CHAPTER THREE

CLASSES AND OBJECTS

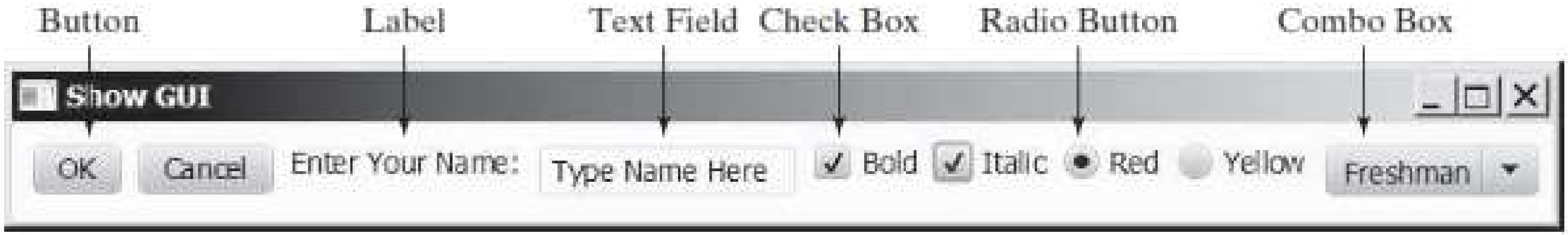
Introduction

- **Object-oriented programming** enables you to develop **large-scale software** and **GUIs** effectively.

- You are **solved many programming problems** using **selections**, **loops**, **methods**, and **arrays** in the previous chapters.

- However, these **Java** features are **not sufficient** for developing **graphical user interfaces** and **large-scale software systems**.

- Suppose you want to develop a **graphical user interface (GUI)** as shown as follows:



Class Fundamentals

- **Class** introduces a **new data type i.e.;** describes a **set of objects** that have identical **characteristics(data elements) and behaviors (methods).**
- **Class defines what** all **objects** of the **class represents.**
- It is a **template(model or blue print)** that represent what the **object looks like** or to **model** the **real world**.
- Shortly it defines the **shape** and **nature** of an **object.**

➢ A **class** may be:

  - **Existing classes provided by JRE**
  - **Classes** defined by the **Programmer**

- Once a **class** is **defined**, you can make as **many objects** of it as you like, or none.

Defining Classes for Objects

- ➢ A **class** contains **two elements** i.e.; are **properties** (**attributes**) and a **behavior** (**methods**).
- ▪ Therefore a **class** defines the **properties** and **behaviors** for **objects**.
- ➢ **Object-oriented programming (OOP)** involves programming using **objects**.
- ▪ An **object** represents an **entity** in the **real world** that can be **distinctly identified**.
- ✓ For **example**, a **student**, a **desk**, a **circle**, a **button**, and even a **loan** can all be viewed as **objects**.

Defining Classes for Objects-----

- An **object** is an **instance** of a **class.**
- A class is a **template** or **blueprint** from which **objects** are **created**.
- So, an **object** is the **instance(result)** of a **class.**
- An object is a **real-world entity**.
- An object is a **runtime entity**.
- The object is an **entity** which has **state** and **behavior**.

Characteristics of an Object

- An object has a **unique identity**, **state**, and **behavior**.

1. Identity

➢ An **identity** is a **unique address** in **memory—how** is **one object distinguished** from others that may have the **same behavior** and **state.**

- An **object identity** is typically implemented via a **unique ID**.
- The **value** of the **ID** is **not visible** to the **external user.**
- However, it is used **internally** by the **JVM** to **identify each object uniquely**.
- **For example**, in an **order-processing system**, **two orders** are **distinct** even if they request **identical items**.

Characteristics of an Object-------

2. State

- It is also known as **properties** or **attributes** which is represented by **data fields** with their **current values**.

➢ **For example:**

- A **circle object** has a **data field** **radius**, which is the **property** that **characterizes** a **circle**.
- A **rectangle object** has the **data fields** **width** and **height**, which are the **properties** that characterize a rectangle.

Characteristics of an Object-------

3. Behavior

- The **behavior** of an **object** (also known as its **actions**) is defined by **methods**.
- It is a **blocks** of **code** that typically **operate** on the **fields** and **perform** a **specific operation** when it is **called.**

✓ It describe **what** an **object** can do.

- To **invoke a method** on an **object** is to **ask** the **object** to **perform** an **action**.

➢ For **example**, you may define **methods** named **getArea()** and **getPerimeter()** for **circle objects**.

Characteristics of an Object-------

- A **circle** object may invoke **getArea()** to return its **area** and **getPerimeter()** to return its **perimeter**.
- You may also define the **setRadius(radius)** **method**.
- A **circle object** can **invoke** this method to change its **radius**.

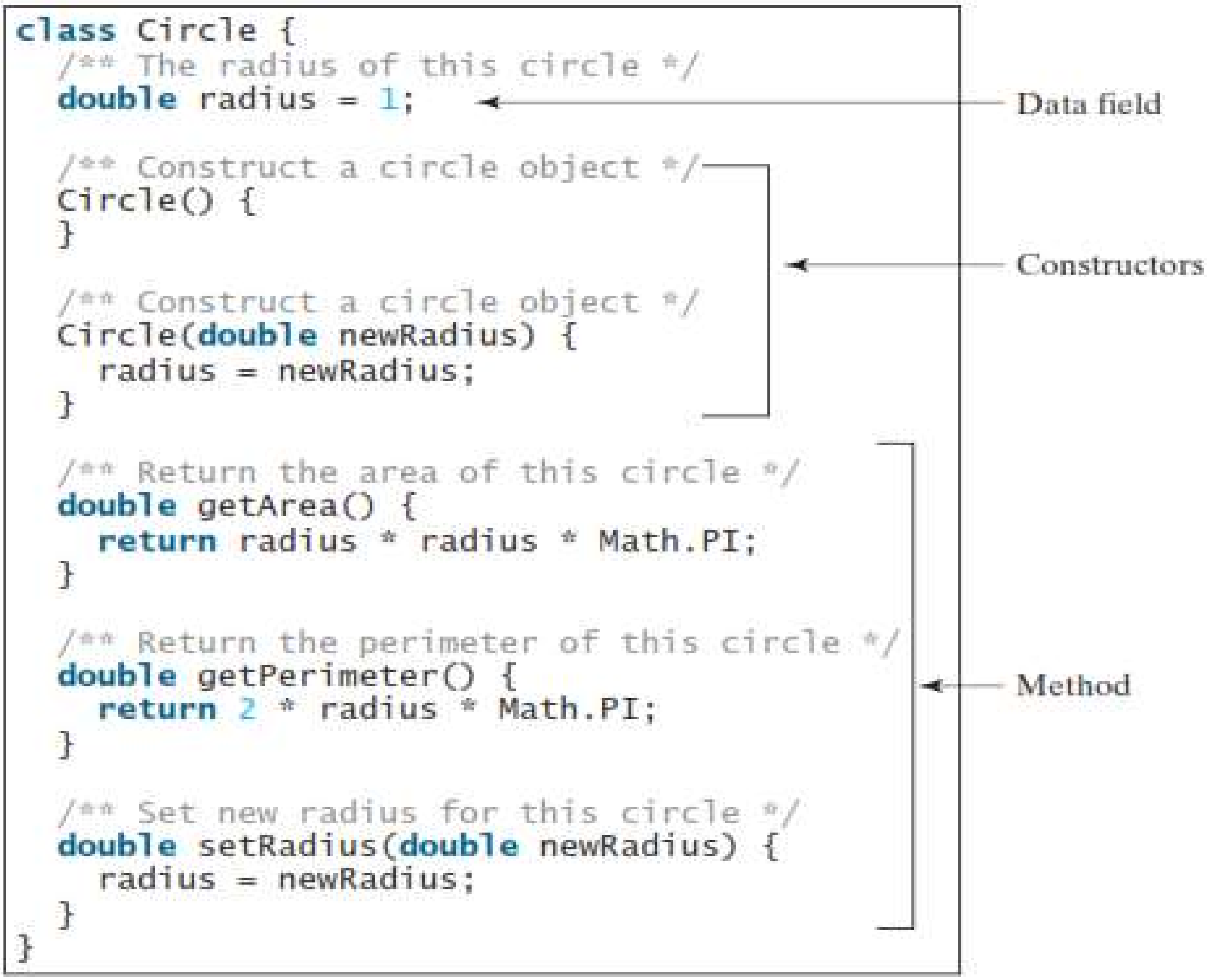
➢ You can create **many instances of a class** within a **class** you defined in a **program**

- **Creating** an **instance** is referred to as **instantiation**.
- The terms **object** and **instance** are often **interchangeable**.

➢ The **relationship** between **classes** and **objects** is **analogous** to that between an **apple-pie recipe** and **apple pies**.

✓ You can **make** as **many apple pies** as you want from a **single recipe**.

Characteristics of an Object-------

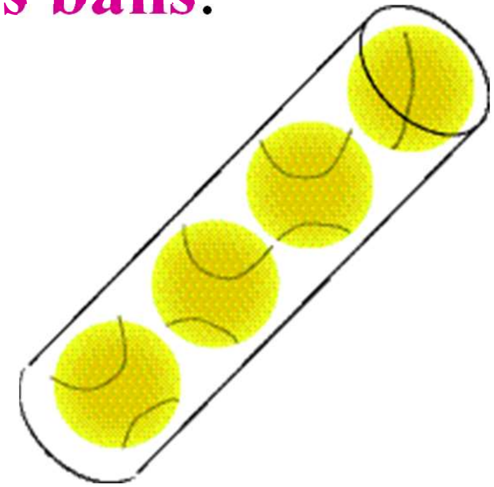


```
class Circle {
  /** The radius of this circle */
  double radius = 1;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set new radius for this circle */
  double setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

- **Figure: A class** is a **construct** that **defines objects** of the **same type.**

Example on Characteristics of Objects……

- **Consider a tube of four yellow tennis balls**.



1. Is the tube of tennis balls an object?

2. Is each tennis ball an object?

3. Could the top two balls be considered a single object?

4. Is the color of the balls an object?

5. Is your understanding of tennis balls an object?

Characteristics of an Object continued--

1. Is the **tube** of **tennis balls** an **object?**

- **Yes.** It has **identity**, it has **state** (opened, unopened, brand name, location), and **behavior** (although not much.)

2. Is each tennis ball an object?

- **Yes**. It is oK for **objects** to be **part of other objects**.
- Although each ball has nearly the **same state and behavior** as the others, each has its **own identity**.

3. Could the **top** two **balls** be **considered** a **single object?**

- **Not ordinarily.** Each has its **own identity independent** of the other.

Characteristics of an Object continued--

- If they were joined together with a stick you might consider them as **one object.**

4. **Is** the **color** of the **balls** an **object?**

- **No**. It is a **property** of **each ball**.

5. **Is** your **understanding** of **tennis balls** an **object?**

- **Probably not**, although it is unclear what it is.
- Perhaps it is a **property** of the **object** called "**your brain**."

General Syntax to Define Classes

- A class is declared by use of the **class keyword**. **Syntax**

```
    [Access Modifier] class <ClassName> {
            data-type1 instance-variable1;
             data-type2 instance-variable2;
            ------------------------------------
            -----------------------------------
             data-typeN instance-variableN;
    type methodName1(parameter-list) {
                    // body of method1
            }//End of methodName1()
```

General Syntax to Define Classes----

```
type methodName2(parameter-list) {
                    // body of method2
              }//End of methodName2()
       -----------------------------------------
       -----------------------------------------
       type methodNameN(parameter-list) {
                    // body of methodN
              }//End of methodNameN()
       }//End of class
```

General Syntax to Define Classes------

- **Access modifiers** define what **classes can access** this **class.**

➢ **Valid access modifiers** for **class** are**:**

- **public**: A public class **means** that**,** other **classes** can **access** this **class**.
- **abstract**:- It is a class that objects are **not instantiated** and its method is **defined** as **abstract** and this **method** is **implemented** by **other class**.
- **final**:-A class that is defined to protect a **super class** is **not extended,** its **instance variable** is **constant** (not changed by other classes) and its **method** is **not overloaded** and **override**.

➢ If **no access modifier** is **declared,** it **defaults** to **public**

- The **naming convention** for the **name** of a **class** (ClassName) **states** that **classes** should **begin** with a **capital letter.**

General Syntax to Define Classes------

- The **data,** or **variables, defined** within a **class** are **called instance variables.**

- The **code** is contained within **methods** (methods contain the **executable code** of a **class** and **define** the **behavior** of **objects)**

✓ Collectively, the **methods** and **variables** defined within a **class** are called **members** of the **class**.

- In most classes, the **instance variables** are **acted upon** and **accessed** by the **methods defined** for that **class**.

✓ Thus, it is the **methods** that determine how a **class' data** can be **used.**

Define Instance Variables of a class

- A **class's variables** are called **fields**.

✓ A field declaration consists of **access modifier, a type name followed** by the **field name** and optionally an initial value for the field

- **Syntax**: **<access modifier> <type> <variable name>;**

e.g. public String name;

private int count;

➢ **Valid access modifiers** for **fields** or **instance variables** are**:**

- **public**:-available to all **methods** and **classes**
- **private**:-available only to methods in the class
- **protected**:-available only to **methods** in the **class**, it's children, and other classes in the same package.

✓ **Naming conventions**, **method name** should **begin** with a **lower case letter** and **followed** by **capital letter** if **method names** are **phrases.**

Example: Defining a class Variables

- A **class Box** defines three instance variables: width, height, and depth. Currently, Box does not contain any methods.

```
class Box {
double width;
double height;
double depth;
}//End of Box class
```

- As **stated earlier**, a **class defines** a **data new type**.
- In this case, the new **data type** is called **Box.**
- Use this name to **declare** and **create** **objects** of **type Box**.

Example: Defining a class Variables------

- ➢ A **class declaration** only **creates** a **template**; it does **not create** an **actual object**.
- ▪ Thus, the preceding java code **does not cause** any **objects** of **type Box** to come into **existence**.
- ▪ To actually **declare** and **create** a **Box object**, you will use a **statement** like the following:

  // Declare and create a Box object called mybox

  Box mybox = new Box();

- ▪ After this **statement executes, mybox** will be an **instance** of **Box**.
- ✓ Thus, it will have **“physical” reality.**

Example: Defining a class Variables------

➢ Again, each time you **create** an **instance** of a **class**.

- You are **creating** an **object** that **contains** its **own copy** of each **instance variable defined** by the **class.**

✓ Thus, every **Box object** will contain its **own copies** of the **instance variables width, height,** and **depth**.

➢ To **access** these instance **variables** defined by a class, use the **dot (.) operator**.

- The **dot operator links** the name of the **object** with the name of an **instance variable**.

Example: Defining a class Variables------

- **For example,** to **assign** the **width variable** of **mybox** the **value 100**, you would use the following java **statement**:

  mybox.width = 100;

- This **statement tells** the **compiler** to **assign** the **copy** of **width** that is contained within the **mybox object** the **value** of **100**.

- ✓ **In general,** use the **dot operator** to **access** both the **instance variables** and the **methods** within an **object**.

Activity 1

1. Define a class named Box with three instance variable namely width, height and depth. Define another class named BoxDemo that this class contains the main method and used to create objects of Box class named mybox1. Assign values to mybox1's instance variable of a class. Write the complete java code to compute volume of Box and prints volume of box as an output .

Activity 1

```
//Define a class named Box
class Box {
//Define instance variable of Box class
double width;
double height;
double depth;
}//End of Box class
//Define another class named BoxDemo to create objects of Box class
class BoxDemo {
//Main Method ()
public static void main(String args[]) {
//Declare and create objects  of Box class named mybox1
```

Activity 1-------

```
Box mybox1= new Box();
//Declare a variable within main to store the value of volume of Box
double vol;
//Assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
vol = (mybox1.width * mybox1.height * mybox1.depth);
System.out.println("Volume of Box:" + vol);
}//End of main ()
}//End of BoxDemo class
```

Activity 1-------

- The previous java program works as follows:
  - **Call** the **file** that contains this program **BoxDemo.java**, because the **main( ) method** is in the class **BoxDemo, not** the class **Box.**
  - When you **compile** this **program**, you will find that **two .class files have been created**, **one** for **Box** and **one** for **BoxDemo.**
  - The **Java compiler automatically** puts each class into its **own .class file.**
  - To **run this program**, you must **execute BoxDemo.class**
- It is **not necessary** for both the **Box** and the **BoxDemo** class to actually be in the **same source file.**
  - You could put **each class** in its **own file,** called **Box.java** and **BoxDemo.java**, respectively.

Activity 1-------

- As stated earlier, **each object** has its own **copies** of the **instance variables**.
  - This means that if you have **two Box objects**, **each** has its **own copy** of **depth, width, and height.**
  - The **changes** to the **instance variables** of **one object** have **no effect** on the **instance variables** of another.
- **For example:**
  - The following java program is based on the previous program to demonstrates to **declares two Box objects** of Box class named mybox1 and mybox2. Each object has its own copy of instance variables of Box class, width, height, and depth respectively**:**

Activity on two objects

```
//Define a class named Box
class Box {
//Define instance variable of Box class
double width;
double height;
double depth;
}//End of Box class
//Define class named BoxDemo to create objects of Box class
class BoxDemo {
//Main Method ()
public static void main(String args[]) {
```

Activity on two objects

```
//Declare and create objects  of Box class, mybox1 and mybox2
Box mybox1= new Box();
Box mybox2=new Box();
//Declare a variable within main to store value of volume of Box
double vol;
//Assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
//Assign values to mybox2’s instance variable
mybox2.width=3;
```

Activity on two objects

```
mybox2.height=6;
mybox2.depth=9;
//Compute Volume of mybox1
vol = (mybox1.width * mybox1.height * mybox1.depth);
System.out.println("Volume of Box1:" + vol);
//Compute Volume of mybox2
vol = (mybox2.width * mybox2.height * mybox2.depth);
System.out.println("Volume of Box2:" + vol);
}//End of main ()
}//End of BoxDemo class
```

Declaring Objects

➢ **Obtaining objects** of a **class** is a **two-step process**.

1. **First**, **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**.

2. **Second, acquire** an **actual, physical copy** of the **object** and **assign** it to that **variable**, using **new operator**.

- The **new operator dynamically allocates** (that is**, allocates** at **run time)** **memory** for an **object** and **returns** a **reference** to it.

➢ This **reference** is, more or less, the **address** in **memory** of the **object allocated** by **new**.

- This reference is then **stored** in the **variable**.

Declaring Objects-----

- ✓ Thus, in **Java,** all **class objects must** be **dynamically allocated**.
- ▪ Let's see the following java statement used to **declare** an **object** of **type Box** named **mybox1:**

  Box mybox1 = new Box();

- ✓ This statement combines the **two steps just described**.
- ▪ It can be **rewritten** like this to **show each step more clearly**:

  Box mybox1; // Declare variable of class type Box

// Allocate memory for Box object and **assign** it to that **variable**

  mybox1 = new Box();

- ➢ The **first** line **declares mybox** as a **reference** to an **object of type Box.**

Declaring Objects-----

- After this **line executes,** **mybox1 contains** the **value null**, which indicates that it **does not** yet point to an **actual object**.
- Any attempt to use **mybox1** at this **point** will **result** in a **compile-time error.**

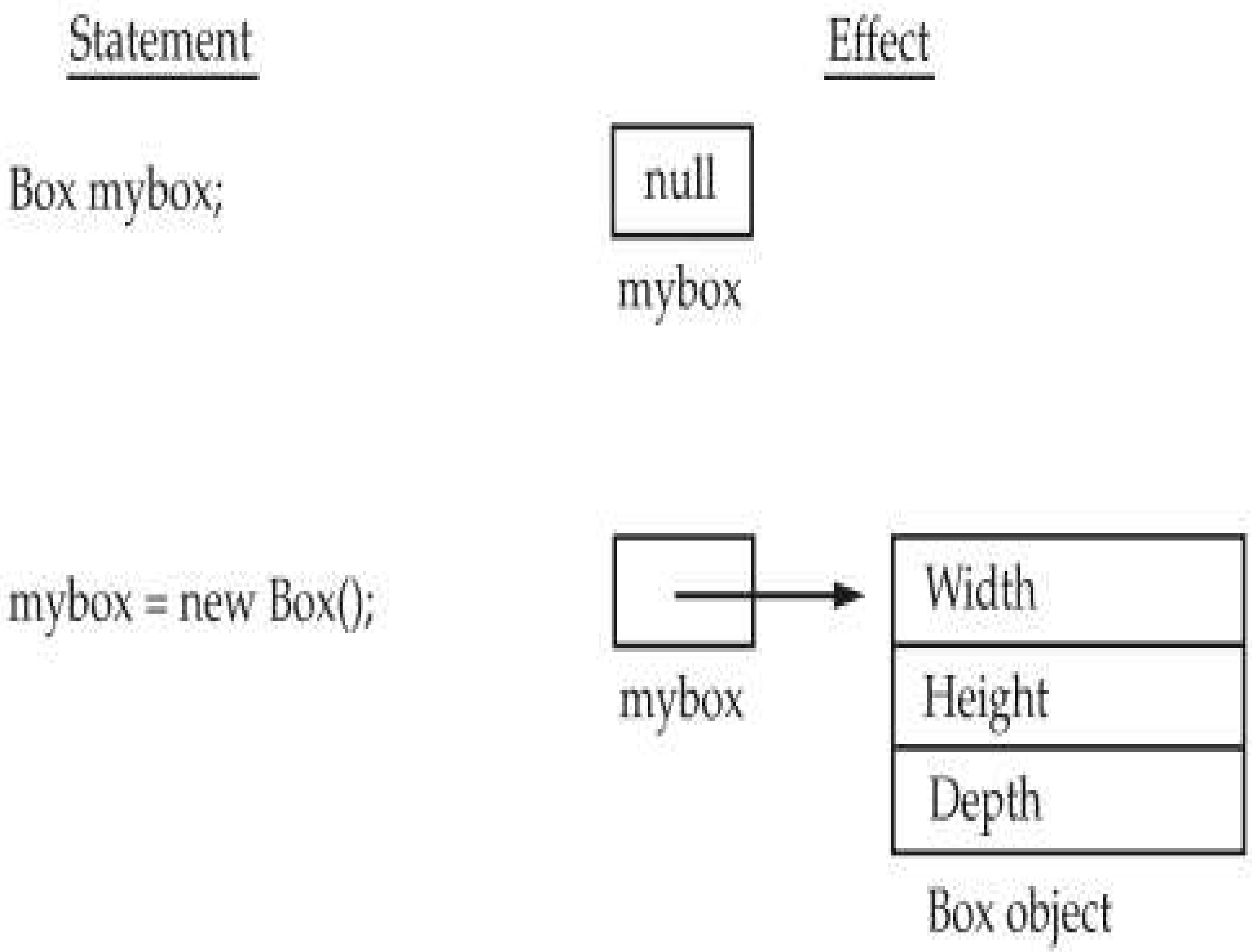
➢ The **next line allocates** an **actual object** and **assigns** a **reference** to it to **mybox1**.

- After the second line **executes**, you can use **mybox1** as if it **were** a **Box object.**
- **But** in **reality,** **mybox1** simply **holds** the **memory address** of the **actual** **Box object**.

✓ The **effect** of these **two lines** of **code** is depicted in Figure.

Declaring Objects-----

- ✓ Fig: Shows the declaration and creation of objects of type Box class



New Operator

- As just explained, the new operator **dynamically allocates memory** for an **object**.
- **Syntax:** **class-var = new classname( );**
- Here, **class-var** is a **variable of the class type being created**.
- The **classname** is the **name** of the **class** that is **being instantiated**.
- The class name **followed** by **parentheses** specifies the **constructor** for the **class**.
- A **constructor** defines **what** occurs **when** an **object** of a **class** is **created**.
- **Most real-world classes** explicitly define their **own constructors** within their **class** definition.

New Operator-----

- However, if **no explicit constructor** is specified, then **Java** will **automatically** supply a **default constructor**.
- This is the case with Box. For now, we will use the **default constructor**.

➢ At this point, you might be wondering **why** you do **not need** to **use new** for such **things** as **integers** or **characters**.

- The answer is that **Java's** simple **types** are not **implemented** as **objects**.
- Rather, they are implemented as **"normal" variables**.

✓ This is done in the **interest** of **efficiency.**

New Operator-----

- **Objects** have many **features** and **attributes** that **require Java** to treat them **differently** than it treats the **simple types**.
- It is important to understand that **new allocates memory** for an **object** during **on fly.**
- The **advantage** of this **approach** is to **create** as **many** or as few **objects** as it needs during the **execution** of the **program**.
- However, since **memory** is **finite**, it is possible that **new** will **not** be able to **allocate memory** for an **object** because **insufficient memory exists**.

New Operator-----

- If this happens, a **run-time exception** will **occur**.

- A **class creates** a **new data type** that can be used to **create objects**.

- That is, a **class creates** a logical **framework** that defines the **relationship between** its **members**.

- When you declare an **object** of a **class**, you are **creating** an **instance** of that **class**. Thus, a **class** is a **logical construct**.

- An **object** has **physical reality**. (That is, an **object occupies space in memory**.)

Assigning Object Reference Variables

➢ **Object reference variables** act differently than you might expect when an **assignment** takes place.

- **For example,** what do you think the **following java fragment** of **code does?**

```
Box b1 = new Box();
Box b2 = b1;
```

- You might think that **b2** is being **assigned** a **reference** to a **copy** of the **object referred** to by **b1**.
- That is, you might think that **b1** and **b2 refer** to **separate** and **distinct objects.**

✓ However, this would be **wrong**.

Assigning Object Reference Variables---

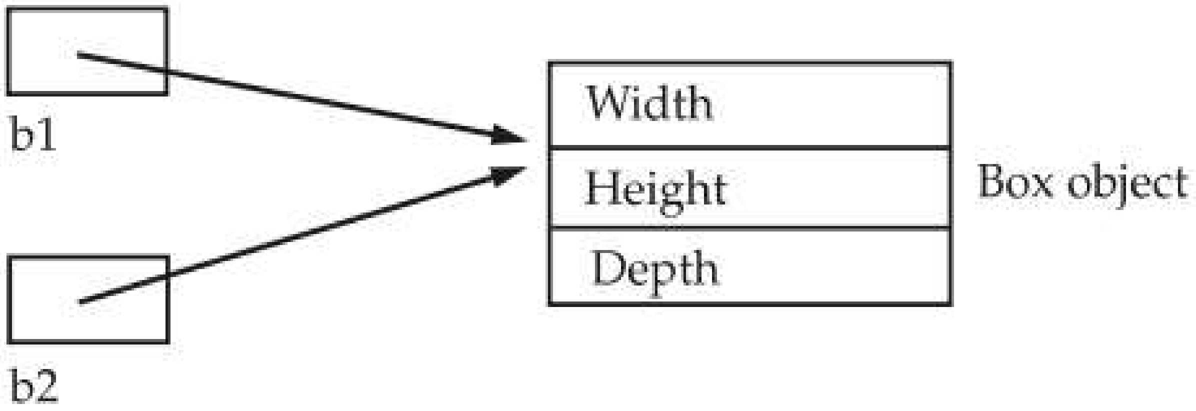
- Instead, after this **fragment executes, b1** and **b2** will **both refer** to the **same object**.

➢ The **assignment** of **b1 to b2** did **not allocate** any **memory or copy any part** of the **original object.**

- It simply makes **b2 refer** to the **same object** as **does b1**.
- Thus, any **changes** made to the **object through b2** will **affect** the **object** to which **b1** is **referring,** since they are the **same object.**

➢ This situation is depicted by the following figure on the next slide:

Assigning Object Reference Variables----



- Although **b1** and **b2** both **refer** to the **same object,** they are **not linked** in any other way.
- For example, a **subsequent assignment** to **b1** will simply unhook **b1** from the **original object** without **affecting** the **object** or **affecting b2**. For example:

Box b1 = new Box();

Box b2 = b1;

Assigning Object Reference Variables----

```
// -------------------------
//---------------------------
b1 = null;
```

- Here, **b1 has been set** to **null,** but **b2** still **points to** the **original object**.

➢ When you **assign one object reference variable** to **another object reference variable**, you are **not creating** a **copy** of the **object**, you are only **making** a **copy** of the **reference**.

Activity 2

- Use the following information and write java program to demonstrate to assign one object reference variable to another object reference variable, to make both refers to the same object. The program shows that changes made to the object through one object will affect the object to which another object is referring to.
- Define a class named Box with three instance variable namely width, height and depth.
- Define another class named BoxDemo that this class is used to creates objects of type Box class
- Declare and create objects of type Box class named b1 and b2 and make b2 refers to b1 (Both b1 and b2 refers to the same object).
- Assign values to b1’s instance variable and any different values assigned to b2’s instance variable affects the volume of box.
- Write the complete java code to compute volume of Box and prints volume of box as an output

Activity 2--------

```
//Define a class named Box
class Box {
//Define instance variable of a class
double width;
double height;
double depth;
}//End of Box class
// Define another class named BoxDemo to declares object of type Box.
class BoxDemo {
```

Activity 2--------

```
//Main method
public static void main(String args[]) {
//Declare and create objects of type Box named b1
Box b1 = new Box();
//Make b2 refers to the same object as does b1
Box b2=b1;
//Declare another variable to store volume of Box
double vol;
//Assign values to b1’s instance variable
b1.width = 10;
b1.height = 20;
```

Activity 2--------

```
b1.depth = 15;
//Assign values to b2’s instance variable
/* Any changes made to the object through b2 will affect the object
to which b1 is referring, since they are the same object */
b2.width=10;
b2.height = 20;
//Change the depth value through b2
b2.depth = 30;
// Compute volume of box 1and print the output
vol = b1.width * b1.height * b1.depth;
System.out.println("Volume is " + vol);
```

Activity 2--------

```
//Compute Volume of box 2
vol=b2.width*b2.height*b2.depth;
//Print volume of box2 as an output
System.out.println("Volume is " + vol);
}//End of main ()
   }//End of BoxDemo class
```

Defining Methods

- **Syntax:**

```
type methodName(parameter-list) {
// body of the method
}//End of methodName()
```

- Here, **type** specifies the **type of data returned by the method**.

- This can be **any valid type, including class types that you create.**
- If the method **does not return a value**, its **return type must be void**.

Defining Methods------

- The **name** of the **method** is **specified** by **name**.
- This can be any legal identifier other than those already used by other items within the current scope.
- The **parameter-list** is a **sequence** of **type** and **identifier pairs** separated by commas.
- **Parameters** are essentially **variables** that **receive** the **value** of the **arguments passed** to the **method when** it is **called.**
- **Arguments** are **values passed** to a **method** and received by the **parameter** when the method is called.

Defining Methods---

- If the **method** has **no parameters**, then the **parameter list will be empty.**
- **Methods** that have a **return type other than void return** a **value to the calling routine** using the following **form of the return statement:**

  return value;

- Here, **value is the value returned**.

Adding a Method to the Box Class

- To **create** a **class** that **contains** only **data,** it rarely **happens.**

- Most of the time you will use **methods** to **access** the **instance variables** defined by the **class**.
- In fact, **methods** define the **interface** to most **classes**.
- This allows the **class implement** or to **hide** the **specific** layout of **internal data structures** behind cleaner method abstractions.
- In addition to **defining methods** that provide **access to data**, you can also **define methods that are used internally by the class itself.**
- Adding methods to a Box class is either **without** a **parameter** or **with** a **parameter**

Adding non-parameterized method to a Box Class

- Define a method without parameter to a Box class. This method compute Volume of Box and prints its output when it is called from the main program.

```
//Define a class named Box
class Box{
//Define instance variable of a class
double width;
double height;
double depth;
//Define instance method, a method without parameter
```

Adding non-parameterized method to a Box Class

```
void Volume( ){
System.out.println(“Volume of Box=“+(width*height*
depth));
}//End of Volume ()
 }//End of Box ()
```

- The method named Volume() is defined as void because the method does not return value.
- The method compute volume of Box and output the value of the method when it is called.

Activity 3

- Use the previous class, instance variable and method definition and write the complete java program based on the following additional information:

- Define another class named BoxDemo. This class used to create objects of Box class.
- Declare and create objects of Box class named mybox1 and mybox2 respectively and assign different values to mybox1's and mybox2's instance variable.
- Call Volume( ) method through mybox1 and mybox2 objects separately. The Volume () compute and output the volume of Box when it is called.

Returning a Value

- The implementation of **volume( )** in Activity 3 does move the computation of a **box's volume inside** the **Box class** where it **belongs,** it is **not** the best way to do it.
  - **For example**, what if **another part** of your **program wanted** to know the **volume** of a box**,** but **not display** its **value?**
  - A better way to **implement volume( )** is to have it **compute** the **volume** of the box and **return** the **result** to the **caller.**
- The following activity is, an improved version of activity 3. This program demonstrate to define a method named Volume() and this method return the value or the result when it is called from the main program.

Activity 4

```
// Define a class named Box and its instance variable
class Box{
double width;
double height;
double depth;
/*Define instance method named volume (), this method compute
   volume of Box and return a value to the caller */
    double volume() {
    return (width * height * depth);
    }//End of volume ()
    }//End of Box ()
```

Activity 4------

```
//Define class named BoxDemo used to create objects of Box class
class BoxDemo{
//Main method()
public static void main(String args[]) {
//Declare and create objects of Box class named mybox1 & mybox2
Box mybox1 = new Box5();
Box5 mybox2 = new Box5();
double vol;
// Assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height=20;
mybox1.depth=15;
```

Activity 4------

```
//Assign different values to mybox2's instance variables
mybox2.width=3;
mybox2.height=6;
mybox2.depth=9;
// Call and get volume of first box
vol = mybox1.volume();
//Output volume of 1st box
System.out.println("Volume is " + vol);
// Call and get volume of second box
vol = mybox2.volume();
```

Activity 4------

```
//Output Volume of second box
System.out.println("Volume is " + vol);
}//End of main ()
}//End of class
```

- **Note**: Two important things to understand about **returning values:**

1. The **type** of **data returned** by a **method** must be **compatible** with the **return type** specified by the **method**.

- **For example**, if the **return type** of some method is **boolean**, you could **not return** an **integer**.

2. The **variable receiving** the **value returned** by a **method** (such as vol, in this case) must also be **compatible** with the **return type** specified for the **method.**

Note

- The following java program is a bit more efficient than the program in activity 4 because this program call volume() through println (), without using vol variable to receive the return value like as follows:

```
System.out.println ("Volume is " + mybox1.volume());
```

- The **call** to **volume( )** could have been used in the **println( ) statement directly.**
- When **println( )** is **executed**, **mybox1.volume( )** will be **called automatically** and its **value** will be **passed** to **println( )**.
- Let's see the following **program** to **implement** the **call** to **volume()** through **println()** statement directly.

Activity 5

- Use the following information and write java program to calculate volume() of Box and return a value when it is called directly through println() method.
- Define a class named Box with instance variable of width, height and depth respectively.
- Define instance method named volume() inside Box class and this method return a value to the main() program when it is called through println().
- Define another class named BoxDemo and this class is used to declare and create objects of Box class and contains the main () method of the program.

Activity 5------

- Declare and create objects of Box class named mybox1 and mybox2 respectively.
- Assign different values to mybox1’s and mybox2’s instance variables separately.
- Call volume() directly through println() and output the returned value of the program as follows:

```
System.out.println("Volume is " + mybox1.volume());
System.out.println("Volume is " + mybox2.volume());
```

Method that takes parameters

- The, **parameterized method** can operate on a **variety** of **data** and/or be used in a **number** of slightly **different situations**.
- Here is a method that returns the **square** of the **number 10**:

```
int square() {
return 10 * 10;
}//End of Square()
```

- While this **method** does**,** indeed**, return** the value of **10 squared,** its use is very limited.
- However, if you **modify** the **method** so that it **takes** a **parameter,** as shown next, then you can make square( ) much more

Method that takes parameters----

```
//Define parameterized method
int square(int i) {
return (i * i);
}//End of square()
```

- Now, square( ) will return the **square** of whatever value it is called with.
- That is, **square( )** is now a **general-purpose method** that can **compute** the **square** of any **integer value**, rather than just **10**.

Method that takes parameters----

- The following java program is to demonstrate parameterized method(). The **square( ) method** **compute** the **square** of any integer value when it is called from the main program in different ways:

```
//Define a class named Parameter
   class Parameter{
//Define parameterized method
        int square(int i){
             return (i * i);
        }//End of square()
   }//End of Parameter class
```

Method that takes parameters----

```
//Define another class to create objects of Parameter class
class ParameterizedMethods {
//Main method ()
public static void main(String args[]) {
//Declare and create objects named par type Parameter class
Parameter par=new Parameter();
//Declare variables named x and y local to main
 int x, y;
/*Call square() by passing 5 as an argument, store the returned
       value to variable x and output its returned value */
    x=par.square(5);
```

Method that takes parameters----

```
System.out.println("X="+ x);
/*Call square() for 2nd time by passing 9 as an argument, store the
        returned value to variable x and output its returned value */
x =par.square(9);
System.out.println("X="+ x);
//Initialize y to 2
y=2;
//Call square() for 3rd time by passing y as an argument and output
y= par.square(y);
 System.out.println("Y="+ y);
    }//End of main ()
}//End of class
```

Method that takes parameters----

- **Note**:
- ✓ It is important to keep the two terms **parameter** and **argument** straight.
- A **parameter** is a **variable** defined by a **method** that **receives** a **value** when the **method** is **called**.
- ✓ **For example**, in square(int i) , **i** is a **parameter**.
- An **argument** is a **value** that is **passed** to a **method** when it is **invoked**.
- ✓ **For example**, the call to x=par.square(100) **passes 100** as an **argument**.
- Inside **square(int i),** the **parameter i receives** that **value**.

Activity 6

➢ Write java program to demonstrate adding parameterized method named volume() to a Box class and write the complete java program based on the following information.

▪ Define a class named Box with instance variable of width, height and depth.

▪ Define parameterized instance method named volume() inside the Box class.

✓ The parameters receive the values when it is called and assigns to each instance variables.

✓ The method compute volume of Box when called and returns the result to the main program

Activity 6-------

- Define another class named BoxDemo, this class is used declare and create objects of Box class
- Declare objects of type Box named box1 and box2 respectively. Assigning values to each box's is not necessary because arguments are passed to a method when volume() is called.
- Call volume() by passing arguments through box1 and box2 respectively and the returned value is stored on vol variable.
- The program outputs the returned result of volume of Box separately

Constructors

- It can be **tedious** to **initialize** all of the **variables** in a class each **time** an **instance** is **created**.
- Because the **requirement** for **initialization** is so common.
- **Java** allows **objects** to **initialize themselves** when they are **created.**
- This **automatic initialization** is performed through the use of a **constructor.**
- A **constructor** **initializes** an **object** immediately upon **creation**.
- It has the **same name** as the **class** in which it resides and is syntactically similar to a **method**

Constructors----

- The **constructor** is **automatically called immediately** after the **object** is **created**, before the **new operator completes**.

➢ **Constructors** look a little strange because they have **no return type, not even void.**

- This is because the **implicit return type** of a **class' constructor** is the **class type itself**.

- It is the constructor's job to **initialize** the **internal state** of an **object** so that the **code creating** an **instance** will have a **fully initialized**, usable **object immediately**.

Constructors----

- You can **rework** the **Box example** so that the **dimensions** of a box are **automatically initialized** when an **object** is **constructed.**
- To do so, **replace setDim( )** with a **constructor**.

➢ Let’s begin by **defining a simple constructor** that simply **sets** the **dimensions** of each **box** to the **same values**.

Activity 7

- Write Java program to define a constructor named Box and to sets the dimensions of each box to the same values.

- Define a class named Box with instance variable of width, height and depth.
- Define constructor without parameter and sets the dimensions of each Box to the same value.
- Define a method without parameter named volume(), the method compute volume of Box and return the result to the caller.
- Define another class to create objects of Box class and this object calls constructors immediately upon an object is created

Activity 7-------

- Declare and create objects of Box class named box1 and box2 and these objects automatically calls the Box constructor.
- Call volume() through box1 and box2 separately and output the returned value.
- The output of the program is the same because at each time when the constructor is called, it sets the same dimensions for Box instance variable.

Activity 7-------

```
class Box {

//Define instance variable of a class

double width;

double height;

double depth;

// Define Box Constructor and sets each dimensions to the same value

Box() {

System.out.println("Constructing Box8");

width = 10;

height = 10;
```

Activity 7-------

```
depth = 10;
}//End of constructor
// Define volume (), compute volume of Box and return a value
double volume() {
return (width * height * depth);
}//End of Volume()
}//End of Box class
class ConstructorExample{
public static void main(String args[]) {
// Declare, allocate, and initialize Box objects
Box box1 = new Box();
Box box2 = new Box();
```

Activity 7-------

```
//Declare variable named vol local to main
double vol;
// get volume of first box and output the returned value
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box and output the returned value
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}//End of main ()
}//End of class
```

Activity 7-------

- As you can see from the previous Java program, both **box1** and **box2** were **initialized** by the **Box( ) constructor** when they were **created**.
- Since the **constructor** gives all **boxes** the **same dimensions, 10** by **10** by **10,** both box1 and box2 will have the **same volume**.
- The **println( ) statement** inside **Box( ) constructor** is for the sake of illustration only.
- Most **constructors** will **not display anything**. They will simply **initialize** an **object**.
- When you **allocate** an **object,** you use the following **general form:**

  class-var = new classname( );
- Now you can understand **why** the **parentheses** are needed after the **class name**.

Activity 7-------

- What is actually happening is that the **constructor** for the **class** is **being called**. Thus, in the line: **Box box1 = new Box();**

  new Box( ) is calling the Box( ) constructor.

- When you do **not explicitly define** a **constructor** for a **class**, then **Java creates** a **default constructor** for the class.

- This is why the preceding line of code worked in earlier versions of Box that did **not define** a **constructor.**
- The **default constructor** automatically **initializes all instance variables to zero**.
- The **default constructor** is often sufficient for simple classes, but it usually **won't** do for more **sophisticated** ones.
- Once you **define** your **own constructor**, the **default constructor** is **no longer** used

Parameterized Constructors

- While the **Box( ) constructor** in the **preceding example (in Activity 7),** does **initialize** a **Box object**, it is **not** very **useful**—all **boxes** have the **same dimensions**.
- What is needed is a way to construct Box **objects** of **various dimensions**?

✓ The easy solution is to **add parameters** to the **constructor**.

➢ **For example,** the following version of **Box defines** a **parameterized constructor** which **sets** the **dimensions** of a box as specified by those **parameters**.

Activity 8

- Write java program to demonstrate to use a parameterized constructor to set the dimensions of the Box instance variable to different values.

```
//Define class named Box
class Box {
//Define instance variable of a Box class
double width;
double height;
double depth;
```

Activity 8---------------

```
// This is the parameterized constructor for Box
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}//End of Box constructor
//Define volume (), compute and return the result of volume
double volume() {
return width * height * depth;
}//End of volume()
}//End of Box class
```

Activity 8---------------

```
//Define another class to create objects of Box class
class ParameterizedConstructor{
public static void main(String args[]) {
//Ddeclare, allocate, and initialize Box objects
Box box1 = new Box(10, 20, 15);
Box box2 = new Box(3, 6, 9);
//Declare a variable named vol, local to main function
double vol;
// get volume of first box and output the returned value
vol = mybox1.volume();
System.out.println("Volume is " + vol);
```

Activity 8--------------

```
// get volume of second box and output the returned value
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}//End of main ()
}//End of class
```

- As you can see, each **object** is **initialized** as specified in the **parameters** to its **constructor**.
- Box box1 = new Box(10, 20, 15); the values 10, 20, and 15 are **passed** to the **Box( ) constructor** when **new creates** the **object.**
- Thus, **box1's copy** of **width, height,** and **depth** will **contain** the **values 10, 20,** and **15**, respectively.

The this Keyword

- ➢ Sometimes a **method** will **need** to **refer** to the **object** that **invoked** it.
- ▪ To allow this, **Java defines** the **this keyword**.
- ✓ **this** 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.**
- ▪ You can use this **anywhere** a **reference** to an **object** of the **current class' type** is permitted.
- ▪ To better understand what **this refers** to, consider the following version of Box( ):

The this Keyword-------

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

- This version of Box( ) operates exactly like the earlier version.

✓ 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, one of which is **explained** in the **next section**.

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

✓ This is why **width, height,** and **depth** were **not** used as the **names** of the **parameters** to the **Box( ) constructor inside** the **Box class**.

Instance Variable Hiding---------

- If they had been, then **width** would have **referred** to the **formal parameter**, **hiding** the **instance variable** **width**.
- While it is usually **easier** to **simply** **use different names**, there is another way around this situation.
- Because **this** lets you refer directly to the **object**, you can use it to **resolve** any **name space collisions** that might **occur between** **instance variables** and **local variables**.

➢ **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 keyword** to **resolve name-space collisions**.

```
  Box(double width, double height, double depth) {
```

Instance Variable Hiding---------

```
this.width = width;
this.height = height;
this.depth = depth;
}//End of constructor
```

- **Caution:**

- The use of **this** in such a context can sometimes be **confusing**, and some programmers are careful **not** to **use local variables** and **formal parameter names** that **hide instance variables.**
- Of course, other programmers believe the contrary—that it is a good convention to use the **same names** for **clarity**, and use **this keyword** to **overcome** the **instance variable hiding**.

Activity 9

- Write Java program to demonstrate this keyword i.e., this keyword refers to refer directly to the **object**, on which the **method invoked.** Use the **this keyword** to **resolve** any **name space collisions** that might **occur** between **instance variables** and **local variables** based on the following additional information:

- Define a class named ThisKeyword with instance variable of m and n.
- Define parameterized constructor, the parameter or local variable to the constructor is the same name as instance variable of the class.

Activity 9

- Define a method named sum() and this method compute and return the sum of (m+n) when it is called through objects of the class.
- Define a method named square() and this method compute and return the square of the of n (n*n) when it is called.
- Define another class to create objects of ThisKeyword class named obj and this objects initialize the instance variable automatically when the constructor is called.
- Call Sum() and Square() method separately, output the returned result.

Activity 9------

```
//Define a class named ThisKeyword
class ThisKeyWord {
//Define instance variable of a class named m and n
int m, n;
/*Define Parameterized constructor, the parameter name is the same
   name as the instance variable. This local variables hides the instance
   variable of a class. Use this keyword to resolve any name space
   collisions */
    ThisKeyWord(int m, int n){
       this.m = m;
       this.n=n;
   }//End of constructor
```

Activity 9------

```
//Define a method named sum, compute (m+n) and return the result
int sum(){
return(m+n);
}//End of sum()
//Define a method named square(), compute (n*n) and return the result
int square(){
   return (n*n);
}//End of square()
}//End of ThisKeyword class
//Define another class to create objects of ThisKeyword class
   class ThisKeyWordTest {
     public static void main(String[] args) {
```

Activity 9------

```
//Declare, create and allocate objects of a class
ThisKeyWord obj = new ThisKeyWord (10,6);
//Declare local variables to main ()
int a, b;
//Call sum() and output the returned value
 a =obj.sum();
System.out.println(“Sum of(m+n)=“+ a);
//Call square() and output the returned value
b= obj.square();
System.out.println(“Square of (n*n)="+ b);
          }//End of main ()
}//End of class
```

Another use of **this keyword**

- The **this** is **always** a **reference** to the **object** on which the **method** was **invoked.**

✓ The **this keyword** also used to resolve any name space collisions such as when **local variable** hides the instance variable by defining with the same name as the instance variable.

- In addition to this, **constructor** of a **class can call** other **constructor** of the **same class** by **using** a **key word this.**

➢ The following **java program** is used to demonstrate constructor of a class can call constructor of the same class by using a keyword this.

Another use of **this keyword-------**

```
//Define a class named Box and instance variable of a Box class
class Box{
double width;
double height;
double depth;
/*Define Parameterized constructor with 1 dimension named w and initialized
   to width*/
Box(double w){
width = w;
  }//End of Box()
/*This constructor is with 2 dimensions (w and h), and use this keyword to call
   constructor of the previous class and initialize h to height */
Box(double w, double h){
     this(w);
     height = h;
  }//End of Box()
```

Another use of **this keyword-------**

```
/*Use Box constructor is with 3 dimensions (w, h, and d), and use this keyword
   to call constructor of the previous class and initialize d to depth */
Box(double w, double h, double d) {
      this(w, h);
      depth = d;
   }//End of Box ()
//Define a method volume(), compute volume of box and return a value
double volume(){
   return (width * height * depth);
   }//End of volume()
}//End of Box class
```

Another use of **this keyword-------**

```
//Define another class to create objects of Box class
class TestConstructor{
//Main method ()
public static void main(String args[]){
//Declare, create and allocate objects of Box class
Box b1 = new Box(5 , 6, 9);
//Declare a variable named vol, to store the returned value of box
double vol;
//Call volume() and output the result of volume of box
vol= b1.volume();
 System.out.println(“Volume of Box=“+vol);
    }//End of main ()
} //End of TestConstructor class
```

Another use of **this keyword-------**

- Let’s add the following additional java code to the previous java program to understand the sequence of the program it follows. Rewrite the previous program to clearly understand the sequence of the program by adding the following java code.

```
Box(double w){
    width = w
System.out.println(“One argument Constructor”);
}
---------------------------------------------------------------
Box(double w, double h){
     this(w);
```

Another use of **this keyword-------**

```
height =h;
System.out.println(“Two argument Constructor”);
}
--------------------------------------------------------------------
Box(double w, double h, double d){
this(w, h);
depth = d;
System.out.println(“Three argument Constructor”);
}
```

Overlloading Methods

- 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**.
- **Method overloading** is one of the ways that Java implements **polymorphism**.
- When an **overloaded method is invoked**, Java uses the **type and/or number of arguments** as its guide to determine which version of the overloaded method to actually call.

Overloading Methods-------

- Thus, overloaded methods must differ in the **type and/or number of their parameters.**
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- 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.

Activity 10

```
// Define a class named OverloadDemo

class OverloadDemo {

//Overload test () without parameter

void test() {

System.out.println("No parameters");

}

// Overload test for one integer parameter.

void test(int a) {

System.out.println("a: " + a);

}
```

Activity 10----------

```
// Overload test for two integer parameters.

void test(int a, int b) {

System.out.println("a and b: " + a + " " + b);

}

// overload test for a double parameter

double test(double a) {

System.out.println("double a: " + a);

return (a*a);

}//End of test ()

}//End of class
```

Activity 10----------

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}//End of main ()
}//End of class
```

Activity 10----------

- As you can see, test( ) is overloaded four times.
- The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter.
- The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.
- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact.
- In some cases Java's ***automatic type conversions*** can play a role in overload resolution.

Activity 11

- ***For example, consider the following program:***

```
// Automatic type conversions apply to overloading.
    class OverloadDemo {
    void test() {
System.out.println("No parameters");
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
```

Activity 11----

```
// overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
Ob.test(10);
```

Activity 11----

```
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}//End of main ()
}//End of class
```

- As you can see, this version of OverloadDemo does not define test(int).
- Therefore, when test( ) is called with an integer argument inside Overload, no matching method is found.
- However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call.

Activity 11----

- Therefore, after test(int) is not found, Java elevates i to double and then calls test(double).
- Of course, if test(int) had been defined, it would have been called instead.
- **Java will employ its automatic type conversions only if no exact match is found**.
- Method overloading supports **polymorphism** because it is one way that Java implements the "**one interface, multiple methods" paradigm.**

➢ To understand how, consider the following.

Activity 11----

- **In languages that do not support method overloading, each method must be given a unique name.**
- However, frequently you will want to implement essentially the ***same method for different types of data.***
- Consider the ***absolute value function***. In languages that ***do not support overloading***, there are usually three or more versions of this function, each with a slightly different name.
- For instance, in C, the function abs( ) returns the absolute value of an integer, labs( ) returns the absolute value of a long integer, and fabs( ) returns the absolute value of a floating-point value.

Activity 11----

- Since C does not support overloading, each function has to have its own name, even though all three functions do essentially the same thing.
- This makes the situation more complex, conceptually, than it actually is.
- Although the underlying concept of each function is the same, you still have three names to remember.
- **This situation does not occur in Java, because each absolute value method can use the same name.**
- Indeed, Java's standard class library includes an absolute value method, called **abs( ).**

Activity 11----

- This method is overloaded by **Java’s Math class** to handle all numeric types.
- Java determines which version of **abs( ) to call** based upon the type of argument.
- The value of overloading is that it allows related methods to be accessed by use of a common name.
- Thus, the name abs represents the general action which is being performed.
- It is left to the compiler to choose the right specific version for a particular circumstance. You, the programmer, need only remember the general operation being performed.

Activity 11----

- Through the application of ***polymorphism***, several names have been reduced to one.
- Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.
- When you overload a method, each version of that method can perform any activity you desire.
- There is no rule stating that overloaded methods must relate to one another.
- However, from a stylistic point of view, method overloading implies a relationship.

Activity 11----

- Thus, while you can use the same name to overload unrelated methods, you should not.
- For example, you could use the name sqr to create methods that return the square of an integer and the square root of a floating-point value.
- But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose.
- In practice, you should only overload closely related operations.

Garbage Collection------

- **Garbage Collection** is process of **reclaiming** the **runtime unused memory automatically**.
- **Garbage collection** in **Java** is the **process** of **automatically freeing heap memory** by **deleting unused objects** that are **no longer accessible** in the **program**.

✓ In other simple words, the **process** of **automatic reclamation** of **runtime unused memory** or to **destroy unused objects**.

- The **program** that **performs garbage collection** is called a **garbage collector** or simply a **collector** in **java**.
- **Garbage collection** is a part of the **Java platform** and is one of the major **features** of the **Java Programming language**.

Garbage Collection-------

- **Java garbage collector runs** in the **background** in a **low-priority thread** and **automatically cleans up heap memory** by **destroying unused objects**.
- However, before **destroying unused objects**, make sure that the **running program** in its current state will never use them **again**.

✓ This **way**, it **ensures** that the **program** has **no reference variable** that **does not refer** to **any object**.

➢ **Advantage of Garbage Collection**

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

Garbage Collection-------

- An **object** that **cannot** be **used** in the **future** by the **running program** is known as **garbage** in **java**.

✓ It is also known as **dead object** or **unused object**.

- **For example**, an **object exists** in the **heap memory**, and it can be **accessed only** through a **variable** that **holds references** to that **object**.

- What should be done with a **reference variable** that is **not pointing** to any **object**?

✓ Consider the following **java statements** below:

```
Hello h1 = new Hello();

Hello h2 = new Hello();

h1 = h2;
```

Garbage Collection-------

- Here, we have assigned one reference variable h1 to another reference variable h2.
- After the assignment statement h1 = h2, h1 refers to the same object referenced by h2 because the reference variable h2 is copied to variable h1.
- Due to which the reference to the previous object is gone.
- Thus, the object previously referenced by h1 is no longer in use i.e. the object referred by the reference variable h1 that is left side to the assignment operator, is not referring to the previous object and therefore is known as garbage or dead object.

Garbage Collection-------

- Since garbage occupies memory space, therefore, the Java runtime system (JVM) detects garbage and automatically reclaims the memory space it occupies.
- This process is called garbage collection in java.
- The garbage collector has the responsibility to keep track of which objects are “garbage.”

How can an object be unreferenced?

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

1. By nulling the reference

- When we explicitly assign null to the reference variable, the object pointed by that reference variable is not referred to or unused.
- Consider the following two statements below:

```
Student st = new Student("Ivaan Sagar");
  st = null;
```

How can an object be unreferenced?------

- In the first statement, a reference to a newly created Student object is stored in the reference variable st.
- But in the next statement, the value of st is changed, and the reference to the Student object is gone.
- In this situation, JVM will automatically detect unused object if an object is not referenced by any reference variable.

How can an object be unreferenced?------

2. By assigning one reference variable to another reference variable

- The object pointed by the reference variable left-side to the assignment operator is unused and it is eligible for garbage collection.

```
 School sc1 = new School();
 School sc2 = new School();
     sc1 = sc2;
```

3. By anonymous object

- When the reference variable is out of scope then the object referred by that reference variable is unused or referred and it is eligible for garbage collection.

How can an object be unreferenced?-----

- Java runtime system (JVM) uses these techniques and identifies the unreachable (unused) object when JVM faces the memory shortage problem.
- JVM calls garbage collection (GC) by handing over the list of unused objects.
- Hence, garbage collection is mainly responsible for cleaning memory of unused objects.
- Sometimes, some unreachable objects may refuse by GC for cleaning up memory because these objects may hold references to some resources such as JDBC database connection, IO stream connection, printer connection, network connection, etc.

How can an object be unreferenced?-----

- When we release these resources which are referred by some unused objects, we can clean the memory space by detecting those unused objects without any trouble.

How can an object be unreferenced?

- finalize() method
- The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:
- protected void finalize(){}
- Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).
- gc() method
- The gc() method is used to invoke the garbage collector to

Ways for Invoking Garbage Collector (GC)

- When the unused object becomes eligible for garbage collection, garbage collector does not destroy them immediately.
- JVM runs garbage collector whenever it runs low in memory.
- It tries its best to clean up the memory of all unused objects before it throws a java.lang.OutOfMemoryError error.
- Therefore, we can only request JVM to run garbage collector. But it has free to ignore the request.
- There are two methods for requesting JVM to run garbage

Ways for Invoking Garbage Collector (GC)------

1. Using Runtime.getRuntime().gc() method:

- Runtime class permits the program to interface with the JVM in which the program is running.
- By using its gc() method, we can request JVM to run Garbage Collector.
- Use the following source code for requesting JVM to run garbage collector.

```
// Get the Runtime object.
Runtime rt = Runtime.getRuntime();
// Call the garbage collector.
 rt.gc();
```

Ways for Invoking Garbage Collector (GC)------

2. Using System.gc() method:

- System class contains a convenience method named gc() for requesting JVM to run Garbage Collector.
- It is a static method that is equivalent to executing the Runtime.getRuntime().gc() statement.
- We can also use the following code to run the garbage collector:

```
// Invoke the garbage collector
        System.gc();
```

- The call to the gc() method of System class is also just a request to the JVM.
- The JVM is free to ignore the call.

Ways for Invoking Garbage Collector (GC)------

- Thus, there is no guarantee that any one of the above two methods will definitely run Garbage Collector by JVM.
- Let’s take a simple example program where we will use the System.gc() method.
- In this program, we will create 1,000 objects of the Object class in the createObjects() method.
- The references of new objects are not stored. So, they are garbage.
- When we will call the System.gc() method, we will try to request to the JVM to reclaim the memory used by these objects.
- We will display the memory freed by the garbage collector on

The finalize( ) Method

- Sometimes an **object** will need to **perform some action when it is destroyed.**
- **For example,** if an object is holding some **non-Java resource** such as a **file handle or window character font**, then you might want to make sure these resources are **freed before an object is destroyed**.
- To handle such situations, Java provides a mechanism called **finalization.**
- By using **finalization**, you can **define specific actions** that will **occur when an object is just about to be reclaimed by the garbage collector**.

The finalize( ) Method-------

- 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** you will specify those actions that must be **performed before an object is destroyed.**

➢ The **garbage collector runs periodically**, **checking for objects that are no longer referenced by any running state** or **indirectly through other referenced objects.**

- **Right before an asset is freed**, the **Java run time calls the finalize( ) method on the object**.

The finalize( ) Method-------

- The **finalize( ) method has the following general form**:

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

- The **keyword protected** is a **specifier that prevents access to finalize( ) by code defined outside its class**.
- The **finalize( ) method is only called just prior to garbage collection**.
- It is **not called when an object goes out-of-scope**, for example.
- This means that you **cannot know when—or even if—finalize( ) will be executed**.

The finalize( ) Method-------

- Therefore, your program **should provide other means of releasing system resources, etc., used by the object**.
- It must **not rely on finalize( ) for normal program operation**

Example

```
class Employee {
    String fName;
    String lName;
    static int count;
    public Employee(String fn, String ln){
        fName= fn;
        lName= ln;
        count++;
    }
    protected void finalize()
    {
    System.out.println("Name:"+ fName + " "+lName+ " "+count);
        count--;
    }
    }
```

```
class TestFinalize{
       public static void main(String[] args) {
      Employee e1, e2;
     e1= new  Employee("Nigussie", "Teferi");
     e2= new  Employee("Zelalem", "Getahun");
     e2= new  Employee("Zinash", "Getachew");
     //If the system.gc() method is not included, there is no
   output displayed
  //so we have to add this method to display an output
     System.gc();
         }
}
```

public static void **gc**()

Runs the garbage collector.

Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse.

When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.

The call System.gc() is effectively equivalent to the call:

Runtime.getRuntime().gc()

- If the ***system.gc() method is not included, there is no output displayed***.
- Because this ***method force the garbage collector and call the finalize method before it removes any memory references***.
- When the above program is executed, the garbage collector removes or destroy the second line of the object cod e2= new Employee (“Zelalem”, “Getahun”) because this memory reference is needed by the second parameter of e2 object.

Activity

1. Write java program to implement garbage collection. Assume the program is to remove a memory reference of one student claimed by another new incoming student

Using Objects as Parameters

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

➢ **For example:**

- Write Java program to demonstrate using objects passed to a method.
- The program **compares the invoking object with the one that it is passed. If the object** contains the same values, then the **method returns true. Otherwise, it returns false**

Using Objects as Parameters

```
//Define a class named
class PassOb {
//Define instance variables of a class
int a, b;
//Define parameterized constructor
 PassOb(int a, int b) {
     this.a=a;
     this.b=b;
    }//End of Constructor
/*Define a method, use objects as a parameter, the method return
boolean values */
```

Using Objects as Parameters----------

```
boolean equals(PassOb o) {
/*Test the condition, if the object contains the same values, with the
invoking object, the method returns true, otherwise, it returns false */
if((o.a== a) && (o.b == b)) {
    return true;
}//End of if()
else {
    return false;
}//End of else()
}//End of equals()
}//End of Class
```

Using Objects as Parameters----------

```
//Define another class to create objects of PassOb class
class TestPassOb {
public static void main(String args[]) {
//Declare, create and allocate objects named ob1 and ob2
PassOb ob1 = new PassOb(100, 22);
PassOb ob2 = new PassOb(100, 22);
PassOb ob3 = new PassOb(-1, -1);
//Call equals through println () and output the returned result
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
System.out.println("ob2 == ob3: " + ob2.equals(ob3));
    } //End of main ()
    }//End of class
```

Using Objects as Parameters----------

- The **equals( ) method** inside **PasOb class compares two objects** for equality **returns** 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 returns true. otherwise,** it **returns false.**
- Notice that the **parameter o** in **equals( ) specifies PassOb** as its type.
- Although **PassOb** is a **class type created** by the program, it is used in just the same way as **Java's built-in types.**

Using Objects as Parameters----------

- One of the most common uses of **object parameters involves constructors**.
- Frequently you will want to **construct a new object so that it is initially the same as some existing object**.
- To do this, you must define a **constructor that takes an object of its class as a parameter.**

Activity 12

- Write Java program to demonstrate one object of a class to initialize another based on the following information:

- Define a class named Box with instance variable of width, height and depth.
- Define three constructor one to pass an object of its class type as a parameter, another when all dimensions of a class are specified and the last constructor to create cube and all dimensions are equal to len when the constructor is called
- Define a method named volume(), compute volume of Box and return the result to the caller.

Activity 12-------

- Define another class named OverloadConstructor to create objects of Box type.
- Declare, create and allocate Box type objects named mybox1, mycube, mybox2 and myclone
- Pass **mycube** object through **mybox2** objects in order to one object of a class to initialize another.
- Pass **mybox1** object through **myclone** objects in order to one object of a class to initialize another.
- Call volume() through all objects of Box type and output the returned value

Activity 12-------

```
//Define a class named Box and its instance variables
class Box {
double width;
double height;
double depth;
//Define constructor that takes an object of its class as a parameter
Box(Box ob) {
width = ob.width;
height = ob.height;
depth = ob.depth;
}//End of the 1st constructor
```

Activity 12-------

```
//Define a constructor when all dimensions are specified
Box(double w, double h, double d) {
width=w;
height=h;
depth=d;
}//End of constructor
/*Define a constructor used when cube is created and the values of all dimensions are equal to len */
Box(double len) {
width = height = depth = len;
}//End of constructor
```

Activity 12-------

```
// Compute and return volume of Box
double volume() {
return (width * height * depth);
}//End of volum()
}//End of Box class
//Define another class to create objects of type Box class
class OverloadConstructor {
//main Method
public static void main(String args[]) {
//Declare, create and initializes objects using various constructors
Box mybox1 = new Box(10, 20, 15);
Box mycube = new Box(7);
```

Activity 12-------

```
//Initialize mybox2  object through mycube object
Box mybox2= new Box(mycube);
//Initialize myclone  object through mybox1 object
Box myclone = new Box(mybox1);
//Declare a variable named vol to store the returned value
double vol;
//Get volume of first box and output of  the returned result
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
//Get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
```

Activity 12-------

```
//Get volume of second box

vol = mybox2.volume();

System.out.println("Volume of mybox2 is " + vol);

//Get volume of clone

vol = myclone.volume();

System.out.println("Volume of clone is " + vol);

}//End of main ()

}//End of class
```

Activity 12-------

- Refer the above Java program and delete or comment the following constructor from the program. Run the Java program and observe and describe what you got from the change.

```
Box(Box ob) {
width = ob.width;
height = ob.height;
depth = ob.depth;
}
```

Argument Passing to a Method

- In general, there are **two ways** that a **computer language** can **pass** an **argument** to a **subroutine** or a **method**.

- **call-by-value** and
- **call-by-reference**

i) Call-by-value

- This method copies the **value** of an **argument** in to the **formal parameter** of the **method**.
- Therefore, **changes** made to the **parameter** of the method have **no effect** on the **argument passed**.

ii) Call-by-reference.

- In this method, a **reference** to an **argument** (not the **value** of the **argument**) is **passed** to the **parameter**.

Argument Passing to a Method

- Inside the method, this reference is used to **access** the **actual argument** specified in the call.
- This means that **changes made** to the **parameter** will **affect** the **argument** used to call the **method**.
- As you will see, Java uses both **approaches**, depending upon what is passed.

➢ In Java, when you pass a **simple type** to a **method**, it is **passed** by **value**.

- Thus, what occurs to the **parameter** that **receives** the **argument** has **no effect outside** the method.

Activity 13

- Write two separate Java program to demonstrate argument passed by value and argument passed by reference respectively. Run both program and observe the output and describe shorty what you observed from the output of the program.

/*Java Simple types program to demonstrate argument passed by value*/

```
//Define a class named CallByValue
class CallByValue {
 /*Define Parameterized method named M1(), compute (i*2) and
(j/2) but this method does not affect the values passed to this
method*/
```

Activity 13--------

```
void m1(int i, int j) {
i=(i*2);
j=(j/= 2);
}//End of meth()
}//End of class
//Define a class to create objects of type CallByValue
class TestByValue{
//main method ()
public static void main(String args[]) {
CallByValue  ob = new CallByValue ();
//Declare variables local to main named a and b
int a, b;
```

Activity 13--------

```
//Initialize a to 15 and b to 20 respectively
a = 15;
b = 20;
//Output the values of a and b before call
System.out.println("a and b before call: " +a + " " + b);
//Call a m1() and pass the value a and b and output
ob.m1(a, b);
System.out.println("a and b after call: " +a + " " + b);
}//End of main ()
}//End of class
```

Activity 13---

The output from this program is shown here:

a and b before call: 15 20

a and b after call: 15 20

- As you can see, the **operations** that occur inside m1( ) have **no effect** on the **values** of **a** and **b** used in the **call**; their values here did **not change** to **30** and **10**.
- When you pass an **object** to a **method**, the situation changes dramatically, because **objects** are **passed** by **reference**.

Activity 13---

- Keep in mind that when you create a **variable** of a **class** type, you are only creating a **reference** to an **object**.
- Thus, when you **pass** this **reference** to a **method**, the **parameter** that **receives** it will refer to the **same object** as that referred to by the argument.
- This effectively means that **objects** are **passed** to methods by use of **call-by-reference.**
- **Changes** to the **object inside** the method do **affect** the **object** used as an **argument.**

Activity 13---

- Consider the following java program o**bjects are passed by reference.**

```
 //Define a class named CallByRef
 class CallByRef {
 //Declaration of instance variables of a class
  int a, b;
 //Define parameterized Constructor
CallByRef(int i, int j) {
a = i;
b = j;
} //End of Constructor
```

Activity 13---

```
//Define a method named m1(), objects as a parameter type
void m1(CallByRef o) {
o.a=((o.a)*2);
o.b=((o.b)/2);
}//End of m1()
}//End of class
//Define a class to create objects of class type CallByRef
class TestRef {
public static void main(String args[]) {
 //Declare, create and initialize objects
CallByRef ob = new CallByRef (15, 20);
```

Activity 13---

```
//Output of a and b before call
System.out.println("ob.a and ob.b before call: " +ob.a + " " +ob.b);
/*Call m1() and output the a and be, this call changes the value of
  a and b, as the call is through reference */
ob.m1(ob);
System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);
}//End of main()
}//End of class
```

Activity 13---

- As you can see, in this case, the actions inside meth( ) have affected the object used as an argument.
- As a point of interest, when an object reference is passed to a method, the reference itself is passed by use of call-by-value.
- However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

➢ When a simple type is passed to a method, it is done by use of call-by-value.

- Objects are passed by use of call-by-reference.

Returning Objects--------

- A **method** can **return any type of data**, including **class types that you create**.
- For example, in the following program, the **incrByTen( ) method** returns an **object** in which the value of **a** is ten greater than it is in the **invoking object**.

```
/* Java Program to define a method to return an object of a class
            to the caller */
//Define a class named TestObj
class TestObj {
//Declare instance variable named a
int a;
```

Returning Objects--------

```
 //Define Parameterized Constructor
 TestObj( int  i) {
 a = i;
 }//End of constructor
 //Define a method that return an object of class type
 TestObj incrByTen() {
TestObj temp = new TestObj (a+10);
return temp;
}//End of incrByTen()
}//End of class
```

Returning Objects--------

```
 //Define Parameterized Constructor
 TestObj( int  i) {
 a = i;
 }//End of constructor
 //Define a method that return an object of class type
 TestObj incrByTen() {
TestObj temp = new TestObj (a+10);
return temp;
}//End of incrByTen()
}//End of class
```

Returning Objects--------

```
//Define another class to create objects of a class
class RetOb {
//main Method()
public static void main(String args[]) {
//Declare, Create and initialize objects
TestObj ob1 = new TestObj(2);
//Declare another Reference variable of a class
TestObj ob2;
//Call incByTen() and assign the return value to ob2
ob2 = ob1.incrByTen();
//Output a value before the method return the object
```

Returning Objects--------

```
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase:" + ob2.a);
}//End of main ()
}//End of class
```

- ➢ The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

Returning Objects--------

- As you can see, each time incrByTen( ) is invoked, a new object is created, and a reference to it is returned to the calling routine.
- The preceding program makes another important point:
- Since all **objects** are **dynamically allocated** using new, you don’t need to worry about an **object going out-of-scope** because the method in which it was created terminates.
- The object will continue to exist as long as there is a reference to it somewhere in your program.
- When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

Understanding static

- Java allows to **define** a **class** **member** that will be used **independently** of any **object** of that **class**.
  - Normally a **class member** must be accessed only in conjunction with an **object of its class**.
  - However, it is possible to create a **member** that can be used by **itself**, without **reference to a specific instance** **or object**.
  - To create such a **member**, precede its **declaration** with the **keyword** **static.**
- When a **member is declared static**, it can be accessed before any **objects of its class are created**, and without **reference to any object**.

Understanding static------

- You can declare both **methods and variables to be static.**
- The most common example of a **static member** is **main( ).**

✓ main( ) is declared as **static** because it must be called before any **objects exist**.

- **Instance variables** declared as **static** are, essentially, **global variables.**

➢ When **objects** of its class are declared, **no copy of a static variable** is made.

- Instead, all **instances** of the class **share the same static variable.**

Understanding static------

- **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** in any way.
- If you need to do computation in order to **initialize** your **static variables**, you can declare a **static block** which gets **executed exactly once**, when the **class is first loaded**.
- The **following example** shows a class that has a **static method, some static variables, and a static initialization block:**

Understanding static------

```
//Define a class named UseStatic
class UseStatic {
//Declaration of Static variables
static int a = 3;
static int b;
//Define static method
static void meth(int x) {
//Output values of instance variables when a method is called
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}//End of  static method
```

Understanding static------

```
/*If you need to do computation in order to initialize your
      static variables, you can declare a static block which gets
executed exactly once, when the class is first loaded */
static {
System.out.println("Static block initialized.");
b = a * 4;
}//End of static block
public static void main(String args[]) {
meth(42);
}//End of main ()
}//End of class UseStatic
```

- As soon as the ***UseStatic*** class is ***loaded***, all of the ***static statements*** are run.
- First, ***a is set to 3***,
- Then the **static block executes** (printing a message),

Understanding static------

- And finally, **b is initialized to a * 4 or 12**.
- Then main( ) is called, which calls meth( ), passing 42 to x.
- The **three println( ) statements** refer to the **two static variables** **a** and **b**, as well as to the **local variable x**.
- It is **illegal to refer** to any **instance variables inside of a static method**.
- Output of the previous program is as follows:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Understanding 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**.
- For example, if you wish to **call a static method** from **outside its class**, you can do so using the following general form: **classname.method( )**
- Here, **classname** is the **name of the class** in which the **static method** is declared.

Understanding static------

- As you can see, this format is similar to that used to **call non-static methods** through **object- reference variables**.
- A **static variable** can be accessed in the same way—by use of the **dot operator on the name of the class**.
- This is how **Java implements** a **controlled** version of **global methods** and **global variables**.

➢ Here is an example.

- **Inside main( ),** the **static method callme( )** and the **static variable b** are **accessed outside of their class**.

```
//Define a class named SaticDemo
public class StaticDemo {
//Declaration of Static variables and initialize with some value
static int a = 42;
```

Understanding static------

```
 static int b = 99;
 //Define a static method
 static void callme() {
//Display the values of static variable a
System.out.println("a = " + a);
}//End of static method
}//End of class
//Define another class to create objects of StaticDemo class
class StaticByName {
//main Method
```

Understanding static------

```
public static void main(String args[]) {
//Acces a static method outside of a class
//using classname.methodname()
StaticDemo.callme();
//Access static variables through classname.static var
System.out.println("b = " + StaticDemo.b);
}//End of main ()
}//End of class
```

Introducing final

- A variable can be declared as final.
- Doing so prevents its contents from being modified.
- This means that you must initialize a final variable when it is declared.
- (In this usage, final is similar to const in C/C++/C#.) For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Introducing final

- Subsequent parts of your program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed.
- It is a common coding convention to choose all uppercase identifiers for final variables.
- Variables declared as final do not occupy memory on a per-instance basis.
- Thus, a final variable is essentially a constant.
- The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

Activity 14

```
//Program to implement final key word to calculate values to be
//Placed into elements of an array
class FinalKeyWordExample {
public static void main( String args[] ){
final int ARRAY_LENGTH=10;//declare constants
int array[]=new int[ARRAY_LENGTH];//create array
// calculate value for each array element
for( int counter = 0; counter < array.length; counter++ )
    array[counter] = 2 + 2 * counter;
    System.out.println("Index"+ " "+"Value"); // column headings
    // output each array element's value
```

Activity 14---------

```
for ( int counter = 0; counter < array.length; counter++ )
   System.out.println(counter+ "\t" + array[ counter ] );
} // end main
} // end class FinalKeyWordExample
```