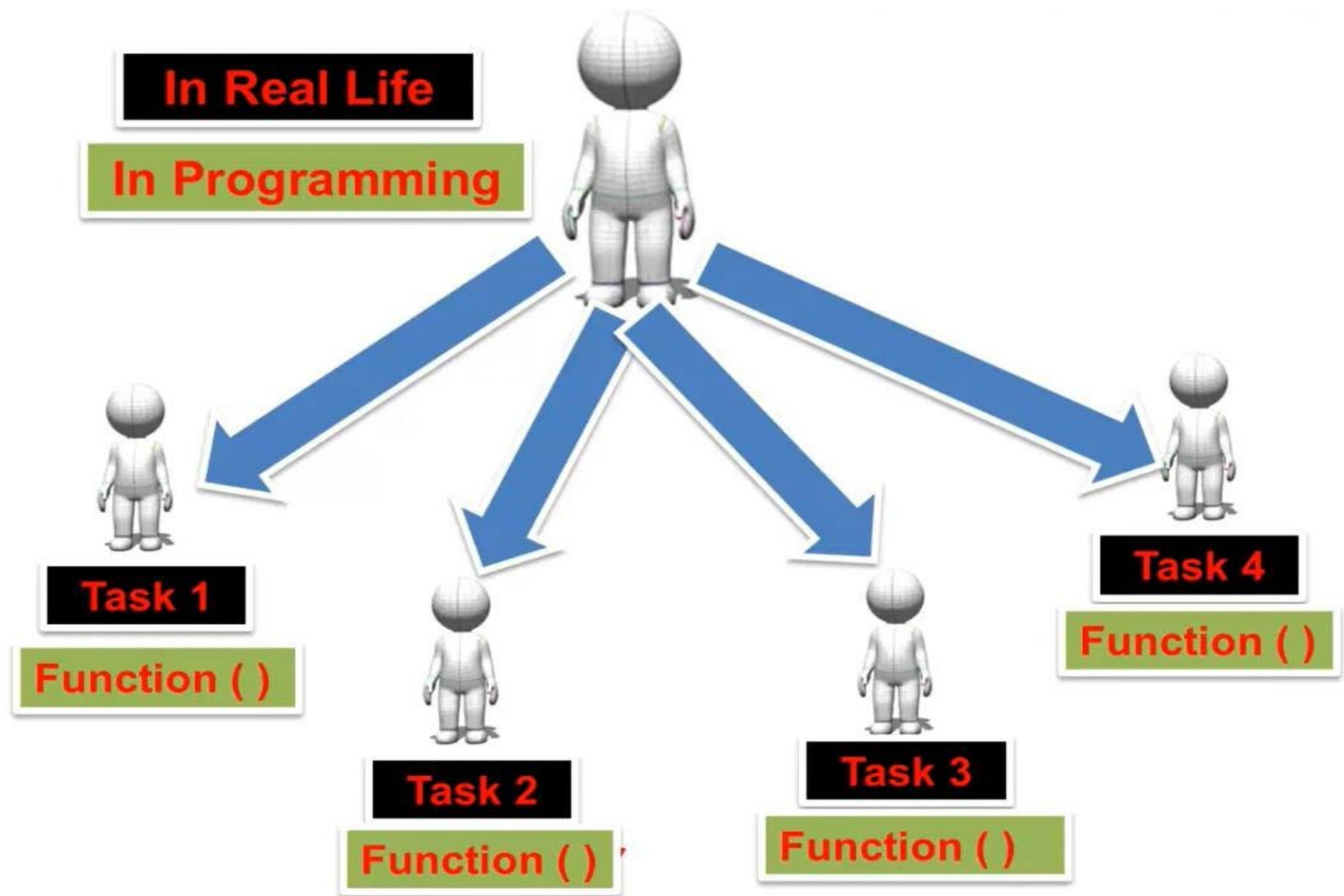
Chapter 4: **Methods (Functions)**

Dr/ Abeer Amer

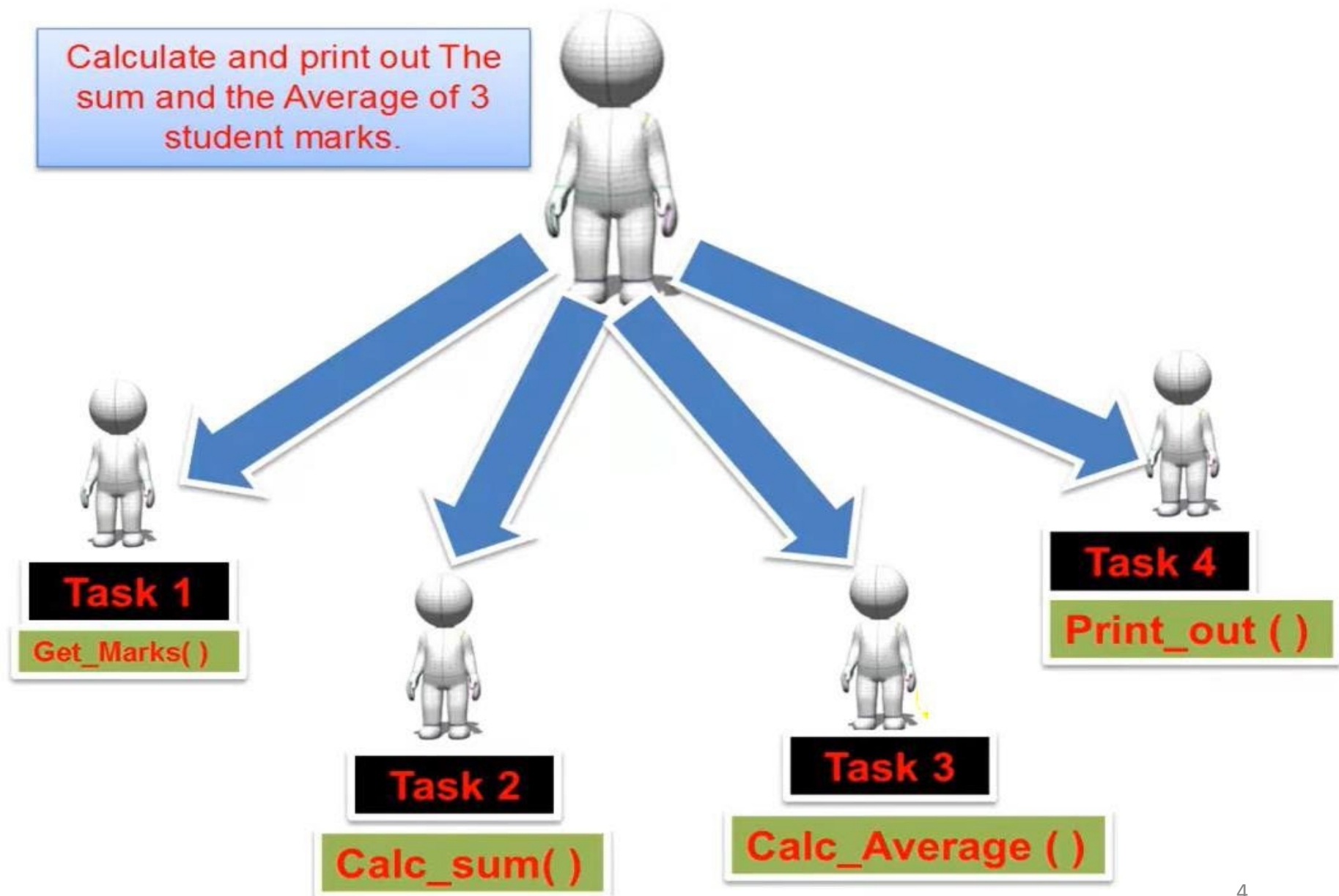
Method (Function):

- Most computer programs that solve real-world problems include hundreds even thousands of lines.
- Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or modules, each of which is more manageable than the original program.
- Method (Function) is a self-contained block of statements that perform a specific task.
- Using a function is something like hiring a person to do a specific job for you.

Method:



Method:



Function Types:

1- Built in Functions:

- **Example: Math functions.**

```
double x = 25;  
System.out.println (Math.sqrt(x));
```

2- User Defined Functions.

2- User Defined Functions:

Syntax:

```
return_value_type function_name ( parameter list)
{
    .....
    body of the function
    .....
    return (...);
}
```

Parameter List:

- It is a list of data values supplied from the calling program as input to the function.
- It is optional.

return-value-type:

- It is the data type for the returned value from the function to the calling program.
- It is Mandatory, if no returned value expected, we use keyword void.

```
void function_name ( parameter list)  
    {  
        .....  
        body of the function  
        .....  
    }
```

- In void functions do not use a **return** statement to return a value.

2- Calling a Function (From the main ())

Syntax:

Function name (Actual Parameter-list)

```
public static void main (string [ ] args )
```

```
{
```

```
... ..  
... ..
```

```
functionName();
```

```
... ..  
... ..
```

```
}
```

```
void functionName()
```

```
{
```

```
... ..  
... ..
```

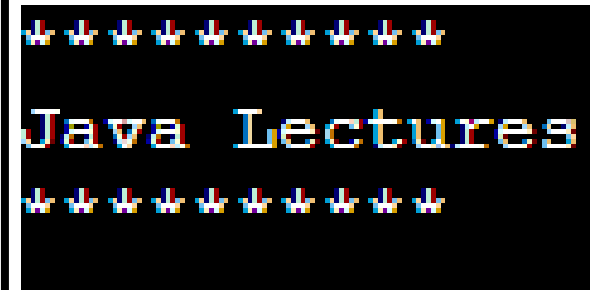
```
}
```

Example

Write a program using functions to write a statement between two-star lines.

```
public class Main
{
    public static void main(String[] args) {
        drawline();
        System.out.println("Java Lectures");
        drawline();
    }
    public static void drawline()
    {
        for (int i=1; i<=10; i++)
        { System.out.print("*");
        }
        System.out.println();
    }
}
```

Output:



```
*****
Java Lectures
*****
```

Example:

Write a program using methods to write a statement between star lines and # line.

```
public class Main
{
    public static void main(String[] args) {
        drawline('*');
        System.out.println("\nJava Lectures");
        drawline('#');
    }
    public static void drawline(char x)
    {
        for (int i=1; i<=10; i++)
        { System.out.print(x);
        }
    }
}
```

Output:

```
*****
Java Lectures
#####
```

Example:

Write a program using methods to write a statement between two lines have varied length using one function.

```
public class Main
{
    public static void main(String[] args) {
        drawline('*', 10);
        System.out.println("\nJava Lectures");
        drawline('#', 20);
    }
    public static void drawline(char x, int length)
    {
        for (int i=1; i<=length; i++)
            System.out.print(x);
    }
}
```

Output:

```
*****
Java Lectures
#####
```

Example:

Write a program using functions to find the sum of two integer numbers.

```
public class Main
{
    public static void main(String[] args) {
        int x = 20;
        int y = 30;
        System.out.println("sum is"+sum(x,y));
    }
    public static int sum (int i, int j)
    { int result = i + j ;
      return result;
    }
}
```

Output:

sum is 50

Example:

Write a program using functions to find the sum of two integer numbers.

```
public class Main
{
    public static void main(String[] args) {
        int x = 20;
        int y = 30;
        int total = sum (x,y);
        System.out.println(" sum is "+ total);
    }
    public static int sum (int i, int j)
    { int result = i + j ;
      return result;
    }
}
```

Output:

sum is 50

Example:

**Write a program using three functions for:
Sum 3 numbers - Find the average - print the results.**

```
import java.util.Scanner;
public class Main
{
    public static void main(String[] args) {
        Scanner input=new Scanner (System.in);
        System.out.println("Enter the three numbers:");
        int number1 =input.nextInt();
        int number2 =input.nextInt();
        int number3 =input.nextInt();
        int total = sum (number1,number2,number3);
        double avg= average (number1,number2,number3);
        display(total,avg);
    }
    public static int sum (int n1, int n2, int n3)
    { return n1+n2+n3;}
    public static double average (int i, int j, int k)
    { return sum (i,j,k)/3f;}
    public static void display (int t , double a)
    { System.out.println(" The sum is "+ t);
      System.out.println(" The average is "+ a);}
}
```

Output:

```
20
30
The sum is 60
The average is 20.0
```

Scope of Variables:

- Scope is the context within a program in which a variable is valid and can be used.
- Variables: Local and Global.

Local variable:

- Declared within a function (or block)
- Can be accessible only within the function of from declaration to the end of the block.

Global variable:

- Declared outside of every function definition.
- Example: `static int x;` (before main function)

Local Variable:

```
public class Main
{
    public static void main(String[] args) {
        int x =100;
    }
    public static void display()
    {
        System.out.println(x);
    }
}
```

```
Main.java:16: error: cannot find symbol
        System.out.println(x);
                           ^
    symbol:   variable x
    location: class Main
1 error
```

```
public class Main
{
    public static void main(String[] args) {
        int x =100;
        {int y =200;
        }
        System.out.println(y);
    }
}
```

```
Main.java:15: error: cannot find symbol
        System.out.println(y);
                           ^
    symbol:   variable y
    location: class Main
1 error
```

Local Variable:

```
public class Main
{
    public static void main(String[] args) {
        int x =100;
        {
            System.out.println ("x = "+x);
        }

    }
}
```

x = 100

Global Variable:

```
public class Main
{ static int x;
  public static void main(String[] args) {
    x = 100;
    System.out.println ("x = "+x);
    display();
  }
  public static void display()
  { System.out.println ("x = "+x);
  }
}
```

```
x = 100
x = 100
```

```
public class Main
{ static int x;
  public static void main(String[] args) {
    x = 100;
    System.out.println ("x = "+x);
    int x = 200;
    System.out.println ("x = "+x);
    display();
  }
  public static void display()
  { System.out.println ("x = "+x);
  }
}
```

```
x = 100
x = 200
x = 100
```

Method Overloading:

- Defining several methods within a class with the same name. As long as every method has a different signature.

Method Signature:

The signature of a method consists of the following:

- Method name.
- Formal parameter list
- The returned type value is not part of its signature.

Examples:

- `public int Sum(int x, int y)`
- `public int Sum(int x, int y , int z)`
- `public double Sum(double x, double y, double z)`
- `public int Sum(int x, double y)`

Example:

```
public class Main
{ static int x;
    public static void main(String[] args) {
        int x = 20, y = 30, z = 70;
        System.out.println(getlarger(x,y));
        System.out.println(getlarger(x,y,z));
    }
    public static int getlarger (int a , int b)
    { if (a > b)
        return a;
      else return b;
    }
    public static int getlarger (int a , int b , int c)
    { if (a > b && a > c)
        return a;
      else if (b > a && b > c)
        return b;
      else return c;
    }
}
```

30

70

Programming II

Object Oriented Programming (OOP) **Introduction to OOP**

Dr/ Abeer Amer

What you know?

- How to write a structured java code
- How to compile and run a java code
- How to use character in java
- Mathematical functions
- Conditional statements
- Repetition statements : Loops
- Methods
- Arrays (1D and 2D)

Course Content

- Recursion
- Object and Classes
- Encapsulation
- Constructors
- Inheritance
- Polymorphism + Abstract classes
- Interfaces and Super Classes
- Exception Handling
- Array List Class

Structure Programming:

- Structure programming is a program written with only the structured programming constructions:
 - (1) Sequence
 - (2) Repetition
 - (3) Selection.

Object Oriented Programming (OOP):

- Object oriented programming is a type of programming based on the concept of "objects".
- Objects can contain data and code.
- data are represented in the form of fields which are known as *attributes* or *properties of the object*
- And code, in the form of procedures which are often known as methods.

Object-oriented programming languages:

- The most popular OOP languages are:

❖ [Java](#)

❖ [JavaScript](#)

❖ [Python](#)

❖ [C++](#)

❖ [Visual Basic .NET](#)

❖ [PHP](#)

Principles of OOP:

- Encapsulation.
- Abstraction.
- Inheritance.
- Polymorphism.

What is OOP?

- Object-oriented programming (OOP) is a programming paradigm based on the concept of “objects” .
- A programming paradigm is a style of programming, a way of thinking about software construction.
- A programming paradigm does not refer to a specific language but rather to a way to build a program or a methodology to apply.
- Some languages make it easy to write in some paradigms but not others.
- Some programming languages allow the programmer to apply more than one paradigm.

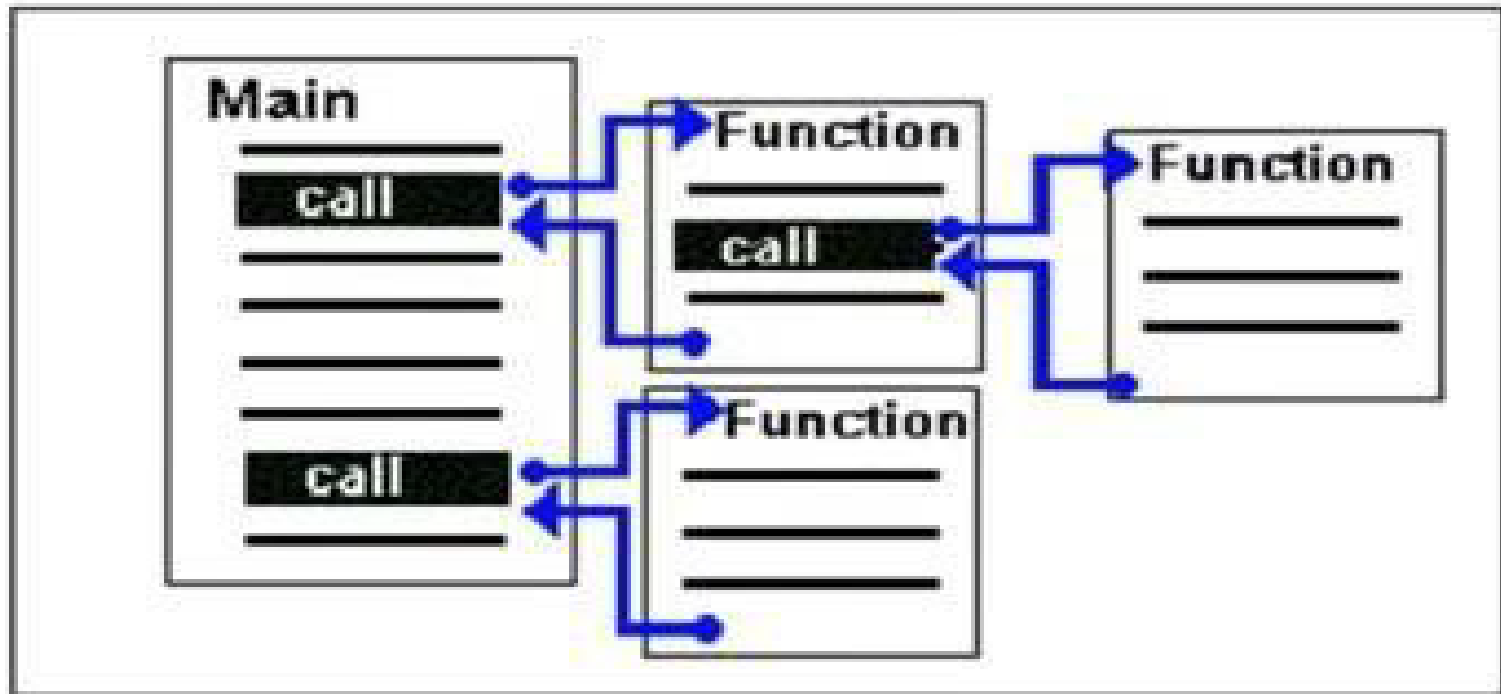
Programming Paradigms:

- **Procedural:** Basic, Pascal.
- **Object-oriented:** C++, Java
- **Declarative:** Prolog.
- **Event-driven:** Visual Basic, C#

Examples on Programming Paradigms:

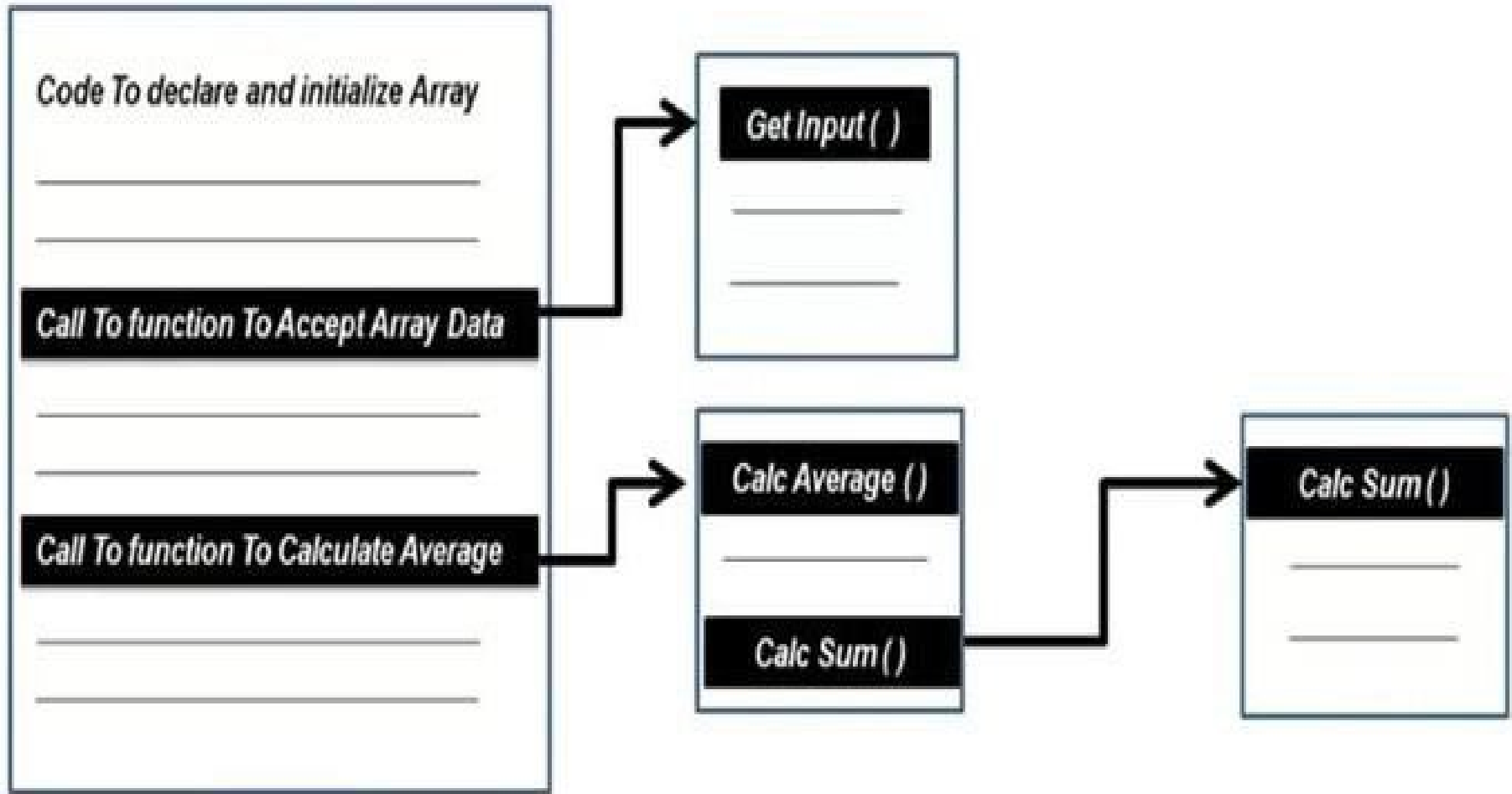
Procedural Programming (PP):

- It is about writing a list of instructions to tell the computer what to do step by step.
- It relies on procedures or routines.



Procedural Programming Example:

- Program to calculate average of array items.



Object Oriented Programming (OOP):

- Object-oriented programming (OOP) is a programming paradigm based on the concept of “**objects**”.

Example on Objects:

Objects in student housing management program:

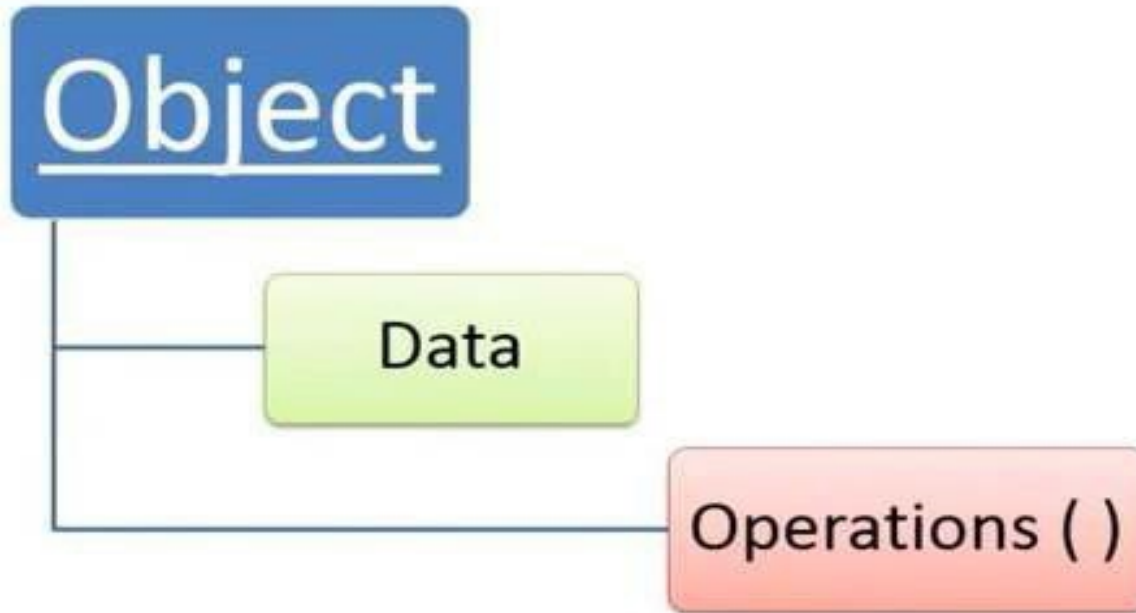


Objects in College Management Program:

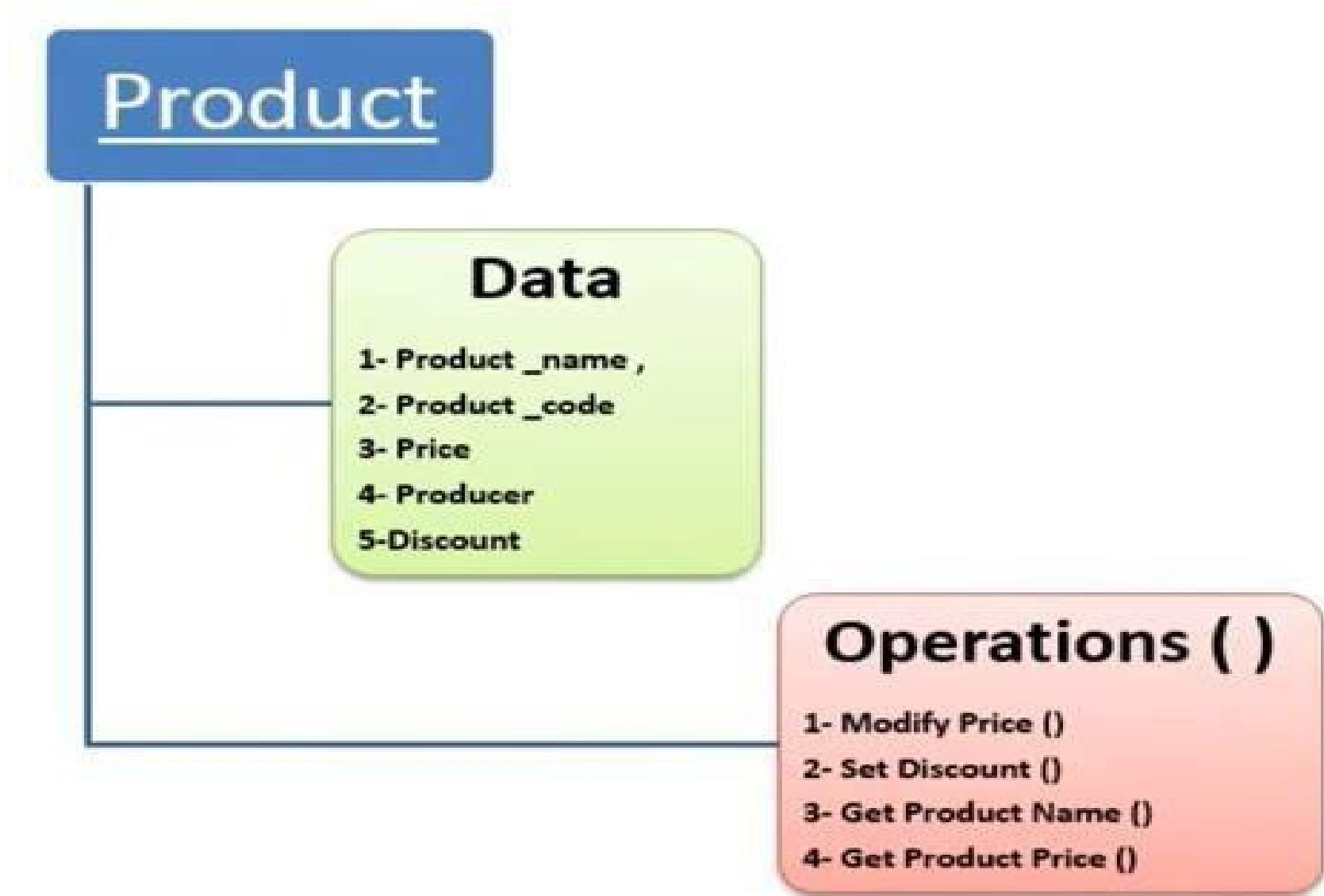


Object:

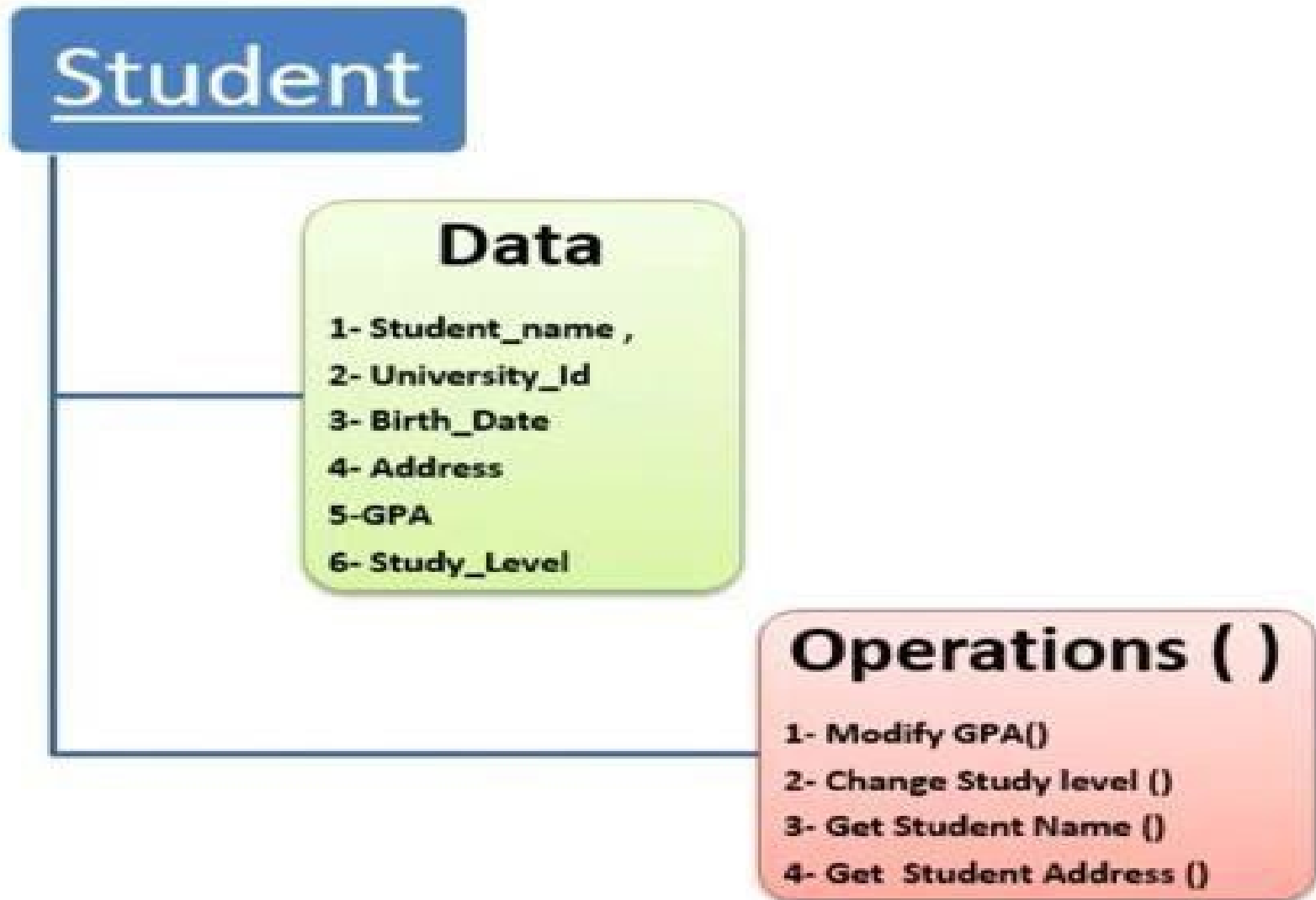
- Object is comprised of (**Data + Operations**)



Examples:



Examples:



What is class? Why we need it?

Student 1

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 2

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 4- Get Student Address ()

Student 3

Data:

- 1- Student_name ,
- 2- University_Id
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

What is class? Why we need it?

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 1

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 3

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 2

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

What is class? Why we need it?

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 1

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 3

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 2

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

What is class? Why we need it?

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

Student 1

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

Student 2

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

Student 3

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

What is class? Why we need it?

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student GPA ()

Student 1

Data:

- 1- Student_name = Ahmed
- 2- University_Id = 1050
- 5-GPA = 3.75
- 6- Study_Level = 5

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student GPA ()

Programming II

Object Oriented Programming (OOP) **Object and Class**

Dr/ Abeer Amer

Objects and Classes:

- Classes: where objects come from
 - A class is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods).
 - You can think of a class as a code “blueprint” that can be used to create a particular type of object.
- When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed.
- Each object that is created from a class is called an **instance** of the class.

Objects and Classes:

Example:

This expression creates a Scanner object in memory.

```
Scanner input = new Scanner(System.in);
```

The object's memory address
is assigned to the Input
variable.



Objects and Classes:

Example:

This expression creates a
Random object in memory.

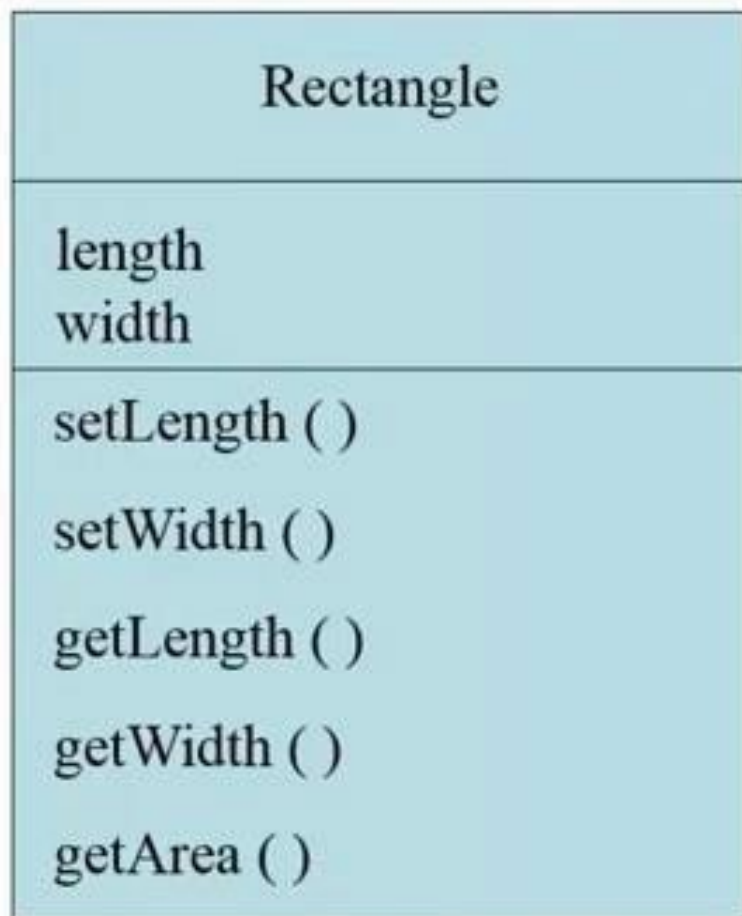
Random rand = new Random();

The object's memory address is
assigned to the rand variable.



Writing a class, step by step:

- A Rectangle class will have the following fields:



- UML Class Diagram

Writing the code:

```
public class Rectangle
{
    private double length;
    private double width;
}
```

Rectangle
length width
setLength() setWidth() getLength() getWidth() getArea()

Access Modifiers:

- An access modifier is a java keyword that indicates how a field or method can be accessed.

Public:

- When the public access modifier is applied to a class member (field or method inside the class), the member can be accessed by the code inside the class or outside.

Private:

- When the private access modifier is applied to a class member, the member cannot be accessed by the code outside the class. The member can be accessed only by methods are members of the same class.

Data Hiding:

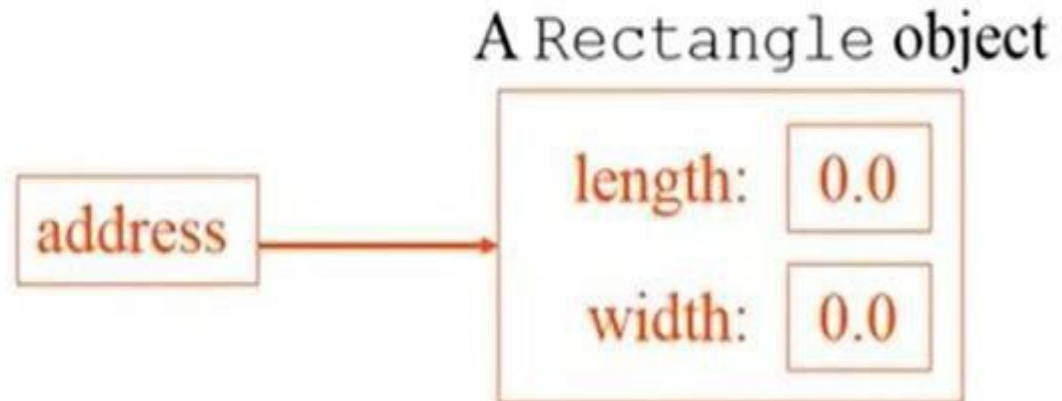
- An object hides its internal private fields from code that is outside the class that the object is an instance of.
- Only the class's methods may directly access and change the object's internal data.
- Code outside the class must use the class's public methods to operate on an object's private fields.
- Data hiding is important because classes are typically used as components in large software systems involving a team of programmers.
- Data hiding helps enforce the integrity of an object's internal data.

Example 1:

Creating a Rectangle object:

Rectangle r1 = new Rectangle ();

The r1
variable holds
the address of
the Rectangle
object.



Example 1:

```
public class Rectangle{  
    private double length;  
    public double width;  
}
```

```
public class Main  
{  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle ();  
        r1.width = 10.5;  
        r1.length = 60;  
    }  
}
```

```
Main.java:14: error: length has private access in Rectangle  
        r1.length = 60;  
            ^
```

```
1 error
```

Example 1:

```
public class Main
{
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle ();
        r1.width = -7;
    }
}
```

Example 1:

```
public class Rectangle{
    private double length;
    private double width;

    public void setlength (double l)
    {
        length = l;
    }
    public void setwidth (double w)
    {
        width = w;
    }
}
```

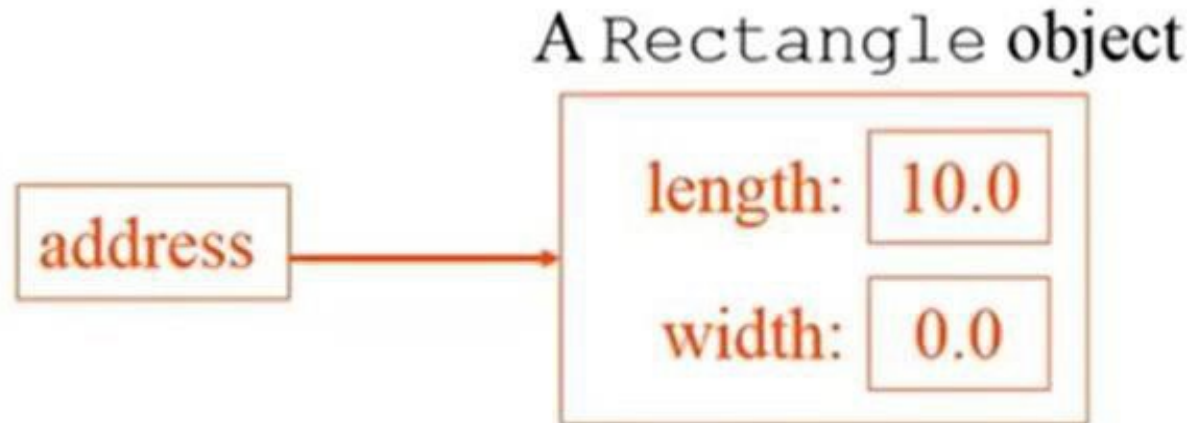
```
public class Main
{
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle ();
        r1.setlength(10);
        r1.setwidth(12.5);
    }
}
```

Calling the setlength Method:

r1.setlength(10);

- This is the state of the r1 object after the **setlength** method executes.

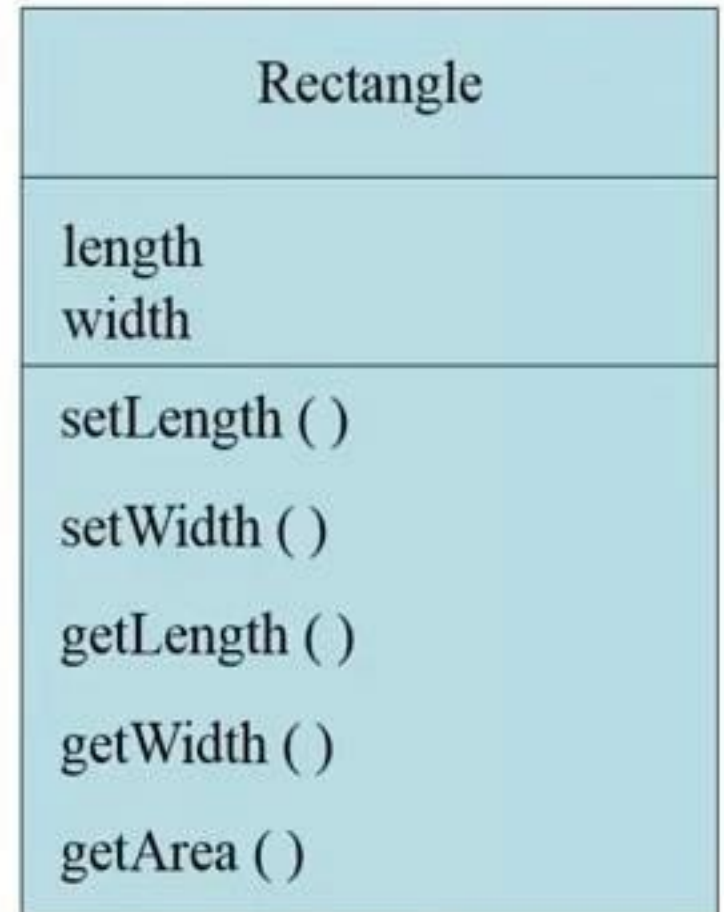
The r1
variable holds
the address of
the
Rectangle
object.



Example 1:

```
public class Rectangle{
    private double length;
    private double width;

    public void setlength (double l)
    {
        length = l;
    }
    public void setwidth (double w)
    {
        width = w;
    }
    public double getlength()
    {
        return length;
    }
    public double getwidth()
    {
        return width;
    }
    public double getarea()
    {
        return length*width;
    }
}
```



Example 1:

```
public class Main
{
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle ();
        r1.setlength(10);
        r1.setwidth(12.5);
        Rectangle r2 = new Rectangle ();
        r2.setlength(18);
        r2.setwidth (20);
        System.out.println (r1.getlength());
        System.out.println (r2.getarea());
    }
}
```

10.0

360.0

Setters (Mutators) and Getters (Accessors):

```
public class Rectangle  
{
```

```
    private double width;  
    private double length;
```

```
    public void setWidth(double w)  
    {  
        width = w;  
    }
```


```
    public void setLength(double len)  
    {  
        length = len;  
    }
```

```
    public double getWidth()  
    {  
        return width;  
    }
```

```
    public double getLength()  
    {  
        return length;  
    }
```

```
    public double getArea()  
    {  
        return length * width;  
    }
```

```
}
```



Setter , Mutator

Getter, Accessor

Uninitialized Local Reference Variables:

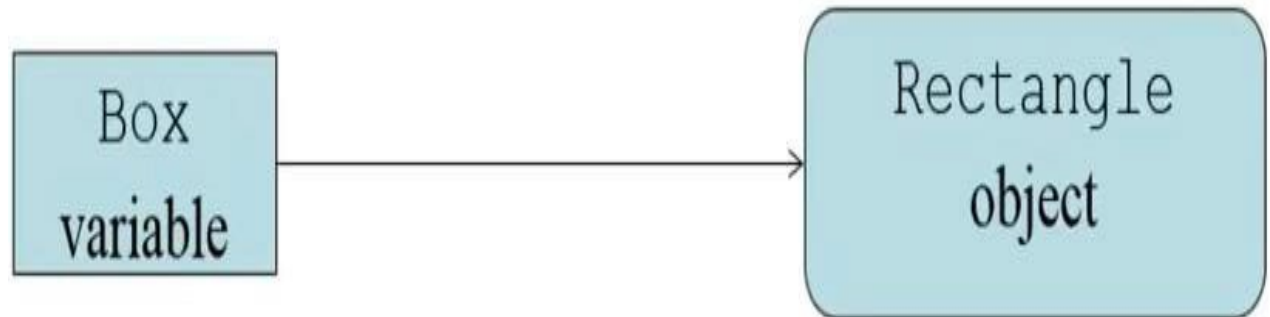
- Reference variables can be declared without being initialized

`Rectangle box;`

box: reference variable (store address of the object)

- This statement does not create a Rectangle object, so it is an uninitialized local reference variable.
- A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

`box = new Rectangle();`



`Rectangle box;`

`box = new Rectangle();`

`= Rectangle box = new Rectangle();`

Example 2:

```
public class Car{  
    private String maker;  
    private int model;  
    public void setmaker (String m)  
    {    maker = m;  
    }  
    public void setmodel (int year)  
    { model = year;  
    }  
    public String getmaker()  
    { return maker;  
    }  
    public int getmodel()  
    { return model;  
    }  
}
```

Car
<ul style="list-style-type: none">- maker- model
<ul style="list-style-type: none">+ setmaker()+ setmodel()+ getmaker()+ getmodel()

Example 2:

```
public class Main
{
    public static void main(String[] args) {
        Car c1;
        c1 = new Car();
        Car c2 = new Car();
        c1.setmaker("Honda");
        c1.setmodel(2016);
        c2.setmaker("Toyota");
        c2.setmodel (2019);
        System.out.println(c1.getmaker());
        System.out.println(c2.getmodel());
    }
}
```

Honda
2019

Example 2(Data Hiding)

```
public class Car{
    private String maker;
    private int model;
    public void setmaker (String m)
    {    if (m == "Toyota" || m== "Honda" || m=="Mercedes")
        maker = m;
        else
            System.out.println("invalid maker");
    }
    public void setmodel (int year)
    {    if (year > 2012)
        model = year;
        else
            System.out.println("invalid model");
    }
    public String getmaker()
    { return maker;
    }
    public int getmodel()
    { return model;
    }
}
```

Example 2(Data Hiding)

```
public class Main
{
    public static void main(String[] args) {
        Car c1;
        c1 = new Car();
        Car c2 = new Car();
        c1.setmaker("Honda");
        c1.setmodel(2016);
        c2.setmaker("Toyota");
        c2.setmodel (2008);
        System.out.println(c1.getmaker());
    }
}
```

invalid model
Honda

Programming II

Object Oriented Programming (OOP) **Constructors**

Dr/ Abeer Amer

Constructors:

- Classes can have special methods called constructors.
- A constructor is a method that is automatically called when an object is created.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

Constructors:

- Constructors have a few special properties that set them apart from normal methods.
 - Constructors have the same name as the class.
 - Constructors have no return type (not even void)
 - Constructors may not return any values.
 - Constructors are typically public.

Example 1:

```
public class Rectangle{
    private double length;
    private double width;

    public Rectangle ()
    { length = 10;
      width = 15;
    }

    public void setlength (double l)
    {
        length = l;
    }
    public void setwidth (double w)
    {
        width = w;
    }
    public double getlength()
    {
        return length;
    }
    public double getwidth()
    {
        return width;
    }
    public double getarea()
    {
        return length*width;
    }
}
```

Example 1:

```
public class Main
{
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle ();
        System.out.println(r1.getLength());
        System.out.println(r1.getWidth());
    }
}
```

10.0

15.0

Example 1:

```
public class Rectangle{
    private double length;
    private double width;
    public Rectangle ()
    { length = 10;
      width = 15;
      System.out.println("a new room created with 10 meter length and 15 meter width");
    }

    public void setlength (double l)
    {
        length = l;
    }
    public void setwidth (double w)
    {
        width = w;
    }
    public double getlength()
    {
        return length;
    }
    public double getwidth()
    {
        return width;
    }
    public double getarea()
    {
        return length*width;
    }
}
```

Example 1:

```
public class Main
{
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle ();
        System.out.println(r1.getLength());
        System.out.println(r1.getWidth());
    }
}
```

a new room created with 10 meter length and 15 meter width

10.0

15.0

Example 1:

```
public class Main
{
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle ();
        System.out.println(r1.getLength());
        System.out.println(r1.getWidth());
        r1.setLength(25);
        r1.setWidth(30);
        System.out.println(r1.getLength());
        System.out.println(r1.getWidth());
    }
}
```

```
a new room created with 10 meter length and 15 meter width
10.0
15.0
25.0
30.0
```

Overloading Methods and Constructors:

- Two or more methods in a class may have the same name as long as their parameter lists are different.
- When this occurs, it is called method overloading. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Overloaded Method add:

```
public int add(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}

public String add (String str1, String str2)
{
    String combined = str1 + str2;
    return combined;
}
```

Rectangle class Constructor Overload:

Rectangle r1 = new Rectangle ();

Rectangle r2 = new Rectangle (5, 10);

Example 1: (Constructor overloading)

```
public class Rectangle{  
    private double length;  
    private double width;  
    public Rectangle ()  
    { length = 10;  
      width = 15;  
    }  
    public Rectangle (double l , double w)  
    {  
        length = l;  
        width = w;  
    }  
}
```

Without Any Arguments

With two Arguments

```
public class Main  
{  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle ();  
        System.out.println(r1.getLength());  
        System.out.println(r1.getWidth());  
        System.out.println();  
        Rectangle r2 = new Rectangle (40 , 60);  
        System.out.println(r2.getLength());  
        System.out.println(r2.getWidth());  
    }  
}
```

10.0

15.0

40.0

60.0

Example 2:

```
public class Car{
    private String maker;
    private int model;
    public Car()
    { maker = "Honda";
      model = 2016;
    }
    public Car(String m,int mo )
    { maker = m;
      model = mo;
    }
    public void setmaker (String m)
    {
        maker = m;
    }
    public void setmodel (int year)
    {
        model = year;
    }
    public String getmaker()
    { return maker;
    }
    public int getmodel()
    { return model;
    }
}
```

The Default Constructor:

- When an object is created, its constructor is always called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that java provides is known as the default constructor.
 - It sets all of the object's numeric fields to 0.
 - It sets all of the object's Boolean fields to false.
 - It sets all of the object's reference variables to the special value null.

The Employee Example:

Employee

Emp_ID : Int
Name : String
Depart : String
Salary : Double
Bonus : Double
Resident : Boolean

Employee ()
Employee (Int, String)
Employee(int, String, bool)
Employee (int, String, String, double, double, bool)

Set_Salary (double): Void
Set_Salary (double, double) : Void
Set_depart(String): Void
Set_Emp_ID (int): void
Print_Emp_Data: Void

Example 3:

```
public class Employee{
    private int emp_id;
    private String ename;
    private String depart;
    private double salary;
    private double bonous;
    private boolean resident;
    public void print_emp_data()
    { System.out.println("ID "+emp_id);
      System.out.println("Name "+ename);
      System.out.println("depart "+depart);
      System.out.println("Salary "+salary);
      System.out.println("Bonous "+bonous);
      System.out.println("Resident "+resident);
    }
}
```

Example 3: (Default Constructor)

```
public class Main
{
    public static void main(String[] args) {
        Employee e1 = new Employee ();
        e1.print_emp_data();
    }
}
```

```
ID 0
Name null
depart null
Salary 0.0
Bonous 0.0
Resident false
```

Example 3: (Constructor without any arguments)

```
public class Employee{
    private int emp_id;
    private String ename;
    private String depart;
    private double salary;
    private double bonous;
    private boolean resident;
    public Employee()
    {
        emp_id = 200;
        ename = "No Name";
        depart = "Not Assigned Yet";
        salary = 2000;
        bonous = 300;
        resident = true;
    }
    public void print_emp_data()
    { System.out.println("ID "+emp_id);
      System.out.println("Name "+ename);
      System.out.println("depart "+depart);
      System.out.println("Salary "+salary);
      System.out.println("Bonous "+bonous);
      System.out.println("Resident "+resident);
    }
}
```

Example 3: (Constructor without any arguments)

```
public class Main
{
    public static void main(String[] args) {
        Employee e1 = new Employee ();
        e1.print_emp_data();
    }
}
```

```
ID 200
Name No Name
depart Not Assigned Yet
Salary 2000.0
Bonous 300.0
Resident true
```

Example 3: (Constructor Overloading)

```
public class Employee{
    private int emp_id;
    private String ename;
    private String depart;
    private double salary;
    private double bonous;
    private boolean resident;
    public Employee()
    {
        emp_id = 200;
        ename = "No Name";
        depart = "Not Assigned Yet";
        salary = 2000;
        bonous = 300;
        resident = true;
    }
    public Employee (int idno , String n)
    {
        emp_id = idno;
        ename = n;
    }
    public Employee (int idno , String n , boolean r)
    {
        emp_id = idno;
        ename = n;
        resident = r;
    }
}
```

Example 3: (Constructor Overloading)

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

        Employee e1 = new Employee (300, "Mahmoud" , true);
        e1.print_emp_data();
    }
}
```

```
ID 300
Name Mahmoud
depart null
Salary 0.0
Bonous 0.0
Resident true
```

Example 3: (Constructor Overloading)

- **this** Keyword ...>> (Constructor Chaining)

```
public Employee()
{
    emp_id = 200;
    ename = "No Name";
    depart = "Not Assigned Yet";
    salary = 2000;
    bonous = 300;
    resident = true;
}
public Employee (int idno , String n)
{
    emp_id = idno;
    ename = n;
}
public Employee (int idno , String n , boolean r)
{
    // emp_id = idno;
    // ename = n;
    this (idno , n);
    resident = r;
}
```

Example 3: (Constructor Overloading)

```
public Employee()
{
    emp_id = 200;
    ename = "No Name";
    depart = "Not Assigned Yet";
    salary = 2000;
    bonous = 300;
    resident = true;
}
public Employee (int idno , String n)
{
    emp_id = idno;
    ename = n;
}
public Employee (int idno , String n , boolean r)
{
    this (idno , n);
    resident = r;
}
public Employee (int idno , String n ,double s , double b , String d , boolean r)
{
    this (idno , n , r);
    salary = s;
    bonous = b;
    depart = d;
}
```

this 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

Example 3: (Constructor Overloading)

```
public class Main
{
    public static void main(String[] args) {
        //Employee e1 = new Employee (300, "Mahmoud" , true);
        //e1.print_emp_data();
        Employee e2 = new Employee (400, "Ahmed", 6000,500,"Accounting", true);
        e2.print_emp_data();
    }
}
```

```
ID 400
Name Ahmed
depart Accounting
Salary 6000.0
Bonous 500.0
Resident true
```

Example 3: (Set Functions)

```
public void set_salary (double s)
{
    salary = s;
}
public void set_salary (double s, double b)
{
    // salary = s;
    this.set_salary(s);
    bonous = b;
}
public void set_depart (String d)
{
    depart = d;
}
public void set_Emp_ID (int idno)
{
    emp_id = idno;
}
```

Example 3: (Set Functions)

```
public class Main
{
    public static void main(String[] args) {
        //Employee e1 = new Employee (300, "Mahmoud" , true);
        //e1.print_emp_data();
        Employee e2 = new Employee (400, "Ahmed", 6000,500,"Accounting", true);
        e2.print_emp_data();
        System.out.println("Before set functions");
        e2.set_salary (8000, 1000);
        e2.set_depart ("Attendance");
        e2.set_Emp_ID(100);
        e2.print_emp_data();
    }
}
```

```
ID 400
Name Ahmed
depart Accounting
Salary 6000.0
Bonous 500.0
Resident true
Before set functions
ID 100
Name Ahmed
depart Attendance
Salary 8000.0
Bonous 1000.0
Resident true
```

Example 3:

<i>Employee</i>
Emp_ID : Int Name : String Depart : String Salary : Double Bonus : Double Resident : Boolean
Employee () Employee (Int, String) Employee(int , String , bool) Employee (int , String, String , double ,double, bool) Set_Salary (double): Void Set_Salary (double , double) : Void Set_depart (String): Void Set_Emp_ID (int): void Print_Emp_Data : Void

Example 3:

```
public class Employee{  
    private int emp_id;  
    private String ename;  
    private String depart;  
    private double salary;  
    private double bonous;  
    private boolean resident;
```

Example 3:

```
public Employee()
{
    emp_id = 200;
    ename = "No Name";
    depart = "Not Assigned Yet";
    salary = 2000;
    bonous = 300;
    resident = true;
}

public Employee (int idno , String n)
{
    emp_id = idno;
    ename = n;
}

public Employee (int idno , String n , boolean r)
{
    this (idno , n);
    resident = r;
}

public Employee (int idno , String n ,double s , double b , String d , boolean r)
{
    this (idno , n , r);
    salary = s;
    bonous = b;
    depart = d;
}
```

Example 3:

```
public void set_salary (double s)
{
    salary = s;
}
public void set_salary (double s, double b)
{
    // salary = s;
    this.set_salary(s);
    bonous = b;
}
public void set_depart (String d)
{
    depart = d;
}
public void set_Emp_ID (int idno)
{
    emp_id = idno;
}

public void print_emp_data()
{
    System.out.println("ID "+emp_id);
    System.out.println("Name "+ename);
    System.out.println("depart "+depart);
    System.out.println("Salary "+salary);
    System.out.println("Bonous "+bonous);
    System.out.println("Resident "+resident);
}
```

Example 3:

```
public class Main
{
    public static void main(String[] args) {
        //Employee e1 = new Employee (300, "Mahmoud" , true);
        //e1.print_emp_data();
        Employee e2 = new Employee (400, "Ahmed", 6000,500,"Accounting", true);
        e2.print_emp_data();
        System.out.println("Before set functions");
        e2.set_salary (8000, 1000);
        e2.set_depart ("Attendance");
        e2.set_Emp_ID(100);
        e2.print_emp_data();
    }
}
```

Programming II

Object Oriented Programming (OOP) **Inheritance and polymorphism**

Dr/ Abeer Amer

Inheritance:

- Real life objects are typically specialized versions of other more general objects.
- The term “student” describes a very general type of students with known characteristics.
- Post-graduated students and under-graduated students are students
 - They share the general characteristics of a student.
 - However, they have special characteristics of their own.
 - Post graduated have an interesting research area.
 - Under graduated have a group number and class number.
- Post graduated students and under graduated students are special versions of a student.

Example:

Without Inheritance

Employee

- Last name
- First name
- Address
- Home phone
- Hire date
- Pay grade

- +Update address()
- +Update pay grade()

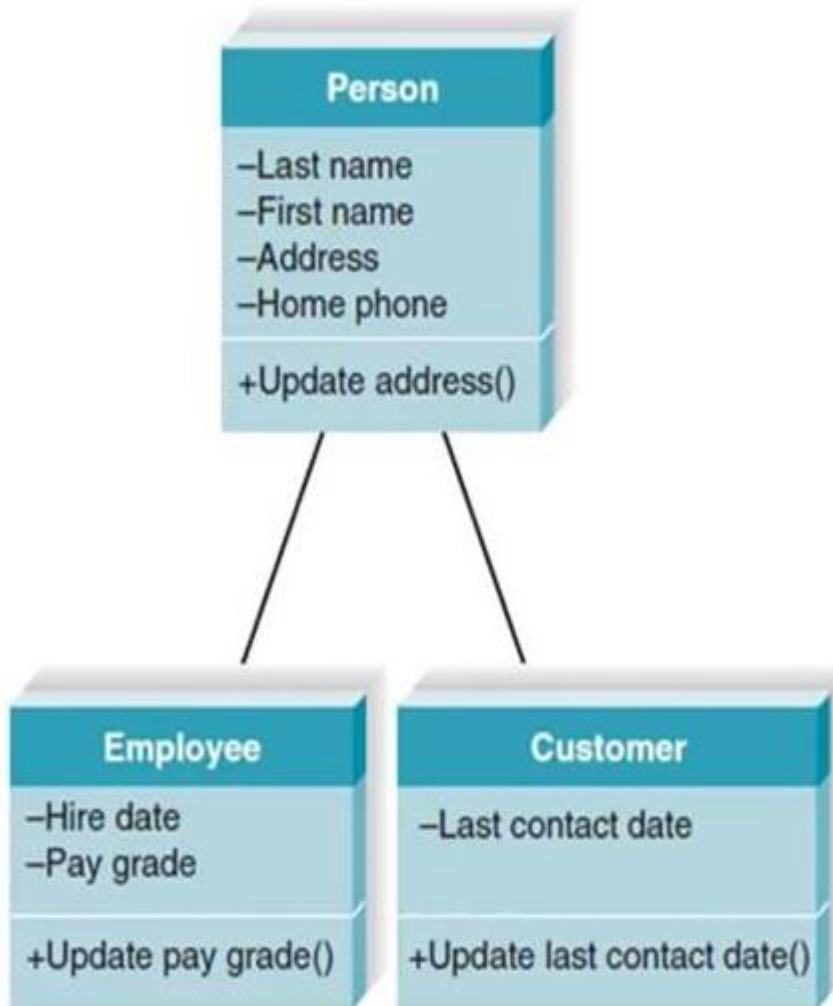
Customer

- Last name
- First name
- Address
- Home phone
- Last contact date

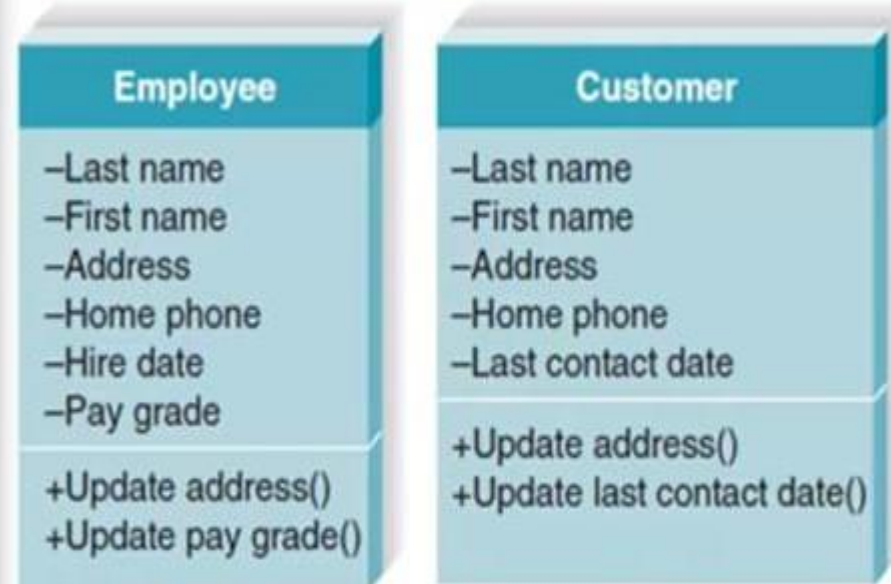
- +Update address()
- +Update last contact date()

Example:

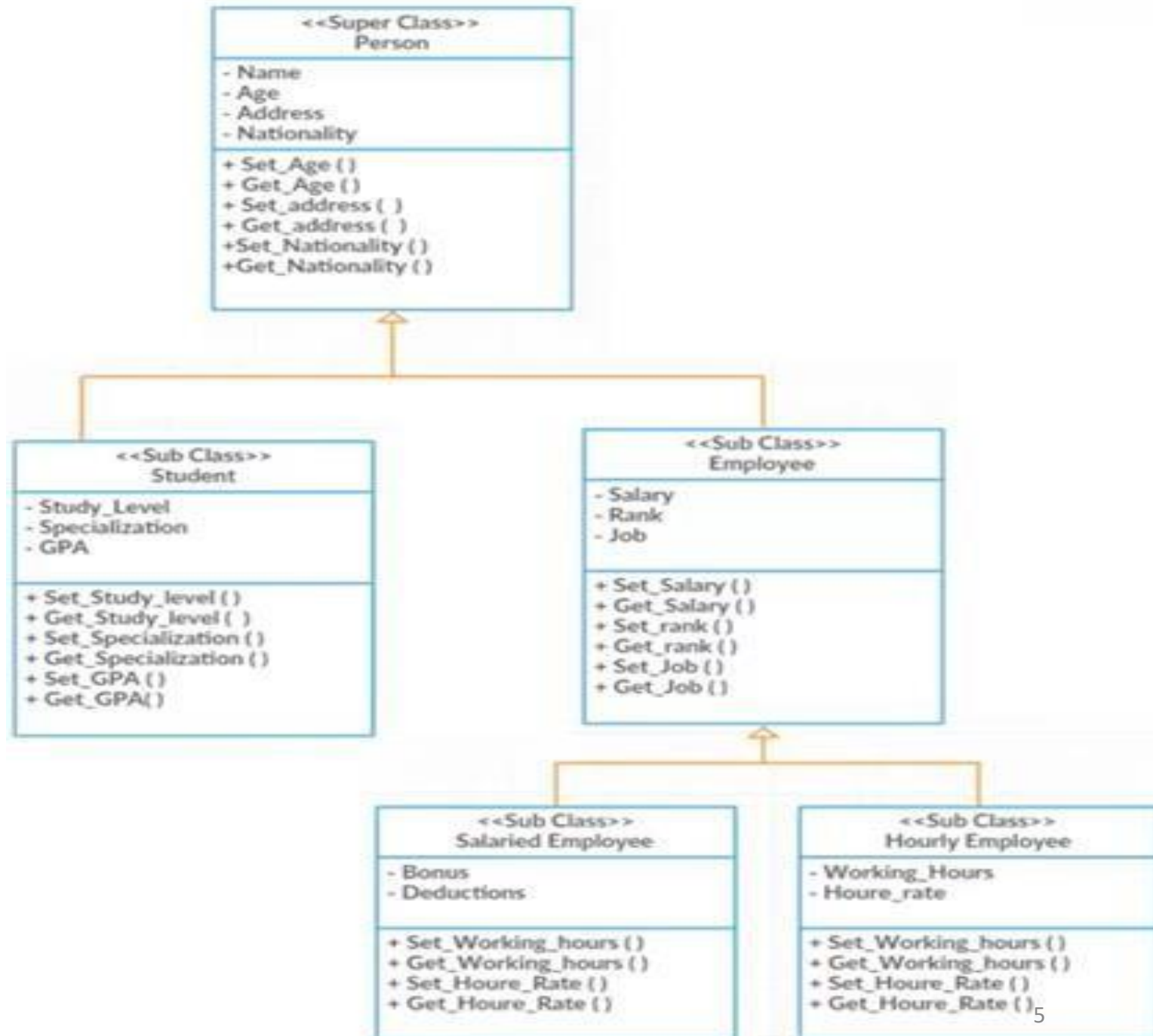
With Inheritance



Without Inheritance



Example:



The “is a “ Relationship:

- The relationship between a superclass and an inherited is called an “is a” relationship.
 - A post graduate student “is a “ student.
 - An Employee “is a” person.
 - Salaried employee “is an” Employee.
- A specialized object has:
 - All of the characteristics of the general object, plus additional characteristics that make it special.
- In object-oriented programming, inheritance is used to create an “is a” relationship among classes.

The “is a “ Relationship:

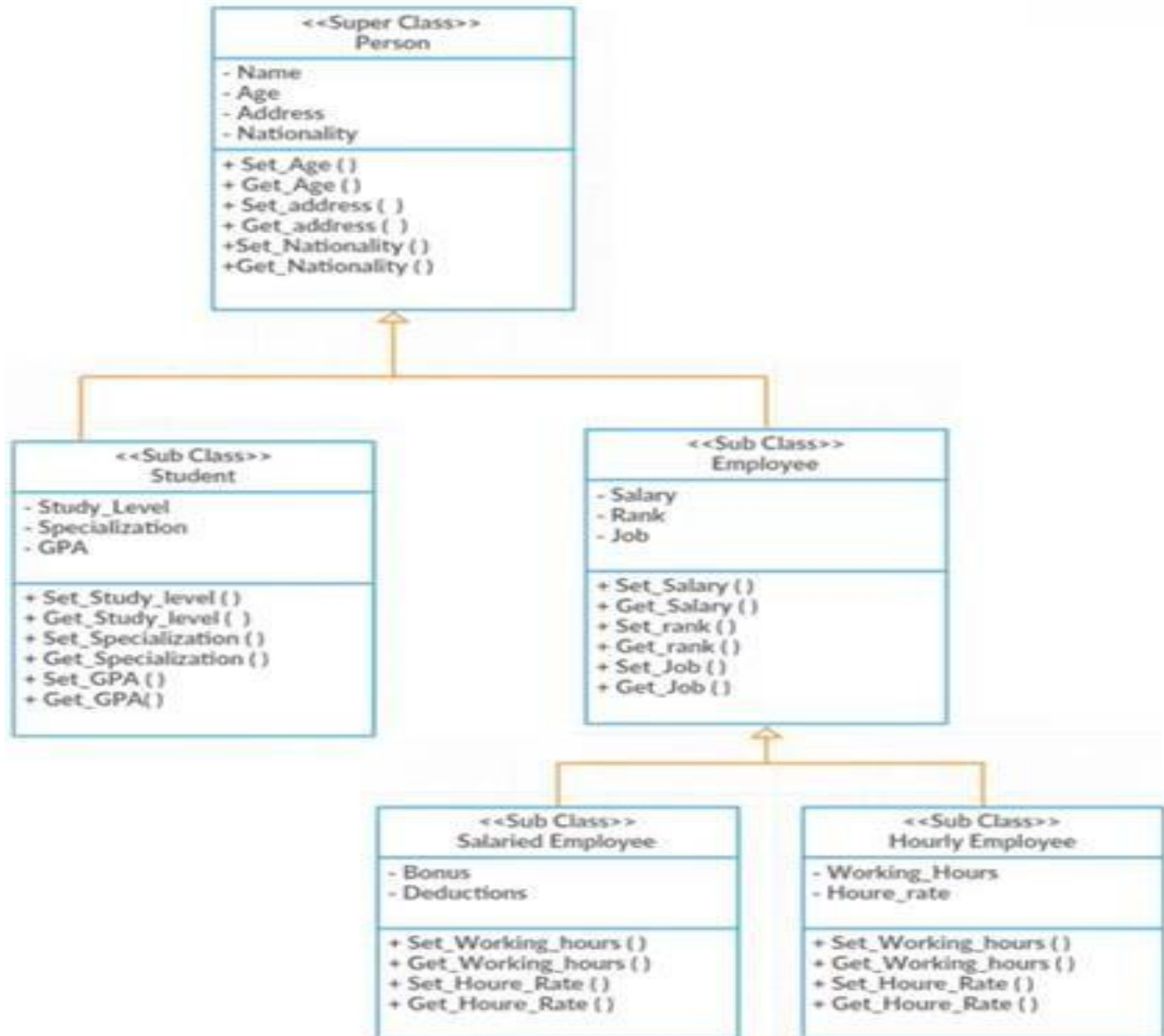
- We can extend the capabilities of a class.
- Inheritance involves a superclass and a subclass.
 - The superclass is the general class and the subclass is the specialized class.
- The subclass is based on, or extended from, the superclass.
 - Super classes are also called base classes, and subclasses are also called derived classes.
- The relationship of classes can be thought of as parent classes and child classes.

Inheritance:

- The subclass inherits fields and methods from the superclass without any of them being rewritten.
- New fields and methods may be added to the subclass.
- The Java keyword, **extends**, is used on the class header to define the subclass.

```
public class Employee extends person
```

Example:



Class person:

```
public class person
{ String name;
  double age;
  String address;
  String nationality;
  public person ()
  {
  }
  public person (String n, double a, String ad, String nat)
  { name = n;
    age = a;
    address = ad;
    nationality = nat;}
  public void set_name (String n){
    name = n;}
  public void set_age (double a){
    age = a;
  }
  public void set_address (String ad){
    address = ad;}
  public void set_nationality (String nat){
    nationality = nat;}
```



Class person:

```
public String get_name (){  
    return name;}  
public double get_age (){  
    return age;}  
public String get_address (){  
    return address;}  
public String get_nationality (){  
    return nationality;}  
}
```

Class student:

```
public class Student extends person
{
}

```

Test Program:

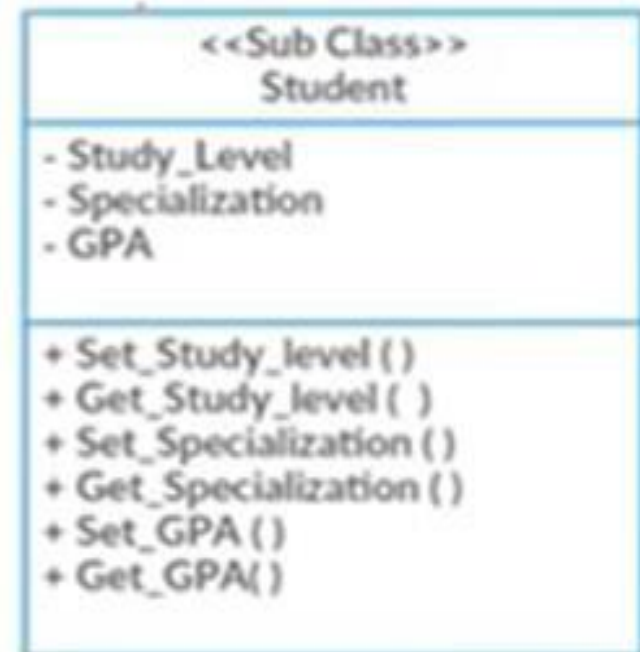
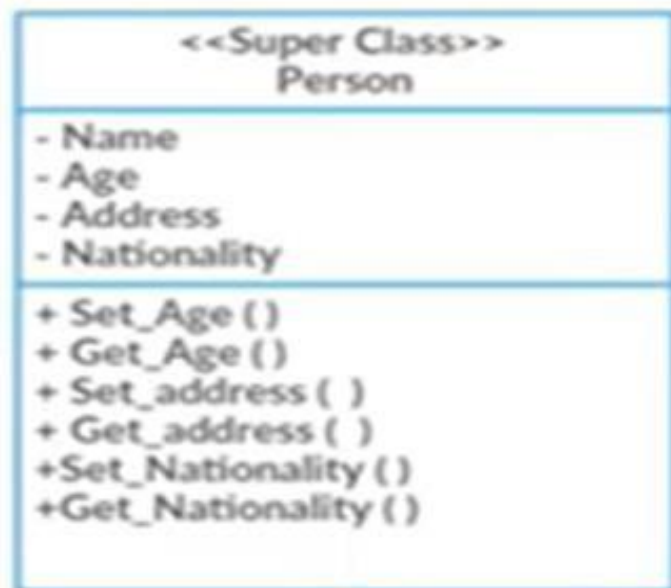
```
public class Main
{
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.set_age (25);
        System.out.println("The age is "+s1.get_age());
    }
}

```

The age is 25.0

Class student (Create the attributes)

```
public class Student extends person
{ private int study_level;
  private String Specialization;
  private double GPA;
  public Student ()
  {
  }
}
```



Class student (Create the constructor)

Super Keyword:

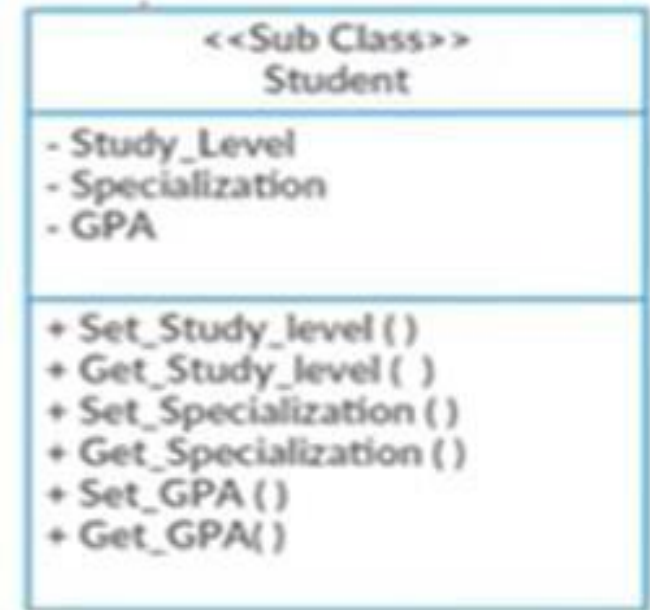
(reusability)

```
/*public Student (String n,double a , String ad, String nat, int lev, String special, double gpa)
{
    name = n;
    age = a;
    nationality = nat;
    address = ad;
    study_level = lev;
    Specialization = special;
    GPA = gpa;
} */
```

```
public Student (String n,double a , String ad, String nat, int lev, String special, double gpa)
{
    super (n,a,ad,nat);    // used to call the constructor of super class
    study_level = lev;
    Specialization = special;
    GPA = gpa;
}
}
```

Class Student (Complete Attributes and methods)

```
public class Student extends person
{ private int study_level;
  private String Specialization;
  private double GPA;
  public Student ()
  {
  }
}
```



```
public Student (String n,double a , String ad, String nat, int lev, String special, double gpa)
{ super (n,a,ad,nat);    // used to call the constructor of super class
  study_level = lev;
  Specialization = special;
  GPA = gpa;
}
```

Class Student (Complete Attributes and methods)

```
public void set_GPA(double GPA)
{ //GPA = GPA;      (Confusion between attribute and argument)
  this.GPA = GPA;   //(this.GPA>>>attribute - GPA>>> argument)
}

public void set_studylevel(int study_level)
{ this.study_level = study_level;
}

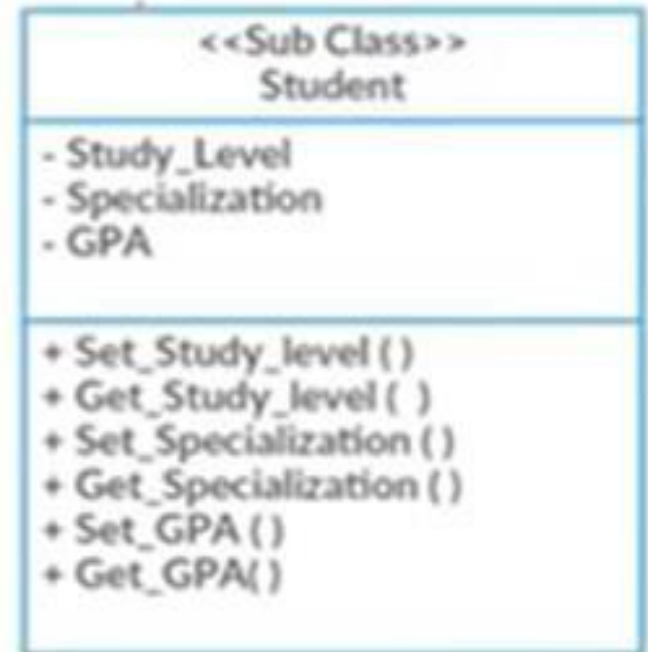
public void set_GPA(String Specialization)
{ this.Specialization = Specialization;
}

public double get_GPA()
{ return GPA;
}

public int get_studylevel()
{ return study_level;
}

public String get_Specialization()
{ return Specialization;
}

}
```



Test Program:

```
public class Main
{
    public static void main(String[] args) {
        Student s1 = new Student("Ahmed", 20, "Alex", "Egyptian", 2, "AI", 3.2);
        System.out.println("The name is "+s1.get_name());
        System.out.println("The nationality is "+s1.get_nationality());
        System.out.println("The specialization is "+s1.get_Specialization());
    }
}
```

Output:

```
The name is Ahmed
The nationality is Egyptian
The specialization is AI
```

Inheritance, Fields and Methods:

- Members of the superclass that are marked private:
 - are not inherited by the subclass, and
 - may only be accessed from the subclass by public methods of the superclass.
- Members of the superclass that are marked public:
 - are inherited by the subclass, and
 - may be directly accessed from the subclass.

Inheritance and Constructors:

- Constructors are not inherited.
- The superclass constructor can be explicitly called from the subclass using the **super** keyword.
- When a subclass is instantiated (create object from subclass), the superclass default constructor is executed first.
 >> incase of object without any arguments

Note:

- When a subclass is instantiated (create object from subclass), the superclass default constructor is executed first.

Superclass:

```
public class person
{ String name;
  double age;
  String address;
  String nationality;
  public person () // default constructor without arguments
  { System.out.println ("Iam the Base Class Constructor");
  }
  public person (String n, double a, String ad, String nat)
  { name = n;
    age = a;
    address = ad;
    nationality = nat;}
}
```

Subclass:

```
public class Student extends person
{ private int study_level;
  private String Specialization;
  private double GPA;
  public Student ()
  {
    System.out.println ("Iam the Derived Class Constructor");
  }

  public Student (String n,double a , String ad, String nat, int lev, String special, double gpa)
  { super (n,a,ad,nat);      // used to call the constructor of super class
    study_level = lev;
    Specialization = special;
    GPA = gpa;
  }
}
```

Test program:

```
public class Main
{
  public static void main(String[] args) {
    Student s1 = new Student();
  }
}
```

Iam the Base Class Constructor
Iam the Derived Class Constructor

Test Program:

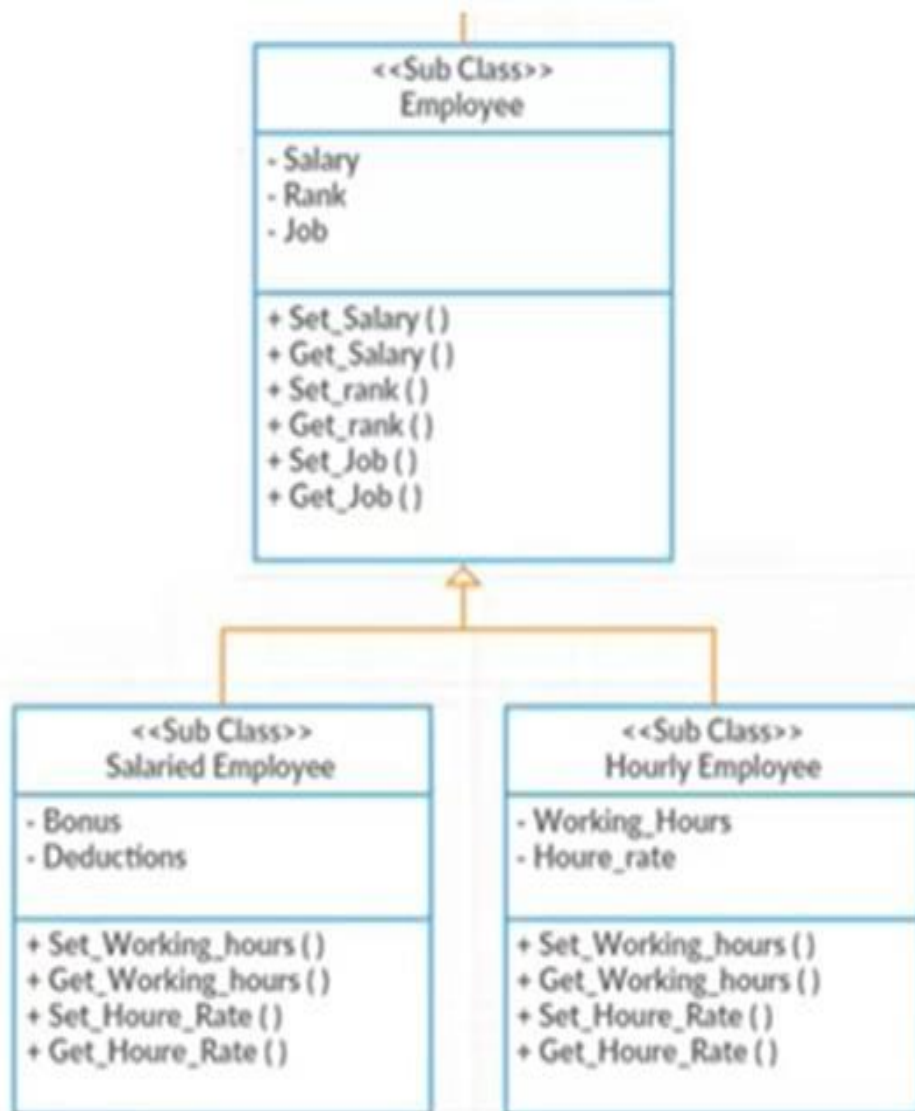
```
public class Main
{
    public static void main(String[] args) {
        Student s1 = new Student("Ahmed", 20, "Alex", "Egyptian", 2, "AI", 3.2);
        System.out.println("The name is "+s1.get_name());
        System.out.println("The nationality is "+s1.get_nationality());
        System.out.println("The specialization is "+s1.get_Specialization());
        Student s2 = new Student();
    }
}
```

```
The name is Ahmed
The nationality is Egyptian
The specialization is AI
Iam the Base Class Constructor
Iam the Derived Class Constructor
```

Overriding Superclass Methods:

- A subclass may have a method with the same signature as a superclass method.
- The subclass method overrides the superclass method.
- This is known as *method overriding*.
- A subclass method that overrides a superclass method must have the same signature as the superclass method.
- An object of the subclass invokes the subclass's version of the method, not the superclass's.
- The *@Override* annotation should be used just before the subclass method.

Example:



Employee

```
public double get_salary()
{
    return salary;
}
```

Salaried Employee

```
public double get_salary()
{
    return salary + bonus - deductions ;
}
```

Hourly Employee

```
public double get_salary()
{
    return working_hours * hours_rate ;
}
```

Example: Class Employee:

```
public class Employee extends person
{double salary;
  String rank;
  String Job;
public Employee()
{
}
public Employee (String n,double a , String ad, String nat, double sal, String r, String J)
{ super (n,a,ad,nat);
  salary = sal;
  rank = r;
  Job = J;
}
public void set_salary (double s)
{
  salary = s;
}
public double get_salary ()
{
  return salary;
}
}
```

Example: Class Salaried Employee:

```
public class Salaried_Employee extends Employee
{
    double bonous;
    double deduction;

    public Salaried_Employee()
    {
    }

    public Salaried_Employee (String n,double a , String ad, String nat, double sal, String r, String J, double b, double d)
    {
        super (n,a,ad,nat,sal,r,J);
        bonous = b;
        deduction = d;
    }


    @Override
    public double get_salary ()
    {
        return salary+bonous-deduction;
    }
}
```

Example:

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        Salaried_Employee sal1 = new Salaried_Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer",1000,500);
        System.out.println(sal1.get_salary());
    }
}
```

Output:



5500.0

Example:

Class Person:

```
public void print_all_details()
{
    System.out.println("Name = "+name+"\nAge = "+age+"\nAddress= "+address+"\nNationality = "+nationality);
}
```

Class Employee:

```
@Override
public void print_all_details()
{ super.print_all_details();
  System.out.println("Job = "+Job+"\nRank = "+rank+"\nSalary= "+salary);
}
}
```

Example:

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        Employee E1 = new Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer");
        E1.print_all_details();
    }
}
```

Output:

```
Name = Ahmed
Age = 25.0
Address= Alex
Nationality = Egyptian
Job = Engineer
Rank = manager
Salary= 5000.0
```

Reference Type & Object Type:

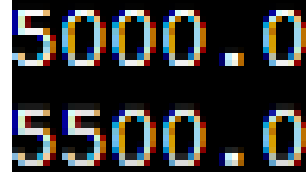
reference_type reference_variable = new object_type()

- You can create an object from a reference type superclass and an object type subclass.
- You can't create an object from a reference type superclass and an object type subclass.
- If you create an object from a reference type superclass and an object type subclass, you can use only the methods in the superclass.

Reference Type & Object Type:

```
public class Main
{
    public static void main(String[] args) {
        // Reference Type is Employee and Object Type is Employee
        Employee E1 = new Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer");
        // Reference Type is Employee and Object Type is Salaried_Employee
        Employee E2 = new Salaried_Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer",1000,500);
        System.out.println(E1.get_salary());
        System.out.println(E2.get_salary());
    }
}
```

Output:



```
5000.0
5500.0
```

Reference Type & Object Type:

```
public class Main
{
    public static void main(String[] args) {
        // Reference Type is Employee and Object Type is Employee
        Employee E1 = new Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer");
        // Reference Type is Salaried_Employee and Object Type is Employee
        Salaried_Employee E2 = new Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer");
    }
}
```

Output:

```
Main.java:15: error: incompatible types: Employee cannot be converted to Salaried_Employee
    Salaried_Employee E2 = new Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer");
                        ^
1 error
```

Reference Type & Object Type:

Salaried Employee Class:

```
public void special_function()
{

}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        // Reference Type is Employee and Object Type is Employee
        Employee E1 = new Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer");
        // Reference Type is Employee and Object Type is Salaried_Employee
        Employee E2 = new Salaried_Employee ("Ahmed",25,"Alex","Egyptian",5000,"manager","Engineer",1000,500);

        E2.special_function();
    }
}
```

Output:

```
Main.java:17: error: cannot find symbol
        E2.special_function();
           ^
    symbol:   method special_function()
    location: variable E2 of type Employee
1 error
```

Preventing a Method from Being Overridden:

- The final modifier will prevent the overriding of a superclass method in a subclass.

Public final void display()

- If a subclass attempts to override a final method, the compiler generates an error.
- This ensures that a particular superclass method is used by subclasses rather than a modified version of it.

Example:

Employee Class:

```
public final double get_salary ()  
{  
    return salary;  
}
```

Salaried Employee Class:

```
@Override  
public double get_salary ()  
{  
    return salary+bonous-deduction;  
}
```

Run from main method:

```
Salaried_Employee.java:16: error: get_salary() in Salaried_Employee cannot override get_salary() in Employee  
public double get_salary ()  
                ^  
    overridden method is final  
1 error
```

Protected Members:

- Using **protected** instead of private makes some tasks easier.
- Any class that is derived from the class, or is in the same package, has unrestricted access to the protected member.
- Any method in the same package may access the protected member.
- It is always better to make all fields **private** and then provide **public** methods for accessing those fields.

Example:

```
public class Shape
{
    private double height; // To hold height.
    private double width; //To hold width or base

    /**
     * The setValue method sets the data
     * in the height and width field.
     */
    public void setValues(double height, double width)
    {
        this.height = height;
        this.width = width;
    }
}
```

```
public class Rectangle extends Shape
{
    /**
     * The method returns the area
     * of rectangle.
     */
    public double getArea()
    {
        return height * width; //accessing protected members
    }
}
```

- **Generates error.**
- **The solution is protected member.**

Example:

```
public class Shape
{
    Protected double height; // To hold height.
    Protected double width; //To hold width or base

    /**
     * The setValue method sets the data
     * in the height and width field.
     */
    public void setValues(double height, double width)
    {
        this.height = height;
        this.width = width;
    }
}
```

```
public class Rectangle extends Shape
{

    /**
     * The method returns the area
     * of rectangle.
     */
    public double getArea()
    {
        return height * width; //accessing protected members
    }
}
```

Example:

```
public class Shape
{private double length;
 private double width;
 public Shape()
 {
 }
 public Shape (double length,double width)
 { this.length = length;
   this.width = width;
 }
}
```

```
Rectangle.java:8: error: length has private access in Shape
    return length*width;
           ^
Rectangle.java:8: error: width has private access in Shape
    return length*width;
           ^
2 errors
```

```
public class Rectangle extends Shape
{
    public Rectangle ()
    {
    }
    public Rectangle (double length,double width)
    { super (length,width);
    }
    public double get_area ()
    {
        return length*width;
    }
}
```


Example:

```
public class Shape
{protected double length;
  protected double width;
  public Shape()
  {
  }
  public Shape (double length,double width)
  { this.length = length;
    this.width = width;
  }
}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        Rectangle r = new Rectangle (6,10);
        System.out.println(r.get_area());
    }
}
```

Output:



60.0

Programming II

Object Oriented Programming (OOP) **Abstract Class and Interfaces**

Dr/ Mahmoud Gamal

Abstract Classes:

- An abstract class cannot be instantiated, but other classes are derived from it.
- An Abstract class serves as a superclass for other classes.
- The abstract class represents the generic or abstract from of all the classes that are derived from it.
- A class becomes abstract when you place the abstract keyword in the class definition:

public abstract class(class name)....

Abstract Methods:

- An abstract method is a method that appears in a superclass, but expects to be overridden in a subclass.
- An abstract method has no body and must be overridden in a subclass.

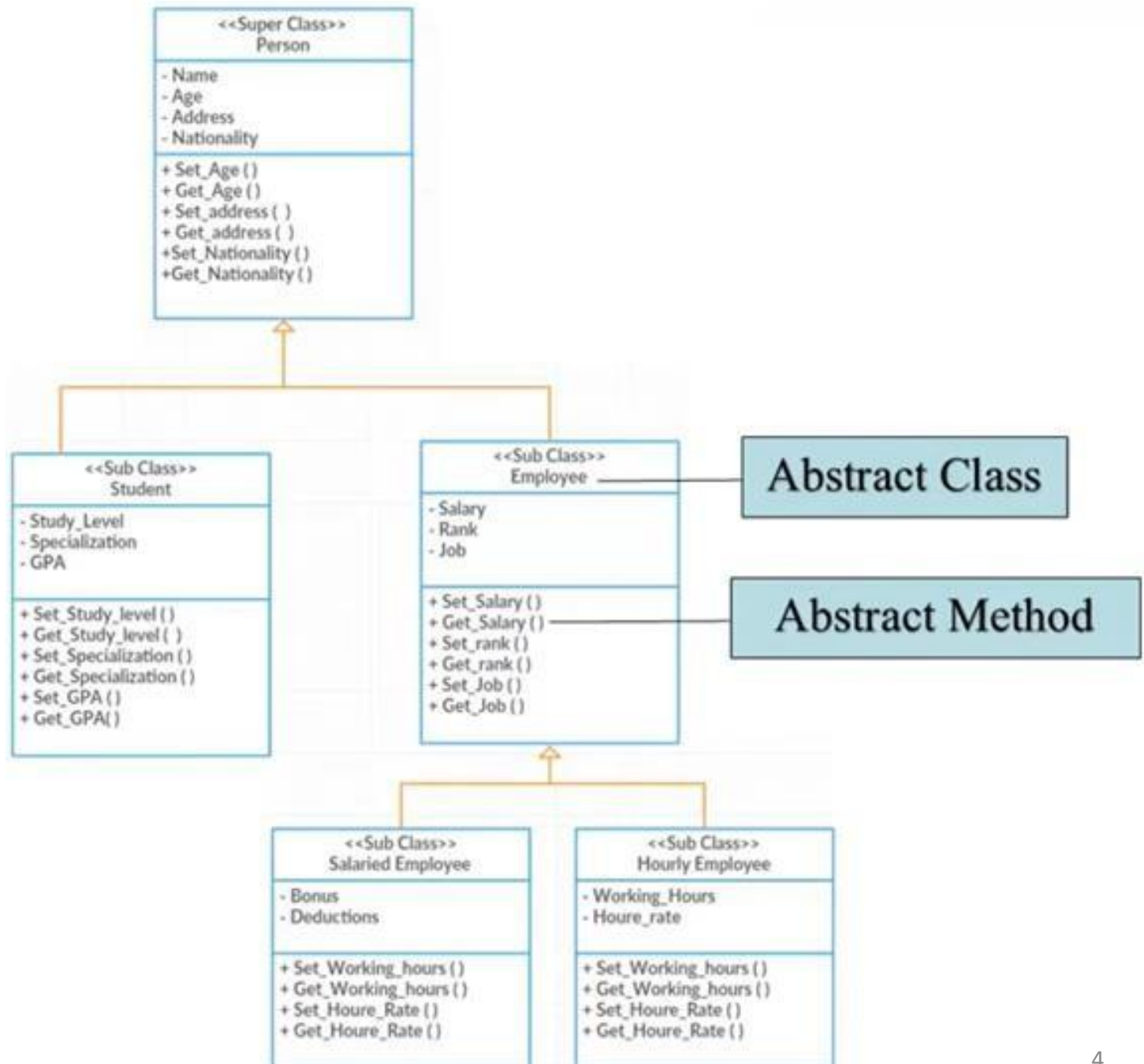
Access_specifier abstract return_type method_name (parameter list)

Ex: public abstract void get_salary();

(function header or function prototype)

- Abstract methods are used to ensure that a subclass implements the method.
- If a subclass fails to override an abstract method, a compiler error will result.

Example:



Abstract Class Employee:

```
public abstract class Employee extends person
{double salary;
  String rank;
  String Job;
public Employee()
{
}
public Employee (String n,double a , String ad, String nat, double sal, String r, String J)
{ super (n,a,ad,nat);
  salary = sal;
  rank = r;
  Job = J;
}
public void set_salary (double s)
{
  salary = s;
}

public abstract double get_salary ();  // In abstract method, write only method prototype
}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        Employee E1 = new Employee ();
    }
}
```

Output:

```
Main.java:12: error: Employee is abstract; cannot be instantiated
        Employee E1 = new Employee ();
                        ^
```

```
1 error
```

Salaried Employee Class:

```
public class Salaried_Employee extends Employee
{
    double bonous;
    double deduction;

    public Salaried_Employee()
    {
    }

    public Salaried_Employee (String n, double a, String ad, String nat, double sal, String r, String J, double b, double d)
    {
        super (n, a, ad, nat, sal, r, J);
        bonous = b;
        deduction = d;
    }
}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        Salaried_Employee sal1 = new Salaried_Employee ();
    }
}
```

Output:

```
Salaried_Employee.java:2: error: Salaried_Employee is not abstract and does not override abstract method get_salary() in Employee
public class Salaried_Employee extends Employee
      ^
1 error
```

Salaried Employee Class:

```
public class Salaried_Employee extends Employee
{
    double bonous;
    double deduction;

    public Salaried_Employee()
    {
    }

    public Salaried_Employee (String n,double a , String ad, String nat, double sal, String r, String J, double b, double d)
    {
        super (n,a,ad,nat,sal,r,J);
        bonous = b;
        deduction = d;
    }

    @Override
    public double get_salary ()
    {
        return salary+bonous-deduction;
    }

}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        Salaried_Employee sal1 = new Salaried_Employee ("Ahmed", 20, "Alex", "Egyptian", 6000, "manager", "Engineer", 2000, 500);
        System.out.println(sal1.get_salary());
    }
}
```

Output:

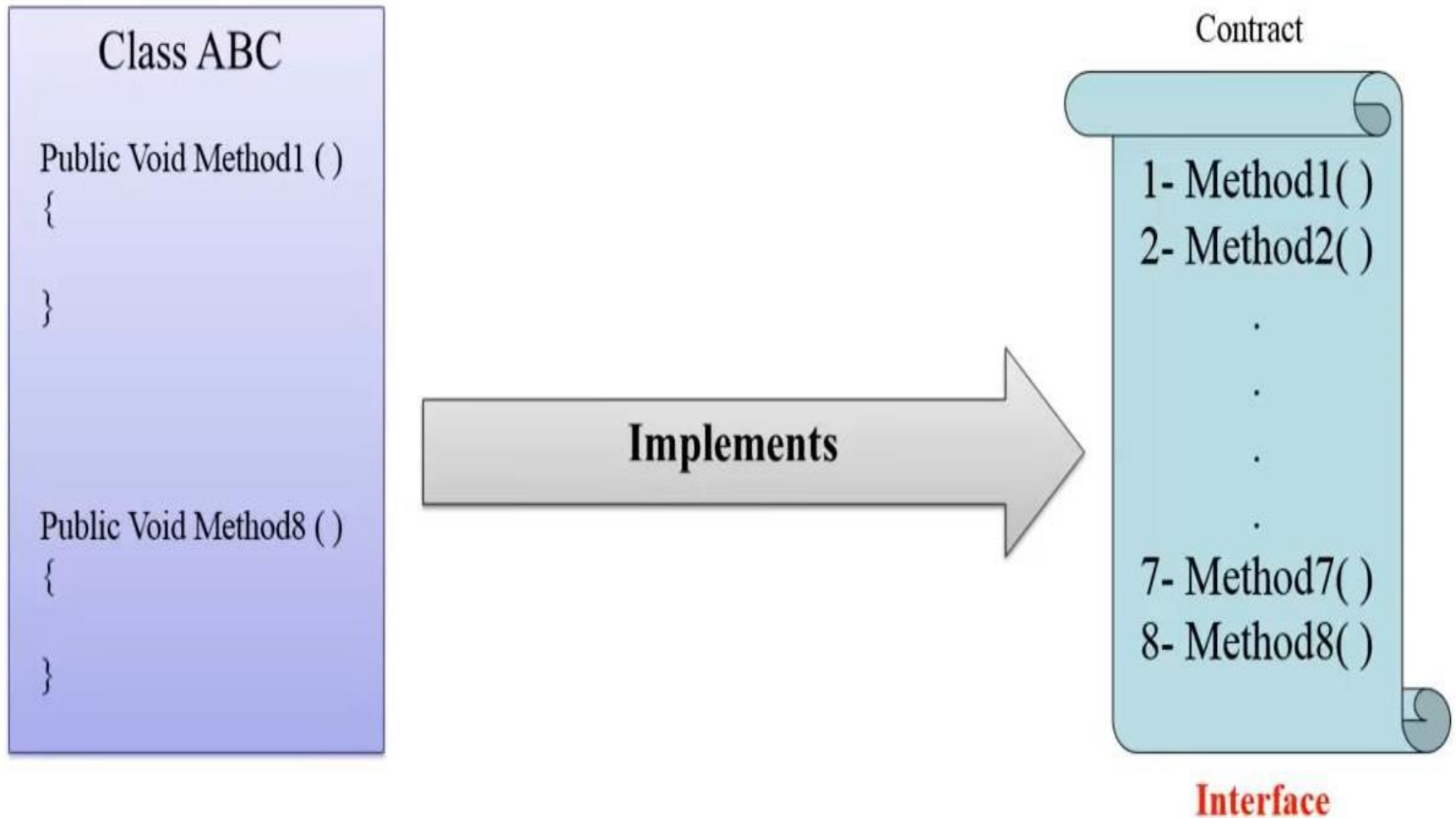
7500.0

Interfaces:

- An interface is similar to an abstract class that has all abstract methods.
- It cannot be instantiated, and all of the methods listed in an interface must be written elsewhere.
- The purpose of an interface is to specify behavior for other classes.
- It is often said that an interface is like a contract, and when a class implements an interface it must adhere to the contract.
- The general format of an interface definition:

```
public interface InterfaceName  
{  
    Method headers  
}
```

Interfaces:



Interfaces:

- If a class implements an interface, it uses the **implements** keyword in the class header.

```
public interface Retail_Item  
{  
    Method headers  
}
```

```
public class CD implements Retail_Item  
public class Book implements Retail_Item
```

Example:

Interface Retail Item:

```
public interface Retail_Item
{
    public double getitemprice();
}
```

Class CD:

```
public class CD implements Retail_Item
{
}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        Retail_Item r1 = new Retail_Item();
    }
}
```

Output:

```
Main.java:12: error: Retail_Item is abstract; cannot be instantiated
    Retail_Item r1 = new Retail_Item();
                        ^
1 error
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        CD cd = new CD();
    }
}
```

Output:

```
CD.java:1: error: CD is not abstract and does not override abstract method getitemprice() in Retail_Item
public class CD implements Retail_Item
      ^
1 error
```

Class CD:

```
public class CD implements Retail_Item
{
    double price;
    String title;
    String artist;
    public CD()
    {
    }
    public CD(double price, String title, String artist)
    {
        this.price = price;
        this.title = title;
        this.artist = artist;
    }
    public double getitemprice()
    {
        return price;
    }
    public void setttitle(String title)
    {
        this.title = title;
    }
}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        CD cd = new CD();
        cd.getitemprice();
        Retail_Item r1 = new CD();
        r1.getitemprice();
    }
}
```

Output:

No error



Test Program:

```
public class Main
{
    public static void main(String[] args) {
        CD cd = new CD();
        cd.getitemprice();
        Retail_Item r1 = new CD();
        r1.getitemprice();
        r1.settitle("*****");
    }
}
```

Output:

```
Main.java:16: error: cannot find symbol
        r1.settitle("*****");
           ^
symbol:   method settitle(String)
location: variable r1 of type Retail_Item
1 error
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        CD cd = new CD();
        cd.getitemprice();
        Retail_Item r1 = new CD();
        r1.getitemprice();
        cd.settitle("*****");
    }
}
```

Output:

No error



Implementing Multiple Interfaces:

- A class can be derived from only one superclass.
- Java allows a class to implement multiple interfaces.
- When a class implements multiple interfaces, it must provide the methods specified by all of them.
- To specify multiple interfaces in a class definition, simply list the name of interfaces separated by commas , after the implements keyword.

```
public class lecture implements Interface1, Interface2, Interface3
```

Example:

Retail Item interface:

```
public interface Retail_Item
{
    public double getitemprice();
}
```

Displayable interface:

```
public interface Displayable
{
    public void display();
}
```

CD Class:

```
public class CD implements Retail_Item, Displayable
{
    double price;
    String title;
    String artist;
    public CD()
    {
    }
    public CD(double price, String title, String artist)
    {
        this.price = price;
        this.title = title;
        this.artist = artist;
    }
    public double getitemprice()
    {
        return price;
    }
}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        CD cd = new CD();
    }
}
```

Output:

```
CD.java:2: error: CD is not abstract and does not override abstract method display() in Displayable
public class CD implements Retail_Item,Displayable
       ^
1 error
```

CD Class:

```
public class CD implements Retail_Item, Displayable
{
    double price;
    String title;
    String artist;
    public CD()
    {
    }
    public CD(double price, String title, String artist)
    {
        this.price = price;
        this.title = title;
        this.artist = artist;
    }
    public double getitemprice()
    {
        return price;
    }
    public void display ()
    {
        System.out.println("title = "+title+"price = "+price+"artist = "+artist);
    }
}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        CD cd = new CD();
    }
}
```

Output:

No error



Fields in Interfaces:

- An interface can contain field declarations.
- All fields in an interface are treated as **final and static**.
- Because they automatically become final, you must provide an initialization value.
- In this interface, FIELD1 and FIELD2 are final static int variables.

```
public interface Doable
{
    int FIELD1 = 1, FIELD2 = 2;
    (Method headers...)
}
```

- Any class that implements this interface has access to these variables.

Example:

Retail item interface:

```
public interface Retail_Item
{
    String storename = "ABC";
    public double getitemprice();
}
```

Displayable interface:

```
public interface Displayable
{
    public void display();
}
```

CD Class:

```
public class CD implements Retail_Item, Displayable
{
    double price;
    String title;
    String artist;
    public CD()
    {
    }
    public CD(double price, String title, String artist)
    {
        this.price = price;
        this.title = title;
        this.artist = artist;
    }
    public double getitemprice()
    {
        return price;
    }
    public void display ()
    {
        System.out.println("Store = "+storename+"\ntitle = "+title+"\nprice = "+price+"\nartist = "+artist);
    }
}
```

Test Program:

```
public class Main
{
    public static void main(String[] args) {
        CD cd = new CD();
        cd.display();
    }
}
```

Output:

```
Store = ABC
title = null
price = 0.0
artist = null
```