UNIT-II

Methods and classes

Unit – II	Contact Hours = 8 Hours
Methods and classes: methods, returning from a method, returning a value, using parameters, constructors, parameterized constructors, the new operator revisited, garbage collection and finalizers, this keyword, controlling access to class members, pass objects to methods, argument passing, returning objects, method overloading.	

Methods

- In Java, a method is a block of code that performs a specific task and can optionally return a value.
- Methods are used to encapsulate and reuse code, making it easier to read and maintain. They can be defined within a class or interface and can be called by other parts of the program.
- Methods can also take parameters, which are values passed to the method when it is called, and can have different access levels, such as public or private.
- Methods are defined using the following syntax:

```
type name(parameter-list) {
    // body of method
}
```

Adding the method to the class

- To add a method to a class in Java, the method must be defined within the class using the proper syntax.
- The method's access level, return type, name, and parameter list are specified, followed by the method's code block
- Here is an example of how a method called "add" can be added to a class called "Calculator":

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    // other class members
}
```

- Once the method is defined within the class, it can be called by other parts of the program by creating an instance of the class and using the dot notation to access the method.

```
Calculator calculator = new Calculator();
int result = calculator.add(3, 4);
```

```
//Add Range to Vehicle class
class Vehicle{
   int passengers;
   int fulecap;
   int mpg;
   //display the range
   void range(){
      System.out.println("Range is "  + fulecap * mpg);
        }
}
public class AddMeth {
   public static void main( String[] args){
      Vehicle minivan= new Vehicle();
      Vehicle sportscar =new Vehicle();
      int range1,range2;
      //assign values to fields in sportscar
      minivan.passengers=7;
      minivan.fulecap=16;
      minivan.mpg=21;
 //assigining values to field in sportscar
        sportscar.passengers=2;
        sportscar.fulecap=14;
        sportscar.mpg=12;
             System.out.print("minivan can
carry" +minivan.passengers+ ".");
        minivan.range();//display range of
minivan
        System.out.print("minivan can carry"
+ sportscar.passengers+ ".");
        sportscar.range();//display range of
sports car
   }
}
```

OUTPUT:

```
minivan can carry7.Range is 336
sportscar can carry2.Range is 168
```

Returning from a method:

- Two conditions can cause a method to return

1. When the method's closing brace is encountered
2. When a return statement is executed.

- There are 2 forms of return
    1. One for use in void methods (which do not return a value)
    2. Other for returning values

- In a void method, we can terminate a method by using the following form of return:

```
return;
```

- When this statement executes, program control returns to the caller, skipping any remaining code in the method.

- For ex:

```
void sample( ){
 for(int i=0; i<10; i++) {
     if(i==5)
           return; //stop at 5
           System.out.println(i);
    }
}
```

Returning a Value:

- Methods return a value to the calling routing using this form of return**:**

return value;

Using Parameters:

- Value passed to a method is called an argument. Variable that receives the argument is called a parameter.
- Parameters are declared inside the parentheses that follow the method's name.

For ex:

```
class ckNum{
    //return true if x is even
     boolean isEven(int x){

         if((x%2)==0) return true;
         else

         return false;

    }
}
```

```
Class Demo{
        public static void main(String[ ] args){
        ckNum e=new ckNum();
        if(e.isEven(10))
             System.out.println("10 is even");
             if(e.isEven(9))
                System.out.println("9 is even");
     }
}
```

Constructors

- A constructor in Java is a special type of method that is used to initialize an object when it is created.
- It has the same name as the class and does not have a return type.

Rules for creating java constructor:

- It has the same name as the class in which it resides and is syntactically similar to a method.
- They have no explicit return type.

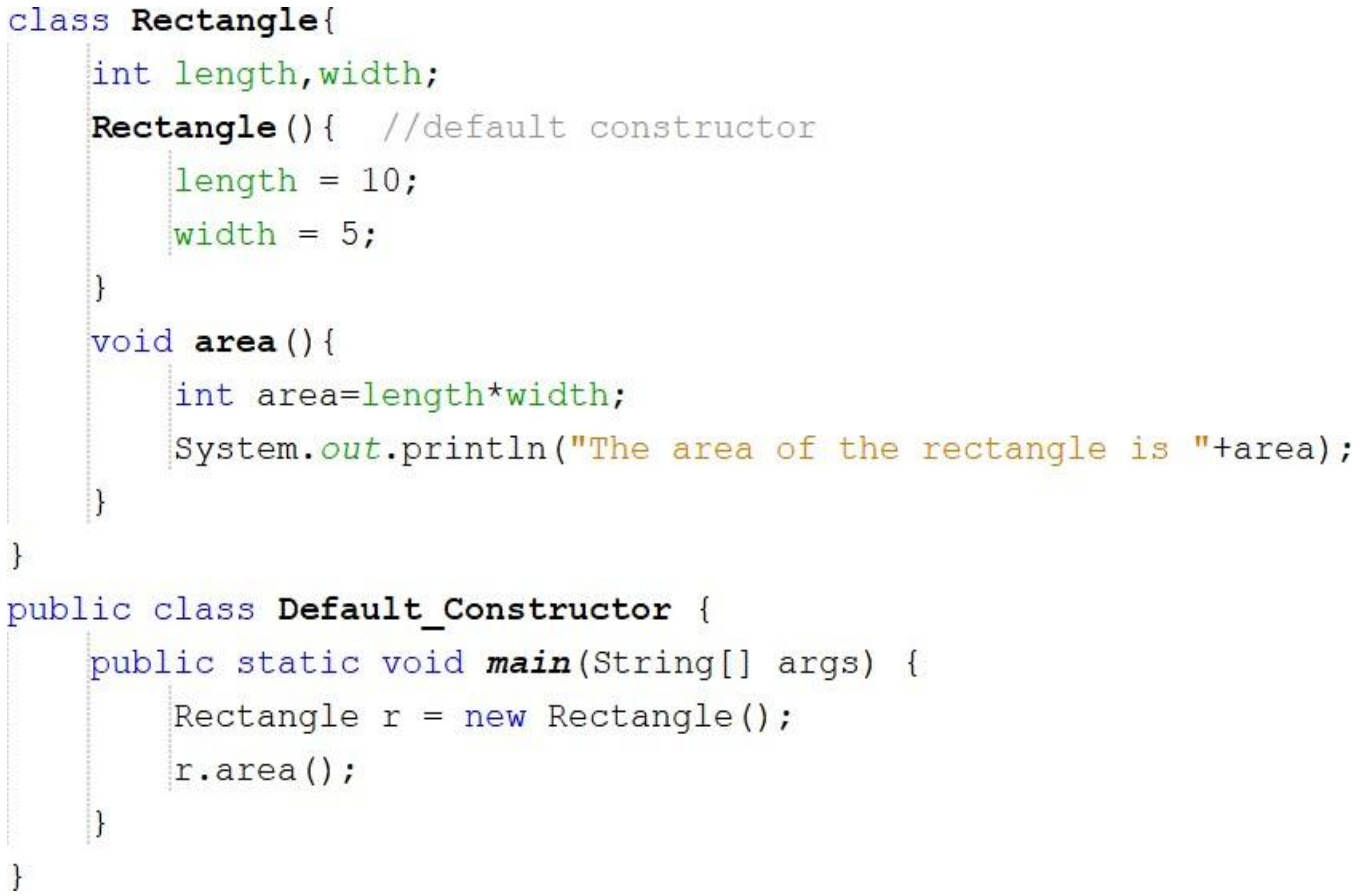
There are two types of constructors:

- Default constructor (no-arg constructor)
- Parameterized constructor

Default Constructor:

- Constructor which does not take any argument is called default constructor.

```
class MyClass{
      int x;
      MyClass( ) {     //default constructor for MyClass
      x=10 ;   }
}
Class ConsDemo {
Public static void main(String[ ] args ) {
      MyClass t1= new MyClass( );
      MyClass t2= new MyClass( );
      System.out.println(t1.x + “ “ +t2.x);
      }
}
```



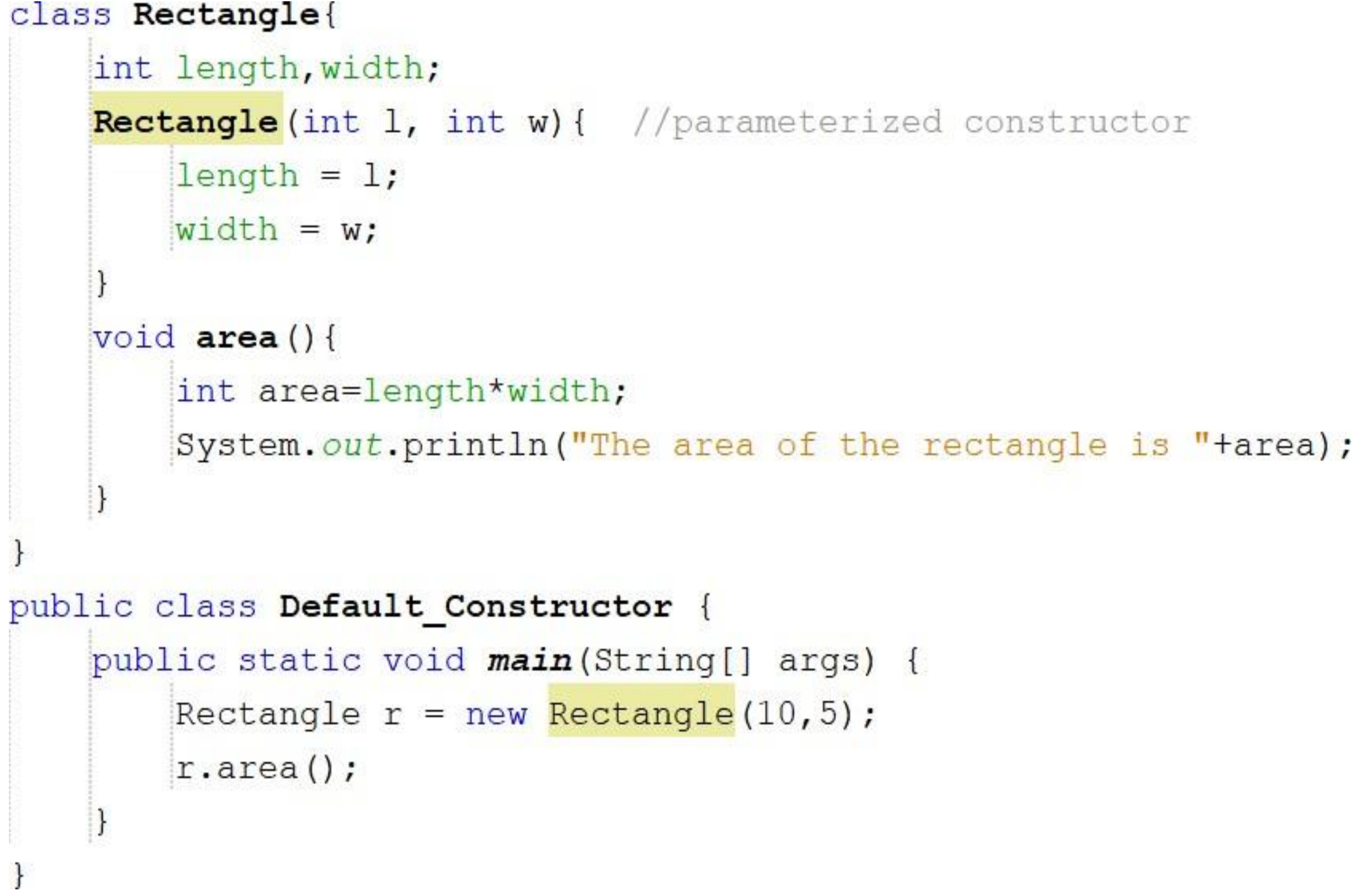
```java
class Rectangle{
    int length,width;
    Rectangle(){  //default constructor
        length = 10;
        width = 5;
    }
    void area(){
        int area=length*width;
        System.out.println("The area of the rectangle is "+area);
    }
}
public class Default_Constructor {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.area();
    }
}
```

Parameterized Constructors

- A parameterized constructor in Java is a constructor that accepts one or more arguments/parameters.
- It allows you to initialize an object with specific values when it is created. Here is an example:

```
class MyClass{
int x;
MyClass( int i ) {  // This is a constructor for MyClass which has a parameter
x=i;    }
}
Class ParmConsDemo {
Public static void main(String[] args) {
MyClass t1= new MyClass(10);
MyClass t2= new MyClass(88);
System.out.println(t1.x + " " +t2.x);
}
}
```

- Out Put of the program is:
- 10 88



```
class Rectangle{
    int length,width;
    Rectangle(int l, int w){  //parameterized constructor
        length = l;
        width = w;
    }
    void area(){
        int area=length*width;
        System.out.println("The area of the rectangle is "+area);
    }
}
public class Default_Constructor {
    public static void main(String[] args) {
        Rectangle r = new Rectangle(10,5);
        r.area();
    }
}
```

ADDING Constructors for Vehicle Class

```
class Vehicle{
        int passengers;
        int fulecap;
        int mpg;
Vehicle(int p, int f, int m) { //constructor for Vehicle
passangers=p;
fuleCap=f;
mpg=m; }
//Return the range
int range(){
return mpg*fuleCap; }
//compute fule needed for a given distance.
double fuleNeeded(int miles){
return(double) miles/mpg; }
}
class VehicleConstructorDemo{
public static void main(String[] args){
        Vehicle minivan= new Vehicle(7, 16, 21);
        Vehicle sportscar= new Vehicle(2, 14, 12);
        double gallons;
        int dist =252;
```

```
gallons=minivan.fuleNeeded(dist);
        System.out.println("To go "+ dist + "miles minivan
needs" + gallons + "gallons of fule");

gallons=sportscar.fuleNeeded(dist);
        System.out.println("To go "+ dist + "miles
sportscar needs" + gallons + "gallons of fule");
}
```

the new operator revisited

new operator has this general form:

class_var = new class_name(arg_list);

- class_var is a variable of the class type being created.
- The class_name is the name of the class that is being instantiated
- The class_name followed by a parenthesized argument list specifies the constructor for the class.
- If a class does not define its own constructor, new will use the default constructor supplied by Java
- The new can be used to create an object of any class type.
- It returns a reference to the newly created object, which is assigned to class-var

Garbage collection and finalizers

- A you have seen objects are dynamically allocated from a pool of free memory by using the **new operator**.

- Memory is not infinite, and free memory can be exhausted, hence memory has to be freed after its usage.

- In Java, **garbage collection is a mechanism that automatically frees objects that are no longer being used by the program**,

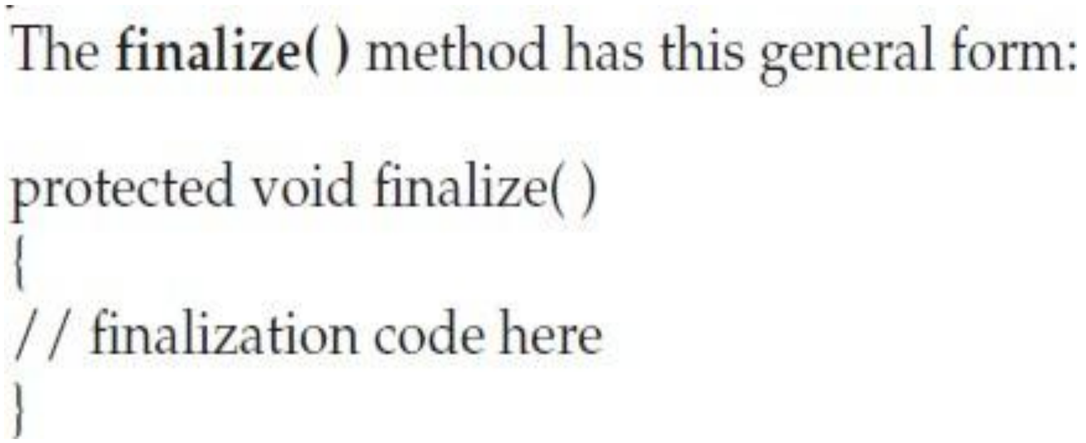
- It work like this: When no references to an object exist that object is assumed to be no longer needed, and the memory occupied by the object is released . This memory can then be used for a subsequent allocation.
- The garbage collector will usually run only when two conditions are met:
- There are objects to recycle, and there is a need to recycle them..

The Finalize() Method

- Finalizers are a mechanism for releasing resources associated with an object before it is garbage collected. i.e **finalizer** will be called just before an object's final destruction by the **garbage collector.**

- They are implemented as a method called "finalize()", and it can be used in very specialized cases to ensure that object terminates cleanly.

- However, the use of finalizers is generally discouraged because they can make it difficult to predict when resources will be released, and they can lead to performance problems if not used carefully.



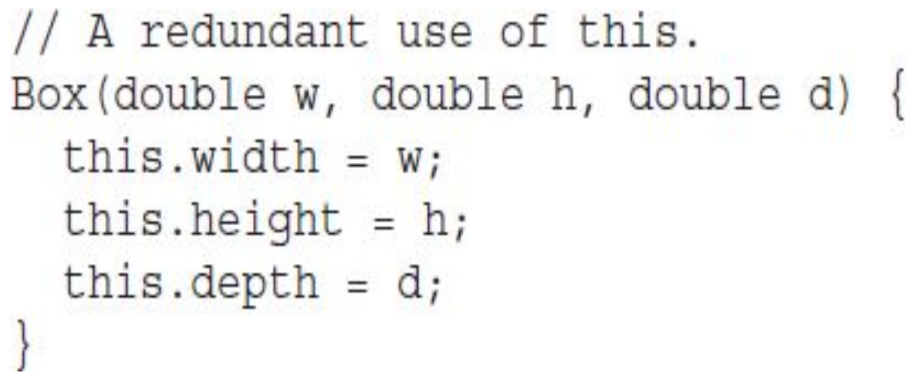
The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- To add a finalizer to a class, you simply define the finalize( ) method.
- The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( ) method, actions that must be performed before an object is destroyed will be specified.
- **The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects**
- The keyword **protected is a specifier that prevents access to finalize( ) by code defined outside its class**
- It is important to understand that finalize( ) is only called just prior to garbage collection.

The this Keyword

- this keyword can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked



```
// A redundant use of this.
Box(double w, double h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

- The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object. While it is redundant in this case, this is useful in other contexts

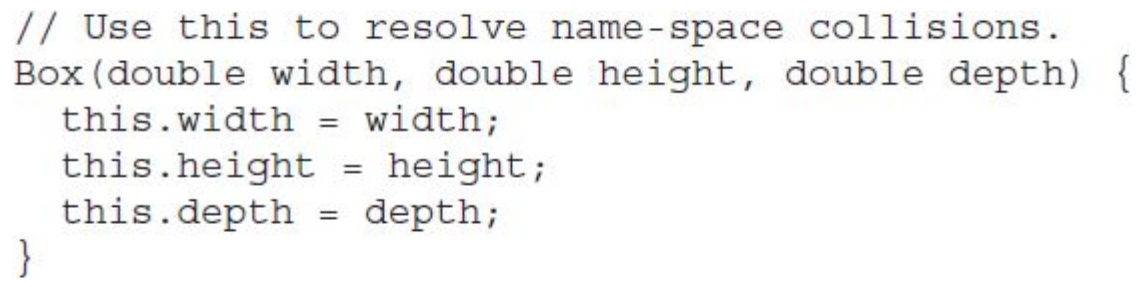
The this Keyword cont…

The this Keyword:

- It is used to refer the current object.
- It is a reference to an object.
- It is used to resolve any Namespace collision between instance variables & local variables.
- In java , you can have local variables including formal parameters to method to overlap. When this is the case, local variables hide the instance variables.
- The problem of instance variable hiding can be overcome by the use of **this** keyword.

Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class’ instance variables
- When a local variable has the same name as an instance variable, the local variable hides the instance variable
- For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name

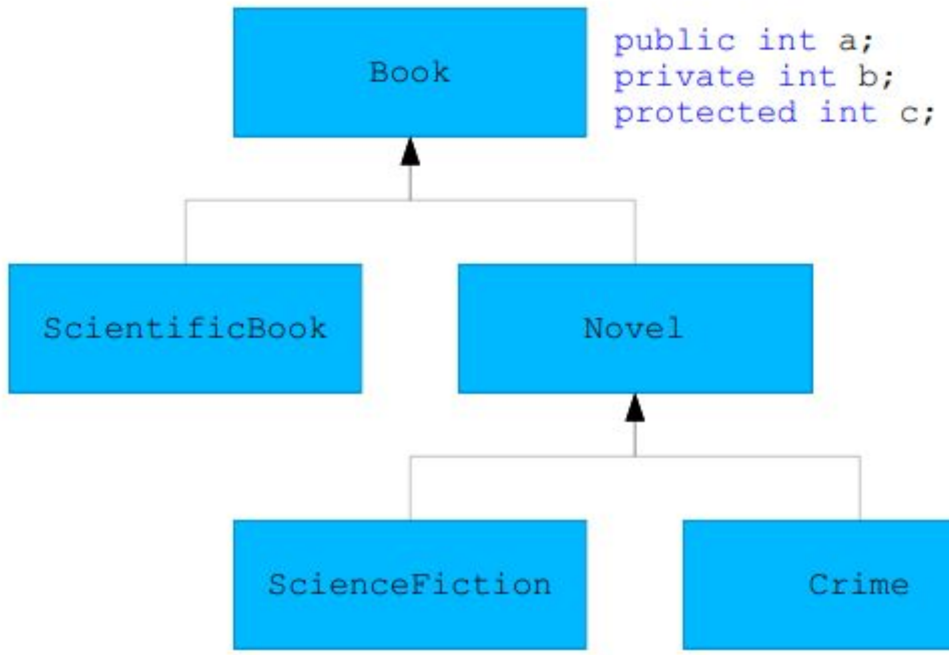


```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

- Use this to overcome the instance variable hiding

Controlling access to class members

- It is possible to control the access to methods and variables from other classes with the modifiers:
  - **public**
  - **private**
  - **protected**



Controlling Access to Class Members

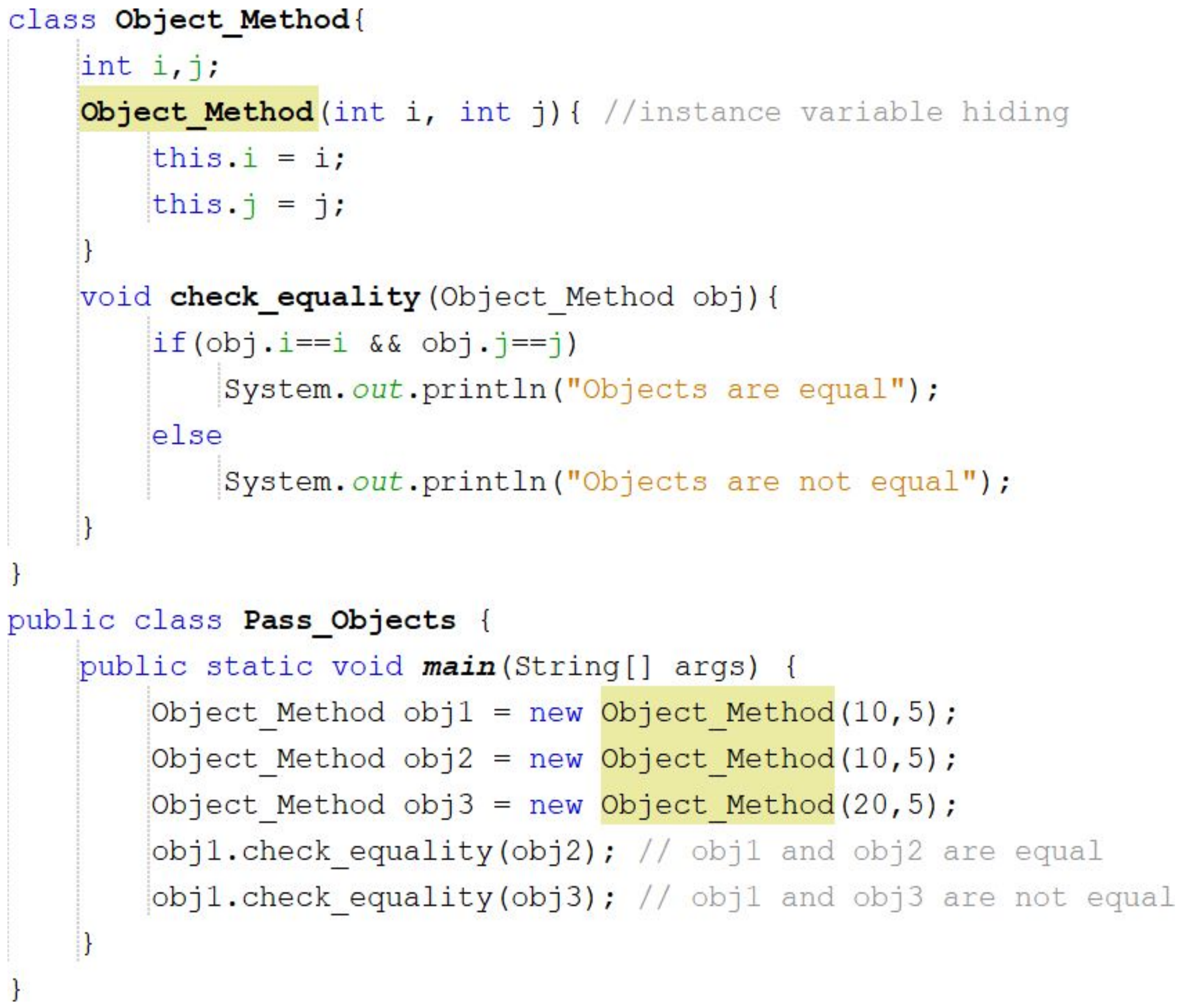
- A member can be accessed is determined by the access specifier that modifies its declaration.
- Member access control is achieved through the use of three access specifiers/modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved
- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- For example,

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
  int a; // default access
  public int b; // public access
  private int c; // private access

  // methods to access c
  void setc(int i) { // set c's value
    c = i;
  }
  int getc() { // get c's value
    return c;
  }
}

class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

    // These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

    // This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

- Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**

Pass Objects to Methods

- So far, we have only been using primitive types such as **int** as parameters to methods. However, it is both correct and common to pass objects to methods.
- For example,



```
class Object_Method{
    int i,j;
    Object_Method(int i, int j){ //instance variable hiding
        this.i = i;
        this.j = j;
    }
    void check_equality(Object_Method obj){
        if(obj.i==i  && obj.j==j)
            System.out.println("Objects are equal");
        else
            System.out.println("Objects are not equal");
    }
}
public class Pass_Objects  {
    public static void main(String[] args)  {
        Object_Method obj1 = new Object_Method(10,5);
        Object_Method obj2 = new Object_Method(10,5);
        Object_Method obj3 = new Object_Method(20,5);
        obj1.check_equality(obj2); // obj1 and obj2 are equal
        obj1.check_equality(obj3); // obj1 and obj3 are not equal
    }
}
```

Output:

```
 Objects are equal
Objects are not equal
```

- The **check_equality( )** method inside **Object_Method** compares two objects for equality and displays the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method displays objects are equal. Otherwise, it displays objects are not equal

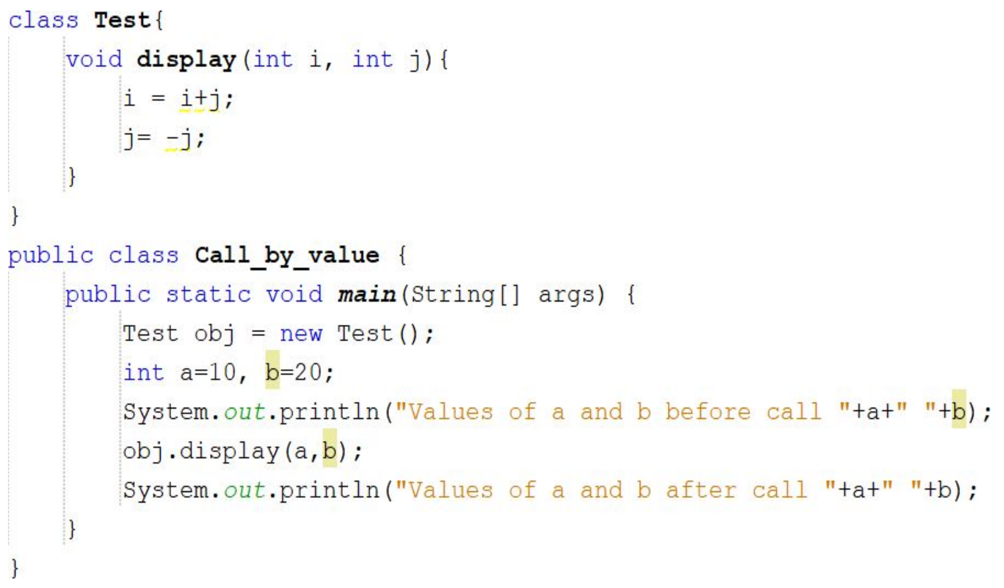
How Arguments are Passed

- There are two ways in which an argument can be passed to a subroutine/method
- **Call-by-value:**
  - Copies the value of an argument into the formal parameter of the subroutine.
  - Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- **Call-by-reference:**
  - Reference to an argument (not the value of the argument) is passed to the parameter
  - Inside the subroutine, this reference is used to access the actual argument specified in the call. Changes made to the parameter will affect the argument used to call the subroutine.

- //Call by value and call by reference.

```
class a{
        int a=5;
        int b=7;
        int sum(int c , int d)   //method header
        {
        int e= c + d ;
        return e ; }
        void add( int f, int g)
        {
        System.out.print(“sum=“+(f+g));
        }    }
class b{
        public static vid main(String[] args){
        a ob=new a();
        int h= ob.sum(15,10);//call by value
        System .out.println(h);
        Ob.add(ob.a,ob.b);//call by reference
        }   }
```

- Example for **call-by-value**



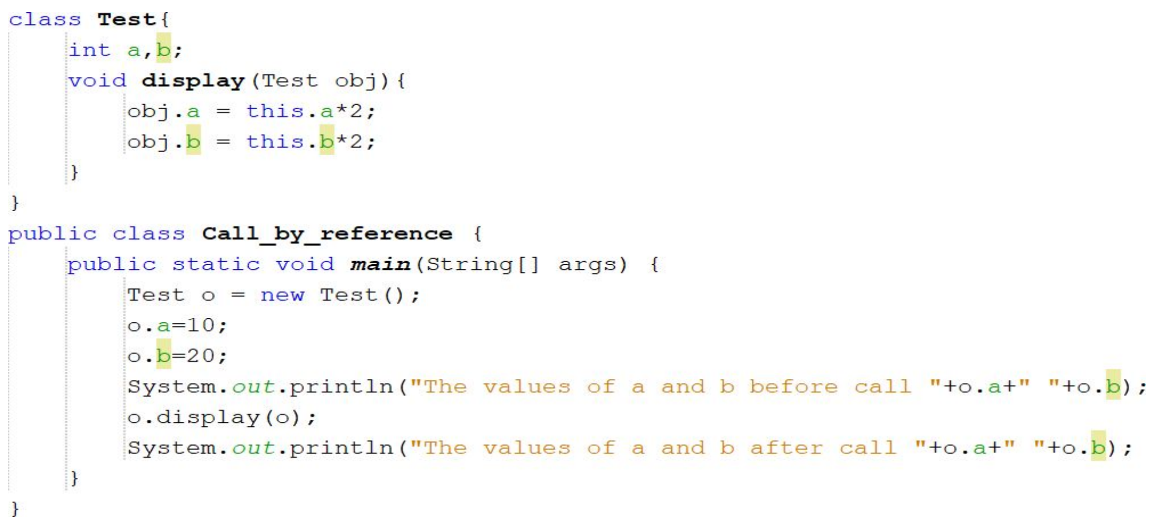
```
class Test{
    void display(int i, int j){
        i = i+j;
        j= -j;
    }
}
public class Call_by_value {
    public static void main(String[] args) {
        Test obj = new Test();
        int a=10, b=20;
        System.out.println("Values of a and b before call "+a+" "+b);
        obj.display(a,b);
        System.out.println("Values of a and b after call "+a+" "+b);
    }
}
```

Output:

```
Values of a and b before call 10 20
Values of a and b after call 10 20
```

- The operations that occur inside **display( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and -20.

Example for call-by-reference



```
class Test{
    int a,b;
    void display(Test obj){
        obj.a = this.a*2;
        obj.b = this.b*2;
    }
}
public class Call_by_reference {
    public static void main(String[] args) {
        Test o = new Test();
        o.a=10;
        o.b=20;
        System.out.println("The values of a and b before call "+o.a+" "+o.b);
        o.display(o);
        System.out.println("The values of a and b after call "+o.a+" "+o.b);
    }
}
```

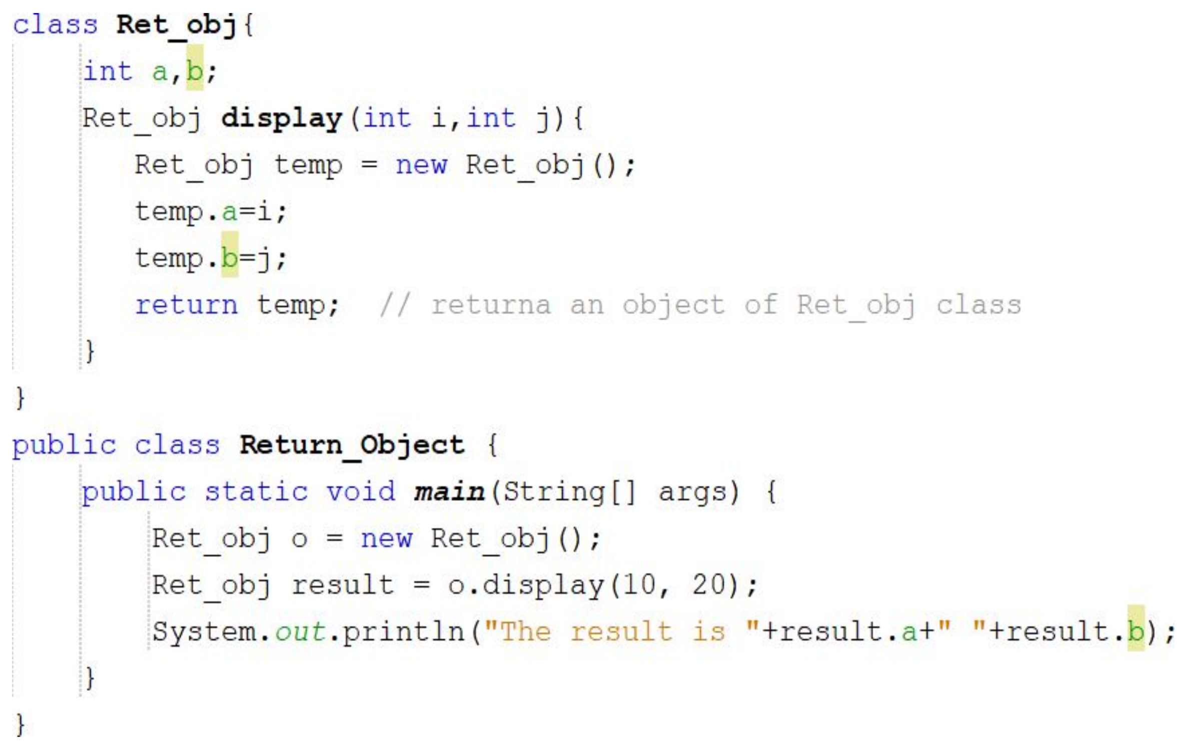
Output:

```
Values of a and b before call 10 20
Values of a and b after call 20 40
```

- In this case, the actions inside display( ) have affected and changed the actual object used as an argument, as copy of the reference (not the value) is passed as the argument. Hence, the reference variables o and obj are referring to the same object.

Returning Objects

- A method can return any type of data, including class types that you create.
- For example



```
class Ret_obj{
    int a,b;
    Ret_obj display(int i,int j){
        Ret_obj temp = new Ret_obj();
        temp.a=i;
        temp.b=j;
        return temp;  // returna an object of Ret_obj class
    }
}
public class Return_Object {
    public static void main(String[] args) {
        Ret_obj o = new Ret_obj();
        Ret_obj result = o.display(10, 20);
        System.out.println("The result is "+result.a+" "+result.b);
    }
}
```

Output:

When **display( )** method is invoked, it returns an object of the class Ret_obj class

Method Overloading

- Method overloading in Java allows a class to have multiple methods with the same name but different parameters. The compiler determines which method to call based on the number and types of arguments passed to the method.
- Method overloading is one of the ways that **Java supports polymorphism**
- Thus, overloaded methods must differ in the **type and/or number of their parameters**.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

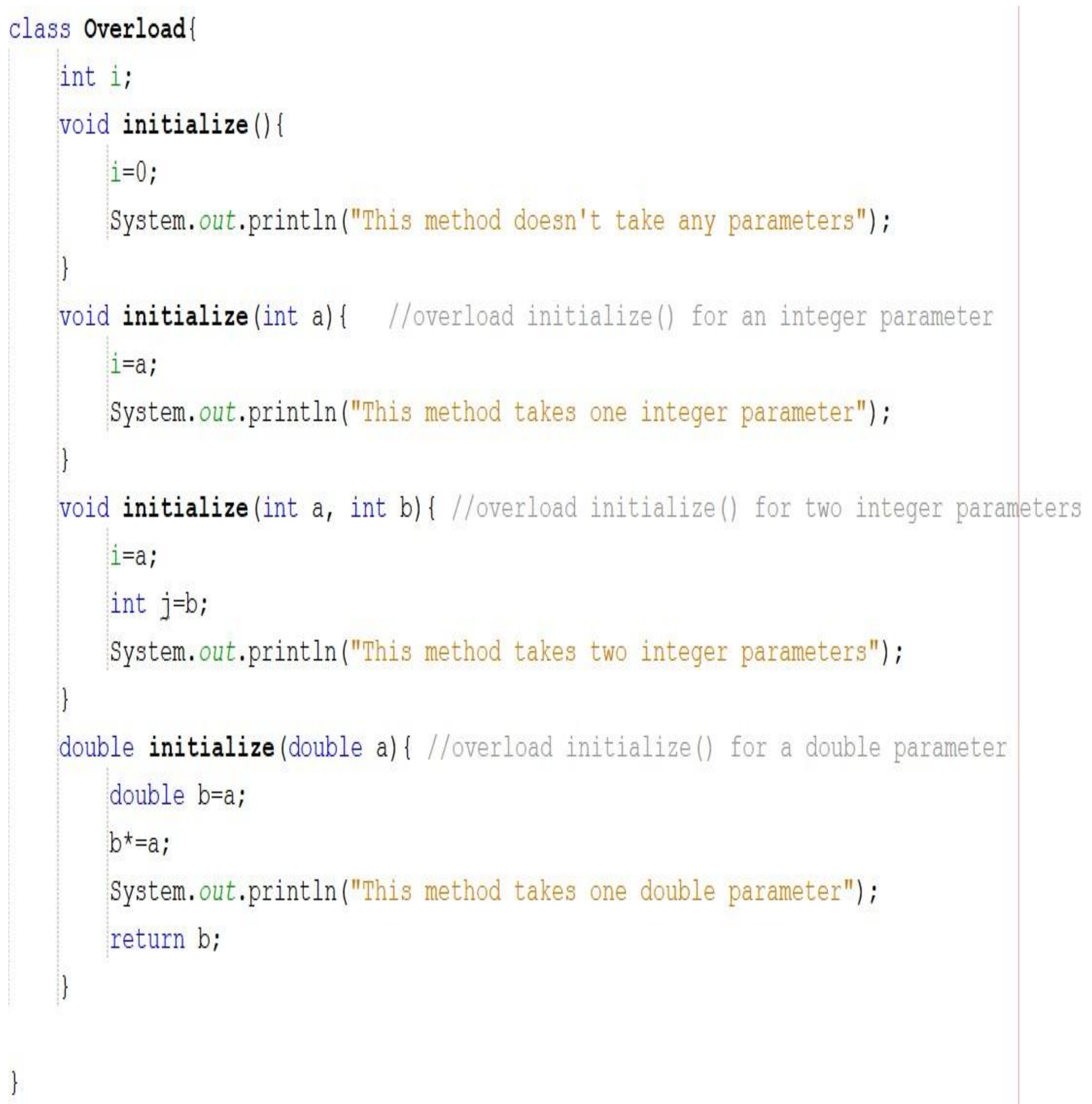
- Here's a simple example demonstrating method overloading using a **DrawShape** class:

```java
public class Calculator {

    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
}
```

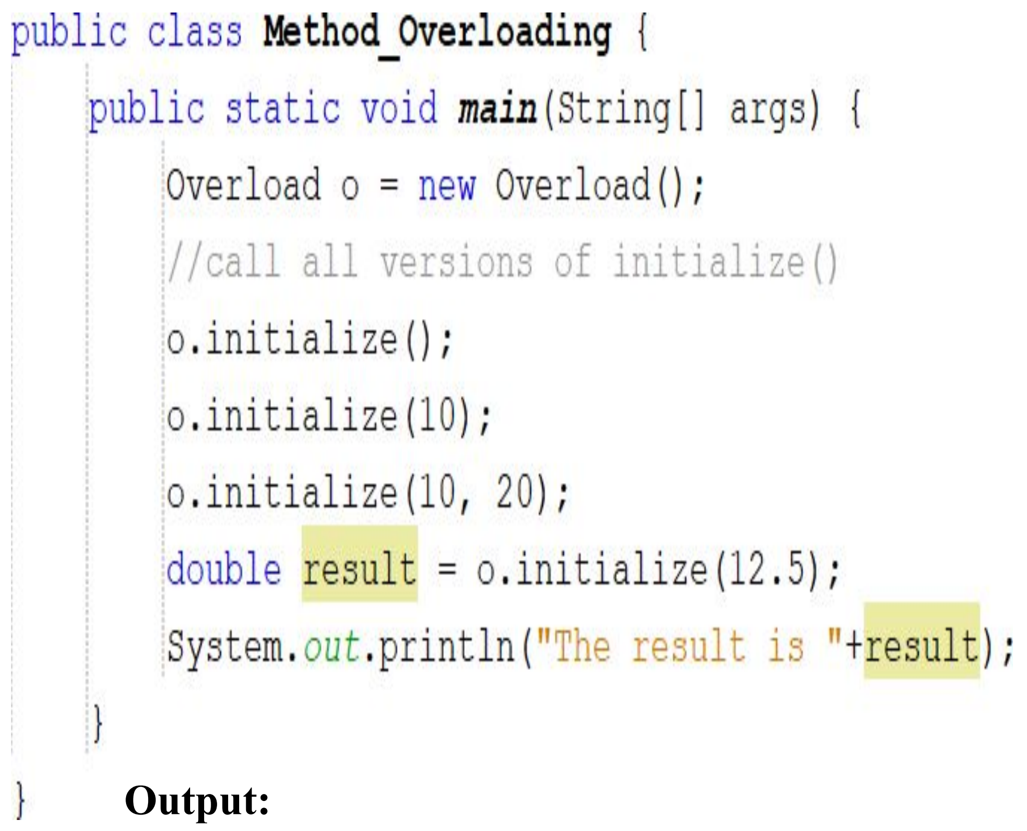
```java
public class Main{
 public static void main(String[] args) {
        Calculator c = new Calculator();

        // Testing addition methods
        System.out.println("Sum of 5 and 7: " + c.add(5, 7));
        System.out.println("Sum of 3, 8, and 10: " + c.add(3, 8, 10) );
        System.out.println("Sum of 2.5 and 3.5: " + c.add(2.5, 3.5) );

}
```

```
class DrawShape {

    // Method to draw a circle
    void draw(int radius) {
        System.out.println("Drawing a circle with radius: " + radius);
    }

    // Method to draw a rectangle
    void draw(int length, int width) {
        System.out.println("Drawing a rectangle with length " + length + " and width " + width);
    }

    // Method to draw a square
    void draw(int sideLength, boolean isSquare) {
        if (isSquare) {
            System.out.println("Drawing a square with side length: " + sideLength);
        } else {
            System.out.println("Drawing a rectangle with length " + sideLength + " and width " + sideLength);
            // Logic to draw a rectangle (for non-square)
        }
    }
}
```

```
public class MethodOverloadingExample {

    public static void main(String[] args) {
        DrawShape drawer = new DrawShape();

        drawer.draw(5);                    // Draw a circle
        drawer.draw(8, 6);                   // Draw a rectangle
        drawer.draw(4, true);                  // Draw a square
    }
}
```



```java
class Overload{
    int i;
    void initialize(){
        i=0;
        System.out.println("This method doesn't take any parameters");
    }
    void initialize(int a){   //overload initialize() for an integer parameter
        i=a;
        System.out.println("This method takes one integer parameter");
    }
    void initialize(int a, int b){ //overload initialize() for two integer parameters
        i=a;
        int j=b;
        System.out.println("This method takes two integer parameters");
    }
    double initialize(double a){ //overload initialize() for a double parameter
        double b=a;
        b*=a;
        System.out.println("This method takes one double parameter");
        return b;
    }

}
```



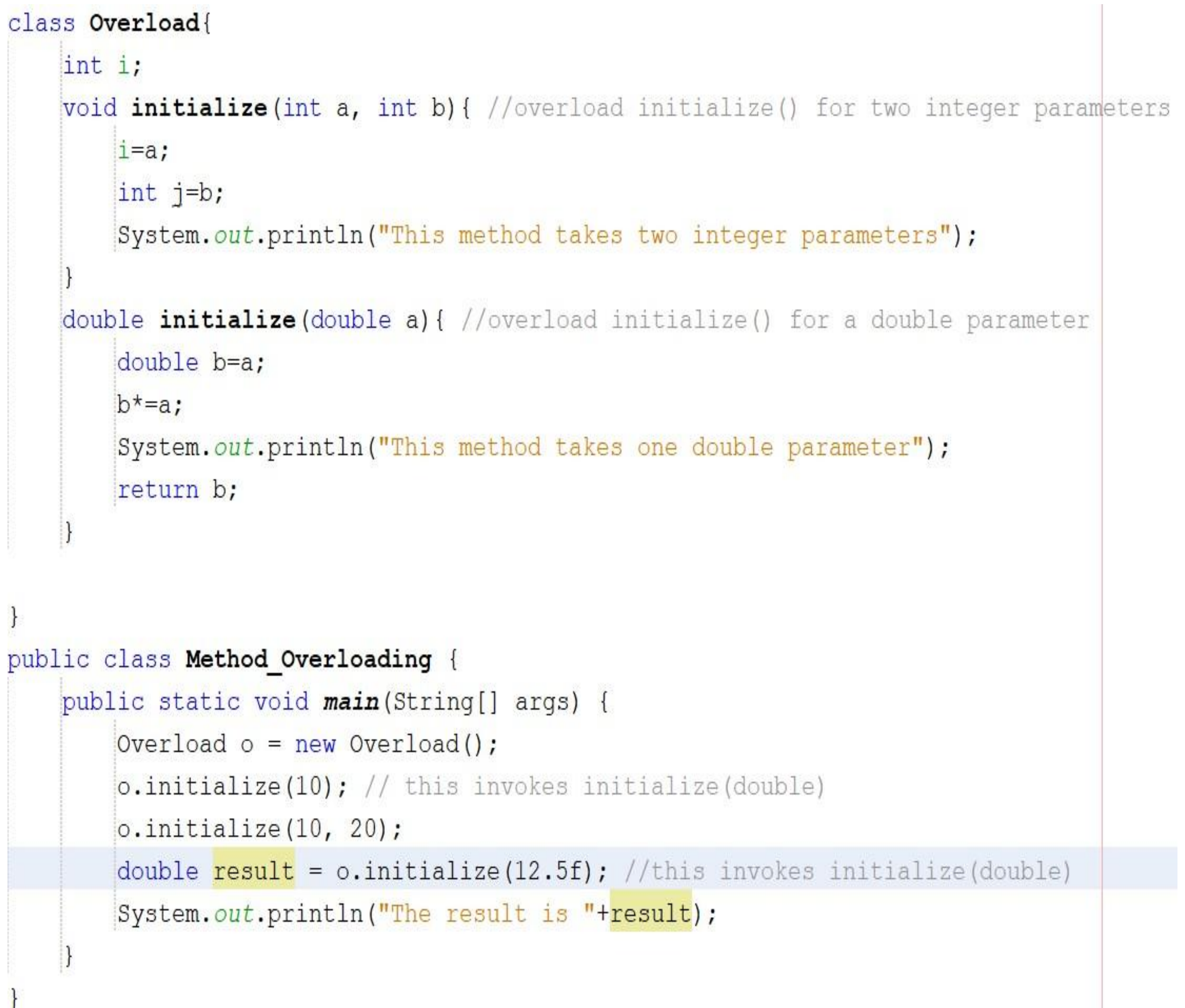
```java
public class Method_Overloading {
    public static void main(String[] args) {
        Overload o = new Overload();
        //call all versions of initialize()
        o.initialize();
        o.initialize(10);
        o.initialize(10, 20);
        double result = o.initialize(12.5);
        System.out.println("The result is "+result);
    }
}
```

Output:

```
This method doesn't take any parameters

This method takes one integer parameter

This method takes two integer parameters

This method takes one double parameter

The result is 156.25
```

- **initialize( )** is overloaded four times

- In some cases, Java’s automatic type conversions can play a role in overload resolution.



```
class Overload{
    int i;
    void initialize(int a, int b){ //overload initialize() for two integer parameters
        i=a;
        int j=b;
        System.out.println("This method takes two integer parameters");
    }
    double initialize(double a){ //overload initialize() for a double parameter
        double b=a;
        b*=a;
        System.out.println("This method takes one double parameter");
        return b;
    }

}
public class Method_Overloading {
    public static void main(String[] args) {
        Overload o = new Overload();
        o.initialize(10); // this invokes initialize(double)
        o.initialize(10, 20);
        double result = o.initialize(12.5f); //this invokes initialize(double)
        System.out.println("The result is "+result);
    }
}
```

Output:

```
This method takes one double parameter
This method takes two integer parameter
This method takes one double parameter
The result is 156.25
```

5a) Write a program to demonstrate the implementation of method overloading

5.1) Create a Stack class having an integer array say elem and top_of_stack as instance variables. Define three overloaded methods having the following signatures:

a. initStack(int size) to create an array of specified size and initialize the top_of_stack

b. initStack(Stack another) to intialize the Stack object with state of the Stack object "another"

c. initStack(int [] a) to initialize contents of a[] to the instance variable elem.

Write following methods:

a. push(): Pushes the element onto the stack,

b. pop(): Returns the element on the top of the stack, removing it in the process, and

c. peek(): Returns the element on the top of the stack, but does not remove it.

Also write methods that check whether stack is full and stack is empty and return boolean value true or false appropriately.

Stack Class:

1. **Instance Variables:**
   1. int[] ele: An array representing the elements of the stack.
   2. int top: An integer indicating the top of the stack.
2. **Methods:**
   1. void initStack(int size): Initializes the stack with a specified size.
   2. void initStack(Stack another): Initializes the stack using another Stack object. Copies the elements from the given stack to the current stack.
   3. void initStack(int[] a): Initializes the stack using an array. Copies the elements from the array to the stack.
   4. void push(int item): Pushes an element onto the stack if there is space.
   5. int pop(): Pops the top element from the stack and returns it.
   6. int peek(): Returns the top element of the stack without removing it.

```
class Stack{

    int[] ele;

    int top;

    void initStack(int size){

        ele=new int[size];

        top=-1;

    }

    void initStack(Stack another){

        ele=new int[another.ele.length];

        top=-1;

        for(int item:another.ele)

            push(item);

    }
```

```
void initStack(int[] a){
        ele=new int[a.length];
        top=-1;
        for(int item:a)
            push(item);
    }
```

```
void push(int item){
        if(top<ele.length){
            ele[++top]=item;
            System.out.println("Pushed element is "+item);
        }
        else
            System.out.println("Stack overflow");
    }
  int pop(){
        if(top==-1){
            System.out.println("Stack underflow");
            return -1;
        }
        else{
            int item=ele[top--];
            return item;
        }
    }
    int peek(){
        return ele[top];
    }
}
```

TW5a Class:

1. **Main Method:**
    1. Creates two Stack objects, s1 and s2.
    2. Initializes s1 with a size of 5 and pushes elements onto it.
    3. Initializes s2 using the initStack method that takes another Stack object (s1).

        This demonstrates the method overloading feature.
    1. Initializes a third Stack object, s3, using an array.
    2. Performs some stack operations:
        1. Pops an element from s1.
        2. Prints the element on top of s1.
        3. Prints the element on top of s2.

```
public class TW5a {
  public static void main(String[] args) {
     Stack s1=new Stack();
     Stack s2=new Stack();
     s1.initStack(5);
     s1.push(10);
     s1.push(20);
     s1.push(30);
     s1.push(40);
     s1.push(50);
    s2.initStack(s1);
    int[] array={1,2,3,4};
     Stack s3=new Stack();
     s3.initStack(array);
    System.out.println("Popped element in S1 object is "+s1.pop());
    System.out.println("Element on top of the stack of object s1 is "+s1.peek());
    System.out.println("Element on top of the stack of object s2 is "+s2.peek());

  }
}
```

1.Output Explanation:

1. The elements 10, 20, 30, 40, 50 are pushed onto s1.
2. s2 is initialized using the elements from s1, so s2 will have the same elements.
3. The top element of s1 is 40 after popping an element (50).
4. The top element of s1 is printed using the peek method.
5. The top element of s2 is also printed using the peek method.
6. Note: The code doesn't print the contents of the stack after each operation, but you can modify it to see the state of the stacks at different points in the program.

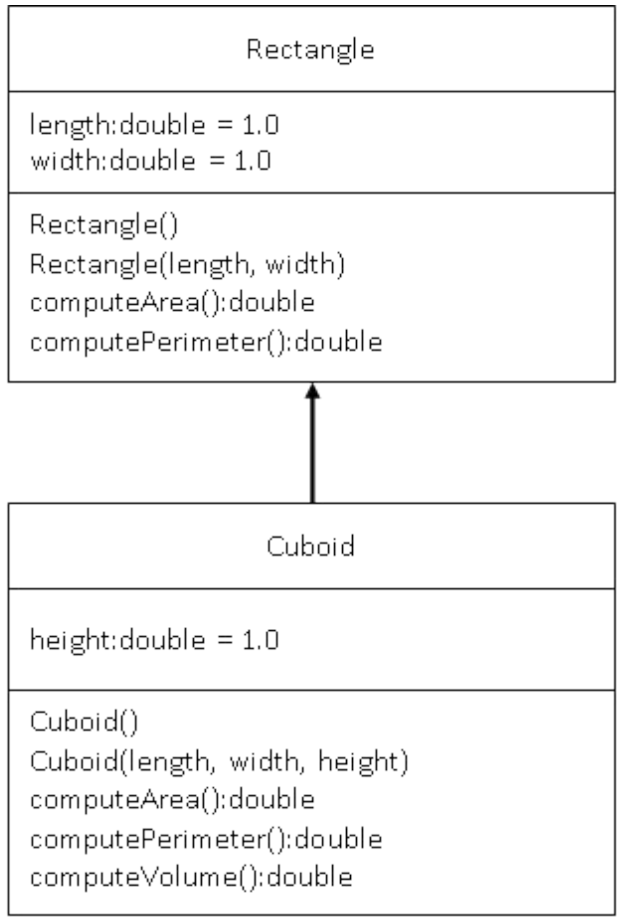
5 b) Overriding.

5 b.1)Implement the following class hierarchy. In the Cuboid class, override the method computeArea() and computePerimeter() of Rectangle class to compute the surface area and perimeter of a rectangle cuboid. Add a method computeVolume() in Cuboid class to compute volume of the cuboid. Assuming length, width and height as l, w and h respectively,

formula to find the surface area = 2(lw) + 2(hl) + 2(hw)

formula to find the perimeter = 2l + 2w

formula to find the volume = l x w x h


Rectangle
length:double = 1.0
width:double = 1.0
Rectangle()
Rectangle(length, width)
computeArea():double
computePerimeter():double
Cuboid
height:double = 1.0
Cuboid()
Cuboid(length, width, height)
computeArea():double
computePerimeter():double
computeVolume():double

```
class Rectangle {
    double length;
    double width;
    Rectangle() {
        length = 1.0;
        width = 1.0;
    }

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double computeArea() {
        return length * width;
    }

    double computePerimeter() {
        return 2 * (length + width);
    }
}
```

```
class Cuboid extends Rectangle {
    double height;
    Cuboid() {
        super();
        height = 1.0;
    }
    Cuboid(double length, double width, double height) {
        super(length, width);
        this.height = height;
    }
    @Override
    double computeArea() {
        return 2 * ((length * width) + (width * height) + (length *
height));
    }
    @Override
    double computePerimeter() {
        return 4 * (length + width + height);
    }
   double computeVolume() {
        return length * width * height;
    }
}
```

```
public class TW5b{
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        System.out.println("Rectangle 1:");
        System.out.println("Area:" + r1.computeArea());
        System.out.println("Perimeter:" + r1.computePerimeter());
        Rectangle r2 = new Rectangle(10,30);
        System.out.println("\nRectangle 2:");
        System.out.println("Area:" + r2.computeArea());
       System.out.println("Perimeter:" + r2.computePerimeter());
        Cuboid c1=new Cuboid();
        System.out.println("\nCuboid 1:");
        System.out.println("Area:" + c1.computeArea());
        System.out.println("Perimeter:" + c1.computePerimeter());
       System.out.println("Volume:" + c1.computeVolume());
       Cuboid c2=new Cuboid(10,30,40);
       System.out.println("\nCuboid 2:");
       System.out.println("Surface area:" + c2.computeArea());
       System.out.println("Perimeter:" + c2.computePerimeter());
       System.out.println("Volume:" + c2.computeVolume());
    }
}
```

Previous Year Question for reference..

Q1. Explain with examples the following terms.

a) this
b) finialize() method
c) returning objects.

Q2. Define constructors. Demonstrate the concept of constructor overloading with an example.

USN : ______________________ Course Code : 18CS/IS34 7

Third Semester B.E. Semester End Examination, FEBRUARY_APRIL_2022

OBJECT ORIENTED PROGRAMMING WITH JAVA

Time: 3 hrs. **Max. Marks :100**

Instructions: 1. Write appropriate comments wherever necessary.
2. Answer any FIVE Full Questions selecting at least ONE Question from Each Unit.

MODULE 1

L CO PO M

1a. Explain the three key attributes of object-oriented principles. Describe the general form of a java program. Write a java program to add 5 numbers.
[2] [1] [1] [10]

1b. Explain any five methods of String. Write a java program to display the numbers from 1 to 9 in the form of a table. The table has 3 rows and 3 columns.
[3] [2] [2] [10]

OR

2a. Define a class. Describe the general form of a class. Write a program to create a class called "Vehicle" and create two objects from the Vehicle class.
[3] [1] [2] [10]

2b. Describe for-each style for loop with an example. Write a Java program to perform addition of two matrices of size 3*3.
[3] [1] [2] [10]

MODULE 2

3a. Design a class named Sorting Class that has three overloaded methods to sort data of int, float and String types. Demonstrate its working by passing array of all the above three types. Use any sorting method.
[3] [2] [3] [6]

3b. Design a Complex class having realPart and imagPart as instance variables. Include the following in the class:
a. Default constructor
b. Parameterized constructor to initialize the instance variables
c. A method that adds two complex and returns the resulting complex object.
[3] [2] [3] [6]

3c. With suitable examples, explain the different ways of controlling access to members of a class.
[2] [1] [1] [8]

OR

4a. Design a class named Search Class that has three overloaded methods to search for a key element in an array of int, float and String types. Demonstrate its working by passing array and key to these three methods. Use any searching method.
[3] [2] [3] [6]

4b. Design a Distance class having feet and inches as instance variables. Include the following in the class:
a. Default constructor
b. Parameterized constructor to initialize the instance variables
c. A method that adds two Distance objects and returns the resulting Distance object.
[3] [2] [3] [8]

4c. With suitable examples, explain the different uses of the static keyword.
[2] [2] [1] [6]

MODULE 3

5a. Define Inheritance. Write a java program that creates a superclass called Two D Shape which stores the width and height of a two-dimensional object. It also has a subclass triangle which has a dimension as style that describes the triangle such as filled, outlined and transparent. It also computes area of triangle and displays triangle style.
[3] [3] [2] [10]

5b. Discuss Interfaces. Write a java program that specifies an interface called Series that describes the methods used to generate a series of numbers. Series defines three methods. A method to obtain next number in the series, second method to reset the series to its starting point and last method to set the starting point
[3] [3] [3] [10]

UNIT - I

Explain the three key attributes of object oriented principles. Describe the general form of a java program. Write a java program to convert 10 gallons to liters and display the result. There are approximately 3.7854 liters in a gallon.

(2) (1) (1) (10)

Explain methods of String class that operate on strings. Write a java program to display twelve names, stored in a one-dimensional array, in a tabular form that has 3 rows and 4 columns.

(2) (2) (3) (10)

OR

Describe the creation of a two dimensional array. Write a java program to sort elements of an array num[ ]={99, -10, 100123, 18, -978, 5623, 463, -9, 287, 49} using Bubble sort and print the sorted array.

(2) (1) (3) (10)

Describe for-each style for-loop with an example. Write a Java program to perform multiplication of two matrices.

(2) (2) (1) (10)

UNIT – II L CO PO M

Illustrate the use of java's access modifiers private and public. Define a class Myclass, alpha is a private data member and beta is public data member in Myclass. It includes accessor methods to get and set member alpha. The program initializes both the members and display messages appropriately.

(2) (2) (1) (08)

Write a java program that illustrates passing of objects to methods, it defines a class called Cuboid that stores the three dimensions namely width, breadth and height. Write a method, computeVolume that accepts object of this class and returns the volume. In the DemoClass, instantiate object of this class, invoke the method and display the result.

(3) (2) (3) (06)

Demonstrate method overloading in a FunClass, in which a method fun( ) is overloaded four times. First version takes no parameters, the second takes one integer parameter, the third takes two integer parameters and fourth takes two double parameters. Each method prints appropriate messages.

(3) (2) (1) (06)

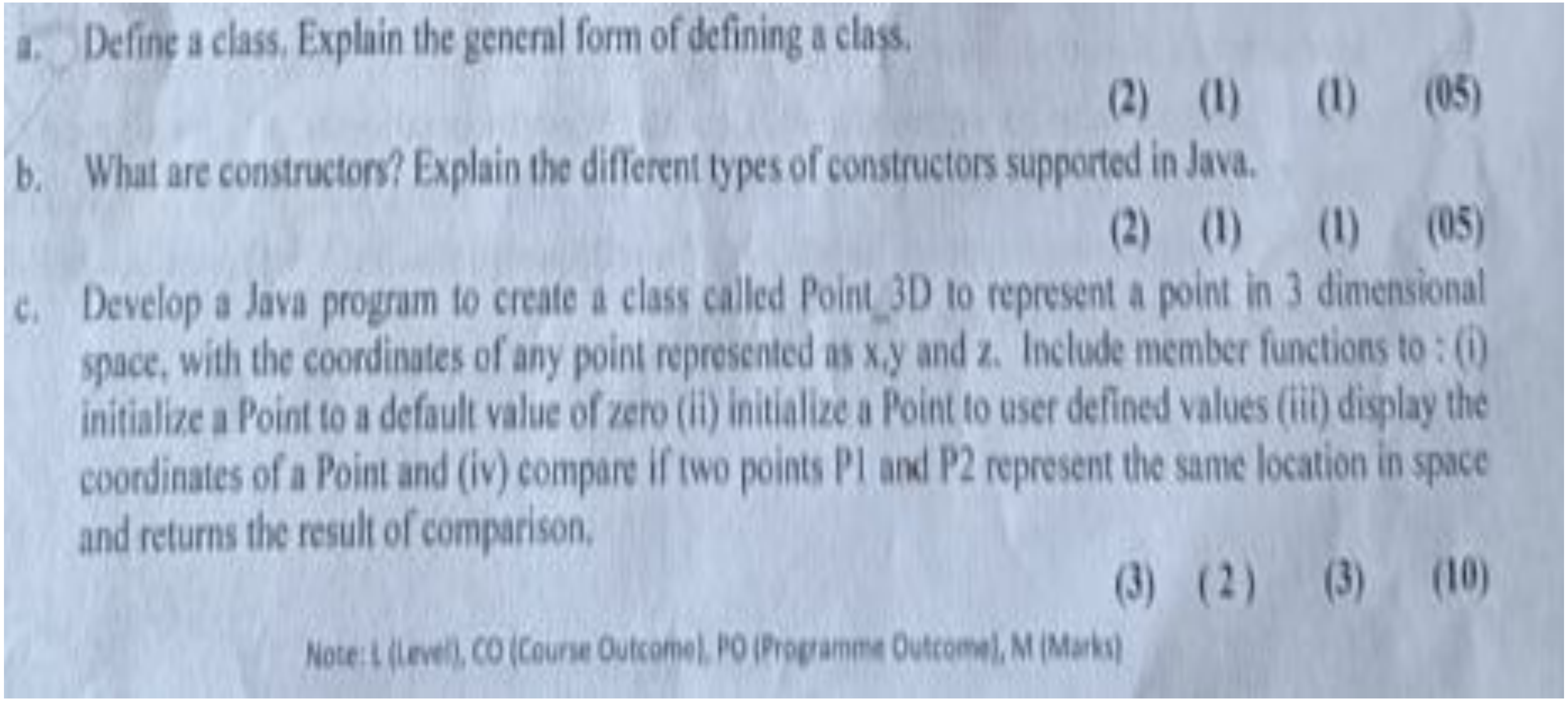
OR

a. Illustrate a class that encapsulates information about a vehicle, it stores 3 attributes about a vehicle namely, the number of passengers, its fuel capacity and its average fuel consumption in miles per gallon(mpg). Use parameterized constructor to initialize 2 vehicles. Add a method computeRange that returns distance covered by the vehicle. Print the values of two vehicles and also the distance travelled.

(2) (2) (1) (10)

b. Describe the use of this keyword. Write a java program that defines a Class Power that stores, base and exponent of the term a^n. Add a parameterized constructor that initializes a and n using this pointer. Also add a method computePower that iteratively computes a^n.

(3) (2) (3) (10)



a. Define a class. Explain the general form of defining a class.

(2) (1) (1) (05)

b. What are constructors? Explain the different types of constructors supported in Java.

(2) (1) (1) (05)

c. Develop a Java program to create a class called Point_3D to represent a point in 3 dimensional space, with the coordinates of any point represented as x,y and z. Include member functions to : (i) initialize a Point to a default value of zero (ii) initialize a Point to user defined values (iii) display the coordinates of a Point and (iv) compare if two points P1 and P2 represent the same location in space and returns the result of comparison.

(3) (2) (3) (10)

Note: L (Level), CO (Course Outcome), PO (Programme Outcome), M (Marks)

Design a class complex which has real part and imaginary part as instance variables. include the following in the class.

i) Default constructor.

ii)Parameterized constructor to initialize the instance variables

iii)A method that add two complex and return a resulting complex object.

```
class Complex {

   // Instance variables
    double real;
    double imaginary;

   // Default constructor
    Complex() {
      // Initializes to 0 + 0i
      this.real = 0.0;
      this.imaginary = 0.0;
   }
   // Parameterized constructor
    Complex(double real, double imaginary) {
      this.real = real;
      this.imaginary = imaginary;
   }
// Method to add two complex numbers
    Complex add(Complex other) {
      double newReal = this.real + other.real;
      double newImaginary = this.imaginary +
other.imaginary;
      return new Complex(newReal, newImaginary);
   }

    // Method to display the complex number
    public void display() {
       System.out.println(this.real + " + " + this.imaginary + "i");
    }
}
class Main{
    public static void main(String[] args) {
       // Example usage
       Complex c1 = new Complex(2.0, 3.0);
       Complex c2 = new Complex(1.5, -2.5);

       // Add two complex numbers
       Complex sum = c1.add(c2);

       System.out.println("Complex1: ");
       c1.display();

       System.out.println("Complex2: ");
       c2.display();

       System.out.println("Sum: ");
       sum.display();
    }
}
```

Design a Distance class having feet and inches as instance variables. include the following in the class.

i) Default constructor.

ii) Parameterized constructor to initialize the instance variables

iii)A method that add two distance object and returns a resulting distance object

```
class Distance {

  // Instance variables
   int feet;
   int inches;

  // Default constructor
   Distance() {
     // Initializes to 0 feet and 0 inches
     this.feet = 0;
     this.inches = 0;
  }

  // Parameterized constructor
   Distance(int feet, int inches) {
     this.feet = feet;
     this.inches = inches;
  }

// Method to add two Distance objects and return the result
    public Distance add(Distance other) {
        int newFeet = this.feet + other.feet;
        int newInches = this.inches + other.inches;

// Adjust if inches exceed 12
        if (newInches >= 12) {
            newFeet += newInches / 12;
            newInches %= 12;
        }

        return new Distance(newFeet, newInches);
    }

    // Display method
    public void display() {
        System.out.println(feet + " feet, " + inches + " inches");
    }
}
```

```
class Main{
    public static void main(String[] args) {
        // Example usage
        Distance d1 = new Distance(3, 9);
        Distance d2 = new Distance(2, 6);

        // Add two distances
        Distance sum = d1.add(d2);

        // Display the results
        System.out.println("Distance1: ");
        d1.display();

        System.out.println("Distance2: ");
        d2.display();

        System.out.println("Sum: ");
        sum.display();
    }
}
```

Q3. With example explain the different uses of static keyword in java

Ans: In Java, the static keyword is used in various contexts, such as with variables, methods, and nested classes.

1. Static Variables:

Static variables belong to the class rather than to instances of the class. There is only one copy of the static variable, shared among all instances of the class.

```
public class ExampleClass {
   // Static variable shared among all instances of ExampleClass
   static int staticVariable = 0;

   public static void main(String[] args) {
      ExampleClass obj1 = new ExampleClass();
      ExampleClass obj2 = new ExampleClass();

      obj1.staticVariable = 10;

      System.out.println("obj1.staticVariable: " + obj1.staticVariable); // Output: 10
      System.out.println("obj2.staticVariable: " + obj2.staticVariable); // Output: 10
```

2. Static Method: Static methods belong to the class rather than to instances of the class. They can be called using the class name without creating an instance of the class.

```
public class ExampleClass {
    // Static method
    static void staticMethod() {
        System.out.println("This is a static method.");
    }

    public static void main(String[] args) {
        ExampleClass.staticMethod(); // Output: This is a static method.
    }
}
```

3. Static Block: A static block is used to initialize static variables. It runs once when the class is loaded into the memory.

```
public class ExampleClass {
    // Static variable
    static int staticVariable;

    // Static block
    static {
        System.out.println("Static block is executed.");
        staticVariable = 5;
    }

    public static void main(String[] args) {
        System.out.println("staticVariable: " + staticVariable); // Output: 5
    }
}
```