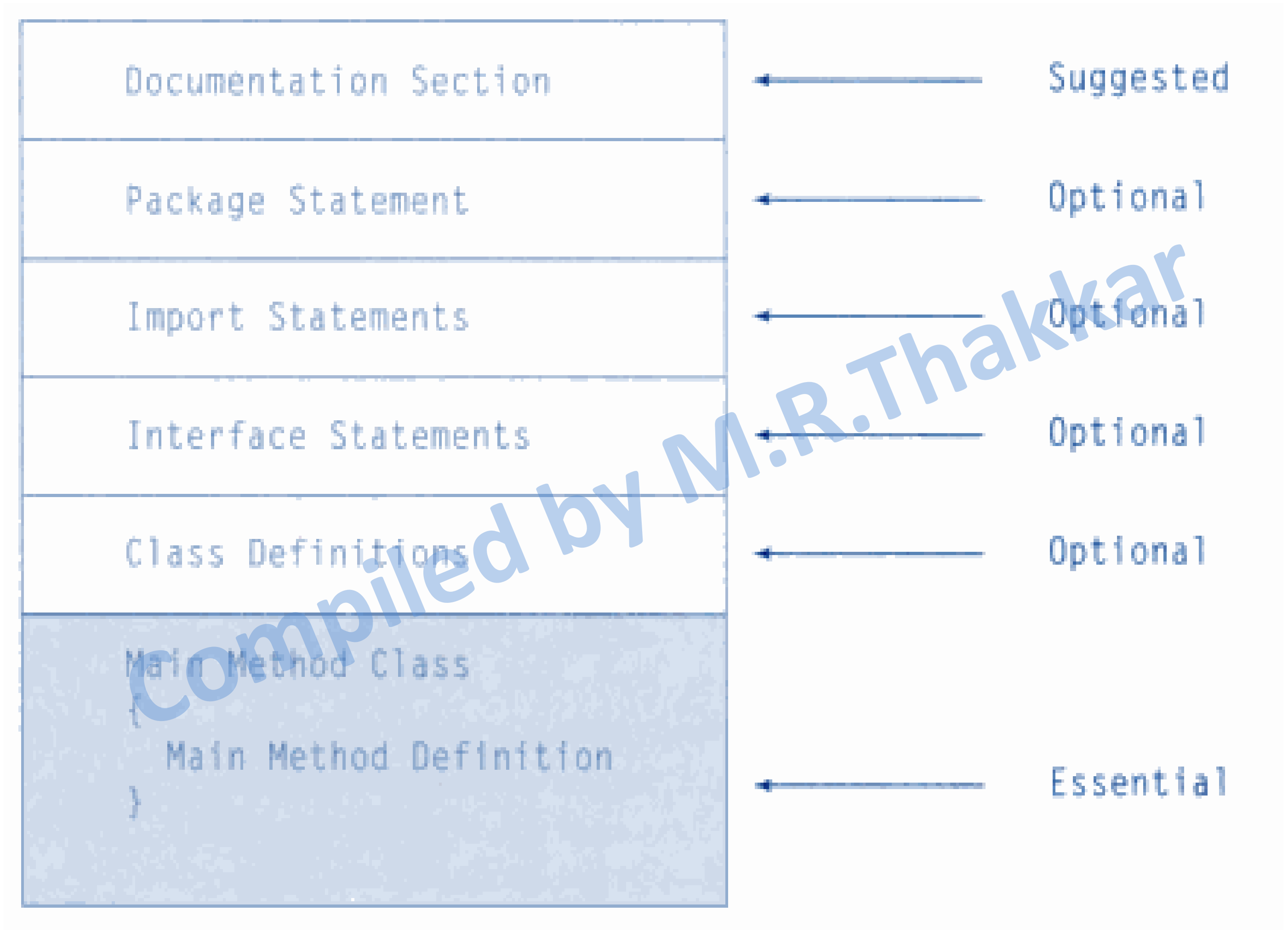
UNIT – III

OBJECT ORIENTED PROGRAMMING CONCEPTS

Java Program Structure



3.1.1 Defining Classes, Fields and Methods

- **Defining Class**
- A class is defined by use of the class keyword.

```
class  <classname>
{
   <body of the class>
}
```

3.1.1 Defining Classes, Fields and Methods

- **Defining Class**
- The variables and methods can be part of the class body.
- The variables, defined within a class are called *instance variables. The code* is contained within *methods.*
- Collectively, the methods and variables defined within a class are called members of the class.

3.1.1 Defining Classes, Fields and Methods

- **Defining Class**

```
class classname
{
      type instance-variable1;
      type instance-variable2;
      // ...
      type instance-variableN;

      type methodname1(parameter-list)
      {
                        // body of method
      }
      type methodname2(parameter-list)
      {
                        // body of method
      }
         // ...
      type methodnameN(parameter-list)
      {
                        // body of method
      }
}
```

3.1.1 Defining Classes, Fields and Methods

- **Defining Class**
- **Example:**

```
class   Test
{
     int   i;
     void  display()
     {
        System.out.print(“ i = “ + i);
     }
}
```

3.1.1 Defining Classes, Fields and Methods

- **Defining Class**

- It is important to remember that a class declaration only creates a template; it does not create an actual object.

- when you create a class, you are creating a new data type. You can use this type to declare objects of that type.

3.1.2 Creating Objects

- Creating objects of a class is a two-step process:

1. You need to declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer to an object.*

Examlple:

```
Test  obj;
```

2. You need to create an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator.

Examlple:

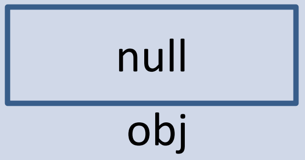
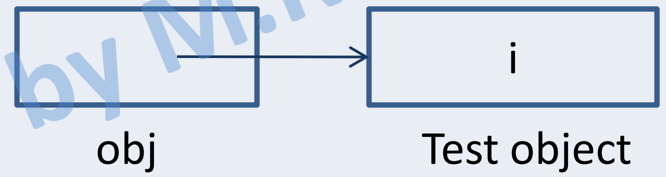
```
obj  = new Test();
```

3.1.2 Creating Objects

- The new operator dynamically (run time) allocates memory for an object and returns a reference to it.
- This reference is then stored in the variable.
- These two statements can be combined in single line :

```
Test  obj = new Test();
```

3.1.2 Creating Objects

Statement	Effect
Test obj;	
Obj= **new Test();**	

Exercise

1. Write a program in JAVA that implements a class with one instance variable and one method as class member.

2. Write a program in JAVA that implements the class **Box** with instance variables length, width, height and calculate the volume of the Box using class method volume.

3. Write a program in JAVA that implements the class **Box** with instance variable length, width, height and method **setdata** to assign the value to these attributes and also calculate the volume of the Box using class method **volume**.

Exercise (1)

```
class   Test
{
     int   i;

     void  display()
     {
        System.out.print(" i = " + i);
     }
}
```

Exercise (1)

```
public class Example
 {
      public static void main(String[] args)
     {
             Test  obj1 = new Test();
             obj1.i = 5;
             obj1.display();
     }
}
```

Output: i = 5

Exercise (2)

```
class Box
{

    int length;
    int width;
    int height;


    public void volume()
    {
        int vol = length * width * height;
        System.out.println("Volume of the Box: " + vol );
    }
}
```

Exercise (2)

```
public class Example
 {
      public static void main(String[] args)
     {
              Box b1 = new Box();

              b1.length=2;
              b1.width = 3;
              b1.height = 5;

              b1.volume();
   }
}
```

Output: Volume of the Box: 30

Exercise (3)

```
class Box
{
    int length;
    int width;
    int height;

    void setdata(int l, int w, int h)
    {
        length=l;
        width = w;
        height =h;
    }

    public void volume()
    {
        int vol = length * width * height;
        System.out.println("Volume of the Box: " + vol );
    }
}
```

Exercise (3)

```
public class Example
{
    public static void main(String[] args)
     {
            Box b1 = new Box();

            b1.setdata(2,3,5);
            b1.volume();
    }
}
```

Output: Volume of the Box: 30

3.1.3 Accessing Rules

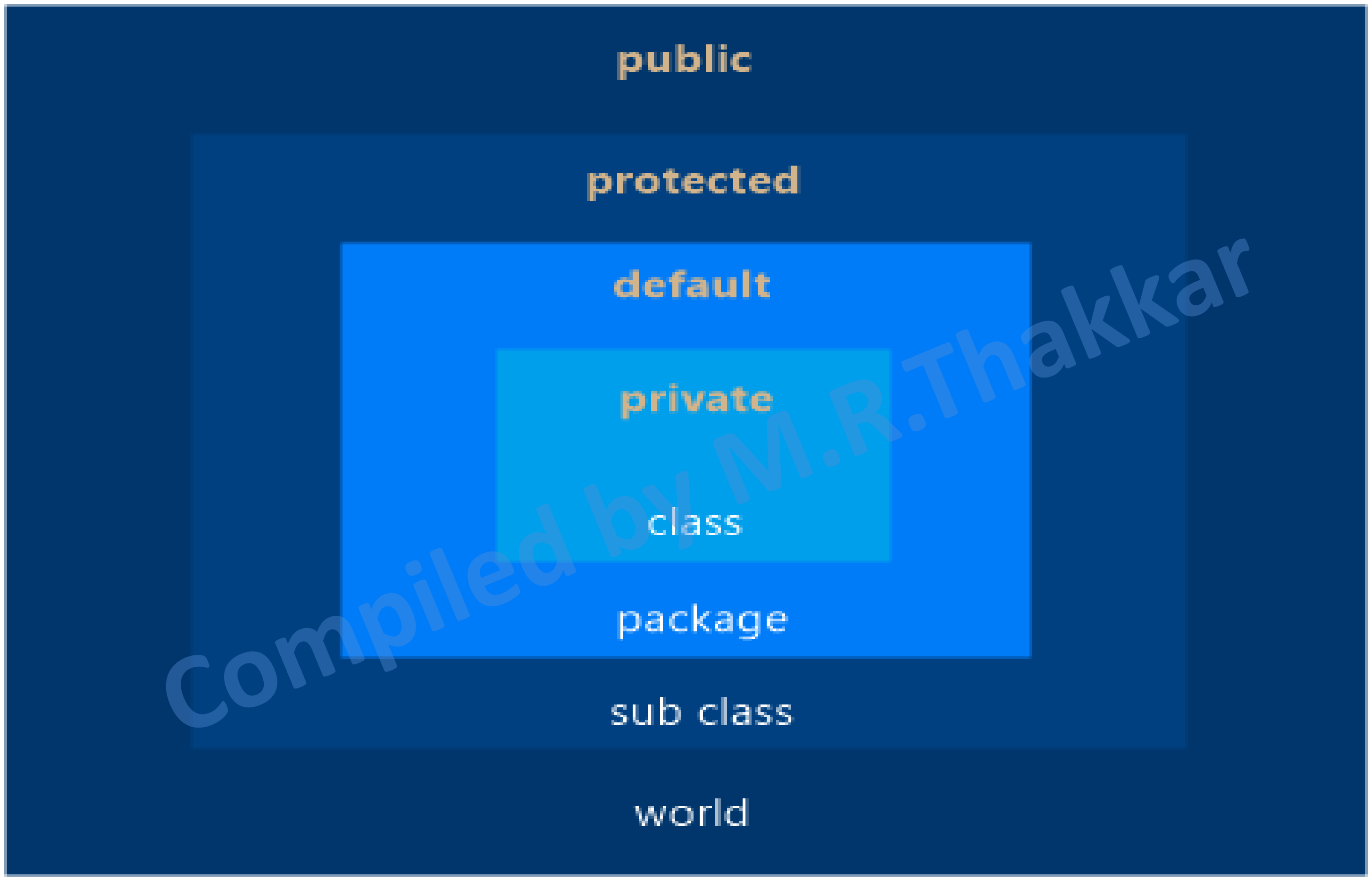
- Access control is the process of controlling visibility of a variable or method.

- There are four levels of visibility :

1. **Public: -** public members can be accessed by any other code from anywhere.
2. **Private: -** Private members can only be accessed by other members of its class.
3. **Protected: -** protected members can only be accessed by classes that are subclass of the class directly. **protected applies only when inheritance is involved.**

- **Default (or Package): -** When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.

3.1.3 Accessing Rules



3.1.3 Accessing Rules

	Private	Package	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package – Sub Class	No	Yes	Yes	Yes
Same Package - Different Class	No	Yes	Yes	Yes
Different Package – Sub Class	No	No	Yes	Yes
Different Package – Different Class	No	No	No	Yes

3.1.3 Accessing Rules

Example:

```
class Protection
{

    public  int  n_pub = 1;
    protected  int  n_pro = 2;
               int   n = 3;
    private  int  n_pri = 4;

}
```

3.1.3 Accessing Rules

```
public class Example
{
    public static void main(String[] args)
    {
        Protection p1 = new Protection();

        System.out.println("n_pub = " +p1.n_pub);
        System.out.println("n_pro = " + p1.n_pro);
         System.out.println("n = " + p1.n);
         System.out.println("n_pri = " + p1.n_pri);
        // error: n_pri has private access in Protection
     }
}
```

3.1.3 Accessing Rules

```
public class Example
{
        public static void main(String[] args)
        {
                 Protection p1 = new Protection();

                  System.out.println("n_pub = " +p1.n_pub);
                  System.out.println("n_pro = " + p1.n_pro);
                  System.out.println("n = " + p1.n);
             // System.out.println("n_pri = " + p1.n_pri);

         }
}
```

Output:

```
n_pub = 1
n_pro = 2
n = 3
```

3.1.4 *this* keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines 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.

3.1.4 *this* keyword

```
class Box
{
    int length;
    int width;
    int height;

    void setdata(int l, int w, int h)
    {
        this.length = l;    // similar to length = l
        this.width = w;  // similar to width = w
        this.height = h;  // similar to height = h
    }

    public void volume()
    {
        int vol = this.length * this.width * this.height;
        System.out.println("Volume of the Box: " + vol );
    }
}
```

3.1.4 *this* keyword

```
public class Example
{
    public static void main(String[] args)
     {
           Box b1 = new Box();

           b1.setdata(2,3,5);
           b1.volume();
    }
}
```

Output: Volume of the Box: 30

3.1.4 *this* keyword

- **Instance Variable Hiding**

- when a local variable has the same name as an instance variable, the local variable *hides the instance variable.*

```
class  Test
{
   int   n;
      void setdata(int  n)
      {
                     n   =   n;
      }

      void getdata()
      {
         System.out.println("n=" + n);
      }
}
```

3.1.4 *this* keyword

```
public class Example
{
    public static void main(String[] args)
    {
        Test obj = new Test();
        obj.setdata(5);
        obj.getdata();
    }
}
```

<u>Output:</u> n=0

3.1.4 *this* keyword

- **Instance Variable Hiding**

- when a local variable has the same name as an instance variable, the local variable *hides the instance variable.*

```
class  Test
{
   int   n;
      void setdata(int  n)
      {
                        n   =   n;
      }

      void getdata()
      {
         System.out.println("n=" + n);
      }
}
```

3.1.4 *this* keyword

- **Instance Variable Hiding**
- when a local variable has the same name as an instance variable, the local variable *hides the instance variable.*

```
class  Test
{
  int   n;
     void setdata(int  n)
     {
                  this.n   =   n;
     }

     void getdata()
     {
        System.out.println("n=" + n);
     }
}
```

3.1.4 *this* keyword

```
public class Example
{
    public static void main(String[] args)
    {
        Test obj = new Test();
        obj.setdata(5);
        obj.getdata();
    }
}
```

Output: n=5

3.1.5 *static* keyword

- When a class member is declared as **static, it can be accessed before any objects of its class are created, and without** reference to any object.
- You can declare both **methods** and **variables** to be **static**.
- **Outside of the class** in which they are defined, **static methods and variables can be** used independently of any object.
- To do so, you need only specify the name of their class followed by the dot operator :
  - *classname.method( )*
  - *classname.variable*

3.1.5 *static* keyword

- **Static Variable:**
- Instance variables declared as static are, essentially, global variables. When objects of its class are declared, **no copy of a static variable is created.**
- Instead, **all instances of the class share the same static variable.**
- **Static Method:**
- Methods declared as static have several restrictions:
  - They can only call other **static methods.**
  - They must only access **static data.**
  - They cannot refer to **this** or **super.**

3.1.5 *static* keyword

- **Example:**

```
class StaticDemo
{
    static int a = 42;

    static void displayA()
    {
        System.out.println("a = " + a);
    }
}
```

3.1.5 *static* keyword

- **Example (Continue):**

```
public class Example
{
    public static void main(String[] args)
    {
            StaticDemo.displayA();
    }
}
```

Output:

```
a = 42
b = 99
```

3.1.5 *static* keyword

- **Static Block:**
- **Static block** is a set of statements, which will be executed by the JVM **before execution of main method**.
- At the time of class loading if we want to perform any activity we have to define that activity inside static block because static block execute at the time of class loading.

3.1.5 *static* keyword

- **Example:**

```
class StaticDemo
{
        static int a ;

        static void display()
        {
           System.out.println("a = " + a);
        }

        static
        {
                a=42;
        }
}
```

3.1.5 *static* keyword

- **Example (Continue):**

```
public class Example
{
    public static void main(String[] args)
    {
                StaticDemo.display();
    }
}
```

Output:

```
a = 42
```

3.1.6 *final* keyword

- In java language final keyword can be used in following way:
  - Final at variable level
  - Final at method level
  - Final at class level

3.1.6 *final* keyword

Final at variable level

- Final keyword is used to make a variable as a constant.
- A variable declared with the final keyword cannot be modified by the program after initialization.
- This means that you must initialize a **final variable when it is declared.**
- **Example:**

```
final double PI=3.14159;
```

3.1.6 *final* keyword

Final at method level

- It makes a method final, meaning that sub classes can not override this method.
- The compiler checks and gives an error if you try to override the method.

- **Example:**

```
final void display()
{
......
......
}
```

3.1.6 *final* keyword

Final at class level :

- It makes a class final, meaning that the class can not be inherited by other classes.
- When we want to restrict inheritance then make class as a final.
- **Example:**

```
final class  Example
{
......
......
}
```

3.1.7 Method Overloading

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.

- When this is the case, the methods are said to be *overloaded, and the process is referred to as method overloading.*

- When an overloaded method is invoked, Java uses the **type of arguments or number of arguments** as its guide to determine **which version of the overloaded method to actually call**.

3.1.7 Method Overloading

- **Example:**

```
class OverloadDemo
{
        void display()
        {
            System.out.println("No parameters");
        }

        void display(int a)
        {
            System.out.println("a: " + a);
        }

        void display(double a)
        {
            System.out.println("double a: " + a);
        }

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

3.1.7 Method Overloading

- **Example:**

```
public class Example
{
    public static void main(String[] args)
    {
        OverloadDemo ob = new OverloadDemo();

        ob.display();
        ob.display(10);
        ob.display(123.25);
        ob.display(10, 20);
    }
}
```

3.1.7 Method Overloading

- **Output:**

```
No parameters
a: 10
double a: 123.25
a and b: 10 20
```

3.2.1 Constructors

- A constructor is a special kind of method that **initializes an object** when created.
- A constructor has **the same name as the class** and **do not have any return type.**
- Constructor is **automatically called immediately after the object is created** using “new” key-word.
- There are three different types of Constructor:
  1. Default Constructor
  2. Parameterized Constructor
  3. Copy Constructor

3.2.2 Default Constructor

- This constructor **does not have any arguments**.
- If you don't write a constructor with arguments in a class **then Java provides a one constructor for you**, which is default constructor.
- Therefore, Every class has at least one constructor, which is called default constructor.
- We can create an object using default constructor as:

```
new class_name( );
```

3.2.2 Default Constructor

```
class Box
{
        int  length;
        int  width;
        int  height;

        Box()
        {
                System.out.println("Inside Default Constructor");
                length = 10;
                width = 10;
                height = 10;
        }

        void volume()
        {
                        int vol = length * width * height;
                        System.out.println("Volume of the Box: " + vol );
        }

}
```

3.2.2 Default Constructor

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

          Box b1= new Box();
          b1.volume();

           Box b2= new Box();
          b2.volume();

      }
}
```

3.2.3 Parameterized Constructor

- Since the Default constructor **gives all the instance variables same value**, it is not very useful.

- What is needed is a way to **add parameters to the constructor**. As you can probably guess, this makes them much more useful.

- The **constructors that can take arguments** are called parameterized constructor.

3.2.3 Parameterized Constructor

```
class Box
{
        int  length;
        int  width;
        int  height;


        Box(int l, int w,int h)
         {
                System.out.println("Inside Parameterized Constructor");
                length = l;
                width = w;
                height = h;
        }

        void volume()
        {
                        int vol = length * width * height;
                        System.out.println("Volume of the Box: " + vol );
        }

}
```

3.2.3 Parameterized Constructor

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

        Box b1= new Box(10,10,10);
        b1.volume();

        Box b2= new Box(20,20,20);
        b2.volume();


     }
}
```

3.2.4 Copy Constructor

- A copy constructor is used to **initialize an object from another object.**
- Copy constructor **takes object as argument**.

3.2.4 Copy Constructor

```
class Box
{
        int  length;
        int  width;
        int  height;

        Box()
        {
                System.out.println("Inside Default Constructor");
                length = 10;
                width = 10;
                height = 10;
        }
```

3.2.4 Copy Constructor

```
Box(Box b)
{
        System.out.println("Inside Copy Constructor");
        length = b.length;
        width = b.width;
        height = b.height;
}

void volume()
{
                int vol = length * width * height;
                System.out.println("Volume of the Box: " + vol );
}

}
```

3.2.4 Copy Constructor

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

           Box b1= new Box();
           b1.volume();



           Box b2= new Box(b1);
           b2.volume();

    }
}
```

3.2.5 Passing Object as Parameter

- We can also **pass object as a parameter to method** in Java programming.

```
class Test
{
        int a;

        Test(int i)
        {
            a = i;
        }

        // return true if t is equal to the invoking object
        boolean equals ( Test  t )
         {
            if (a == t.a)
                        return  true;
            else
                        return  false;
        }
}
```

3.2.5 Passing Object as Parameter

```
public class Example
{
    public static void main(String args[])
    {
        Test t1 = new Test(10);
        Test t2 = new Test(10);
        Test t3 = new Test(20);
        System.out.println(“t1 == t2: " + t1.equals(t2));
        System.out.println(“t1 == t3: " + t1.equals(t3));
    }
}
```

3.2.6 Constructor Overloading

- In addition to overloading normal methods, **you can also overload constructor methods.**

```
class Box
{
        int  length;
        int  width;
        int  height;
        Box()
        {
                System.out.println("Inside Default Constructor");
                length = 10;
                width = 10;
                height = 10;
        }

        Box(int l, int w,int h)
         {
                System.out.println("Inside Parameterized Constructor");
                length = l;
                width = w;
                height = h;
        }
```

3.2.6 Constructor Overloading

```
        Box(Box b)
        {
                System.out.println("Inside Copy Constructor");
                length = b.length;
                width = b.width;
                height = b.height;
        }

        void volume()
        {
                        int vol = length * width * height;
                        System.out.println("Volume of the Box: " + vol );
        }

    }
```

3.2.6 Constructor Overloading

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

          Box b1= new Box();
          b1.volume();

          Box b2= new Box(5,5,10);
          b2.volume();

          Box b3= new Box(b1);
          b3.volume();

    }
}
```

****