Chapter 6- Introducing Classes

The most important thing to understand about a class is that it defines a new data type. Thus, a class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

A class is declared by use of the class keyword. x. A simplified general form of a class definition is shown here:

```
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
   }
}
```

The data, or 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.

A Simple Class:

```
class Box {
  double width;
  double height;
  double depth;
}
```

As stated, a class defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will refer to an instance of **Box**.

As mentioned earlier, 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 variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the width variable of mybox the value 100, you would use the following statement mybox.width=100;

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
  double width;
  double height;
  double depth;
}

// This class declares an object of type Box.
class BoxDemo {
  public static void main(String[] args) {
    Box mybox = new Box();
    double vol;

    // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;

    System.out.println("Volume is " + vol);
  }
}
```

Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. First, you must 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. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the 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, essentially, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure. In the preceding sample programs, a line similar to the following is usedto declare an object of type Box:

Box mybox = new Box();

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

Box mybox; // declare reference to object

mybox = new Box(); // allocate a Box object

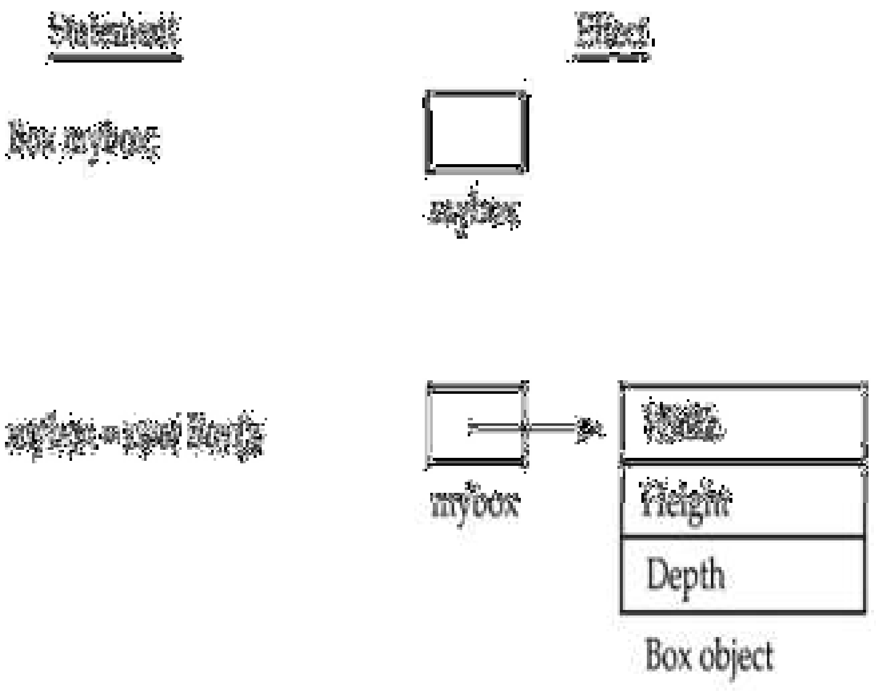


Figure 6-1 Declaring an object of type Box

Assigning Object Reference Variables

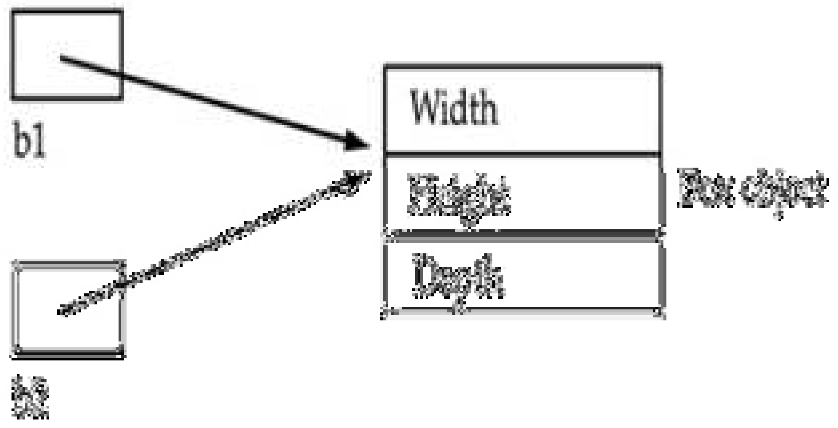
Object reference variables act differently than you might expect when an Assignment takes place. For example,

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. Her both b1 and b2 refer to the same object.

This situation is depicted here:



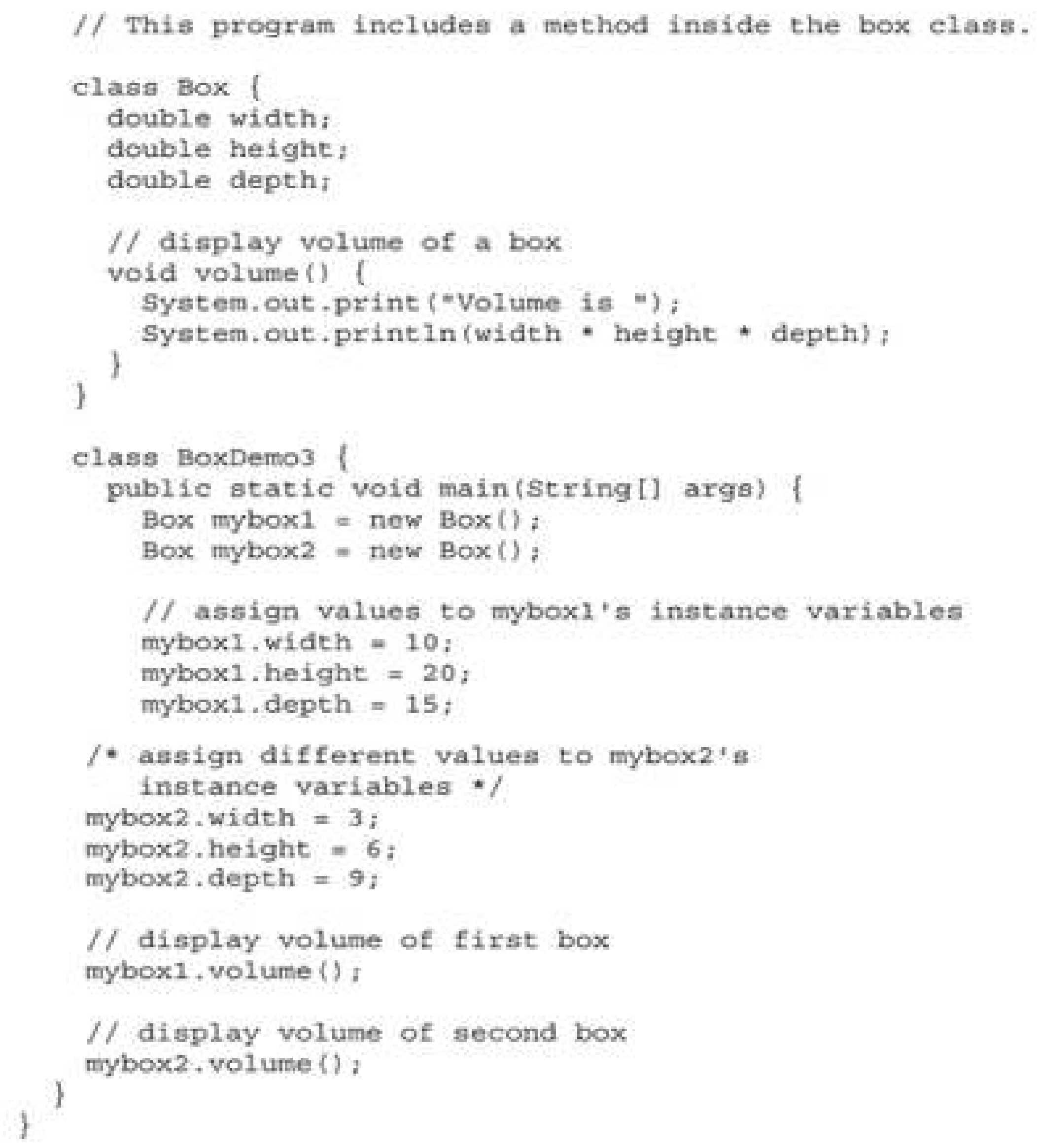
Introducing Methods

Classes usually consist of two things: instance variables and methods.

This is the general form of a method:

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

Adding a Method to the Box Class



```
// This program includes a method inside the box class.

class Box {
  double width;
  double height;
  double depth;

  // display volume of a box
  void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
  }
}

class BoxDemo3 {
  public static void main(String[] args) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

   /* assign different values to mybox2's
      instance variables */
   mybox2.width = 3;
   mybox2.height = 6;
   mybox2.depth = 9;

   // display volume of first box
   mybox1.volume();

   // display volume of second box
   mybox2.volume();
  }
}
```

Returning a Value

While the implementation of volume ( ) 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 example, an improved version of the preceding program, does just that.

```
// Now, volume() returns the volume of a box.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo4 {
  public static void main(String[] args) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10: 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 useful. 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.

Here is an example:

int x, y;

x = square(5); // x equals 25

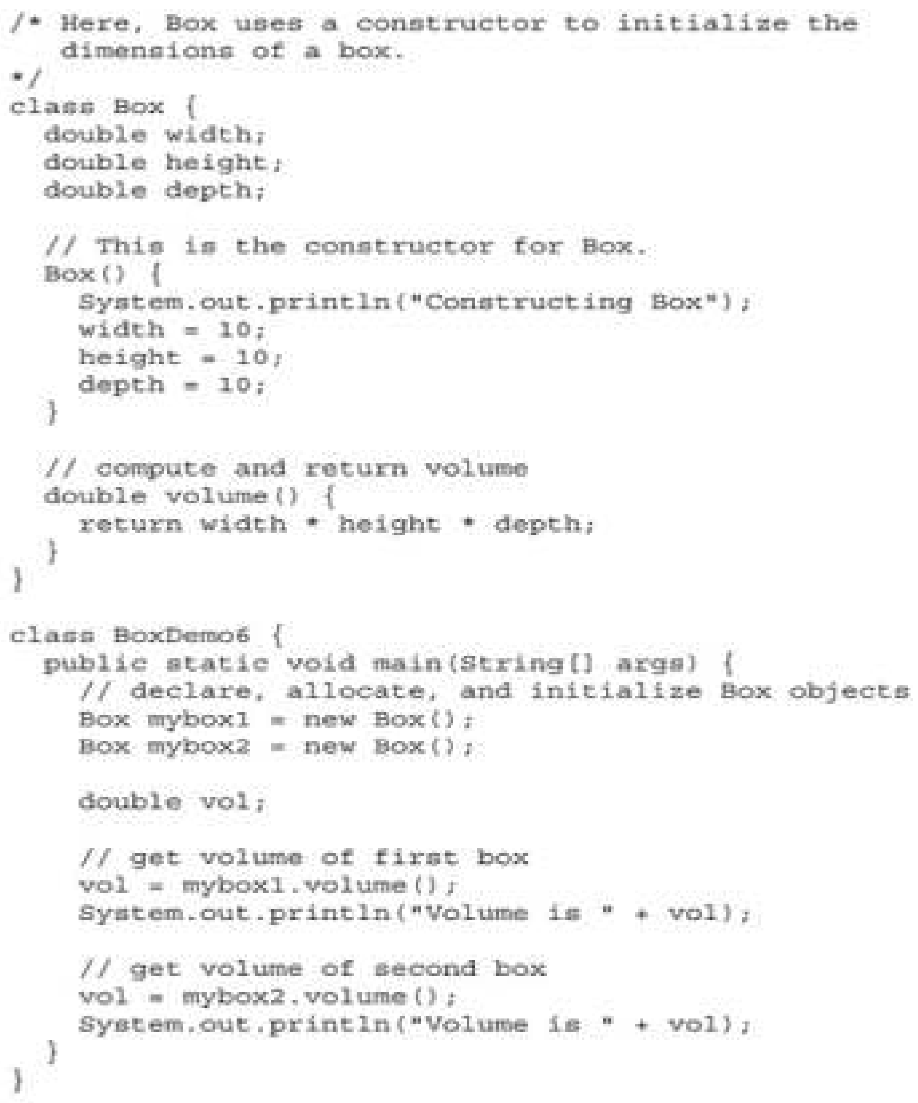
x = square(9); // x equals 81

y = 2;

x = square(y); // x equals 4

Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. 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. Once defined, the constructor is automatically called when the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void.



```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo6 {
  public static void main(String[] args) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

Parameterized Constructors

While the Box( ) constructor in the preceding example 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 that sets the dimensions of a box as specified by those parameters. Pay special attention to how Box objects are created.

```
  /* Here, Box uses a parameterized constructor to
     initialize the dimensions of a box.
  */
  class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box(double w, double h, double d) {
      width = w;
      height = h;
      depth = d;
    }

    // compute and return volume
    double volume() {
    return width * height * depth;
  }
}

class BoxDemo7 {
  public static void main(String[] args) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

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( )**:

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

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.

A Stack Class

Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values
class Stack {
  int[] stck = new int[10];
  int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

MODULE 2
Chapter 7

Overloading Methods

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 supports 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. 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.

1.Overloading by changing the number of parameters.

```
class MethodOverloading {
    private static void display(int a){
        System.out.println("Arguments: " + a);
    }

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

    public static void main(String[] args) {
        display(1);
        display(1, 4);
    }
}
```

2. Method Overloading by changing the data type of parameters

```
class MethodOverloading {
    // this method accepts int
    private static void display(int a){
        System.out.println("Got Integer data.");
    }

    // this method  accepts String object
    private static void display(String a){
        System.out.println("Got String object.");
    }

    public static void main(String[] args) {
        display(1);
        display("Hello");
    }
}
```

Overloading Constructors

Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading.

An example class to understand need of constructor overloading.

```
class Box

{


 double width, height,depth;

   Box(double w, double h, double d)

   {

      width = w;

      height = h;

      depth = d;

   }

   double volume()

   {

      return width * height * depth;

   }

}
```

Argument Passing:

2 ways of passing arguments:

1.Call-by-value: This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

```
public class CallByValueExample {

   public static void main(String[] args) {
      int num = 5;

      System.out.println("Before method call: num = " + num);
      incrementByValue(num);
      System.out.println("After method call: num = " + num);
   }

   public static void increment ByValue(int x) {
      x += 1;
      System.out.println("Inside method: x = " + x);
   }
}
```

2.Call-by-reference: In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

```
public class CallByReferenceExample {

   public static void main(String[] args) {
      MyClass obj = new MyClass(5);

      System.out.println("Before method call: obj.value = " + obj.getValue());
      incrementByReference(obj);
      System.out.println("After method call: obj.value = " + obj.getValue());
   }

   public static void incrementByReference(MyClass myObj) {
      myObj.setValue(myObj.getValue() + 1);
```

```
        System.out.println("Inside method: obj.value = " + myObj.getValue());
    }
}

class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Returning Objects

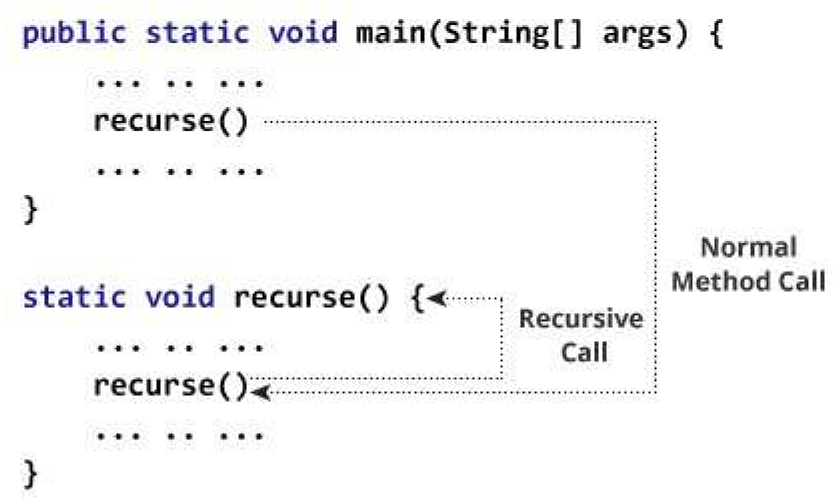
A method can return any type of data, including class types that you create.

```
public class ObjectReturnExample {

    public static void main(String[] args) {
        // Creating an object using the createPerson method
        Person person1 = createPerson("John", 25);

        // Displaying information about the returned object
        System.out.println("Person 1: " + person1.getName() + ", Age: " +
person1.getAge());

        // Modifying the returned object
        modifyPerson(person1, "Jane", 30);

        // Displaying information about the modified object
        System.out.println("Person 1 (after modification): " + person1.getName() +
", Age: " + person1.getAge());
    }

    // Method to create and return a Person object
    public static Person createPerson(String name, int age) {
        return new Person(name, age);
    }

    // Method to modify the attributes of a Person object
    public static void modifyPerson(Person person, String newName, int newAge)
{
        person.setName(newName);
        person.setAge(newAge);
    }
}

// A simple Person class
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Recursion

Java supports *recursion*. Recursion is the process of defining something in terms of itself. Recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

How Recursion works?



```
public static void main(String[] args) {
    ... .. ...
    recurse()
    ... .. ...
}

static void recurse() {
    ... .. ...
    recurse()
    ... .. ...
}
```

In the above example, we have called the recurse() method from inside the main method. (normal method call). And, inside the recurse() method, we are again calling the same recurse method. This is a recursive call.

In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely. Hence, we use the if...else statement (or similar approach) to terminate the recursive call inside the method.

```
class Factorial {

    static int factorial( int n ) {
        if (n != 0)  // termination condition
            return n * factorial(n-1); // recursive call
        else
            return 1;
    }

    public static void main(String[] args) {
        int number = 4, result;
        result = factorial(number);
        System.out.println(number + " factorial = " + result);
    }
}
```

Introducing Access Control

Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: ***access control***. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data.

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. Protected applies only when inheritance is involved.

When a member of a class is modified by **public**, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.

Java program to showcase the example of public access modifier

```
import required packages
import java.io.*;
import java.util.*;

// declaring a public class
public class A {

   // declaring method m1
   public void m1() { System.out.println("GFG"); }
}

class B {

   // main method
   public static void main(String[] args)
   {
      // creating an object of type class A
      A a = new A();

      // accessing the method m1()
      a.m1();
   }
}
```

Java program to showcase the example of private access modifier

```
// import required packages
```

```
import java.io.*;

import java.util.*;

// helper class
class A {

    // helper method
    private void m1() { System.out.println("GFG"); }
}

// driver class
class B {

    // main method
    public static void main(String[] args)
    {
        // creating an object of type class A
        A a = new A();

        // accessing the method m1()
        a.m1();
    }
}
```

Understanding static

when you want 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. 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. 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.

When objects of its class are declared, no copy of a **static** variable is Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

• They can only directly call other **static** methods of their class.
• They can only directly access **static** variables of their class.
• 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 that gets executed exactly once, when the class is first loaded.

Introducing final

A field can be declared as **final**. This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is probably the most common. Here is an 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;
```

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** fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared **final**. Declaring a parameter **final** prevents it from being changed within the method. Declaring a local variable **final** prevents it from being assigned a value more than once.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: ***static*** and ***non-static***. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly.

The second type of nested class is the ***inner*** **class**. An inner class is a nonstatic nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non static members of the outer class do.

Static Inner Class

```
class MotherBoard {

   // static nested class
   static class USB{
       int usb2 = 2;
       int usb3 = 1;
       int getTotalPorts(){
           return usb2 + usb3;
       }
   }


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

       // create an object of the static nested class
       // using the name of the outer class
       MotherBoard.USB usb = new MotherBoard.USB();
System.out.println("Total Ports = " + usb.getTotalPorts());
   }
}
```

Accessing members of Outer class inside Static Inner Class

```
class MotherBoard {
   String model;
   public MotherBoard(String model) {
       this.model = model;
   }

   // static nested class
   static class USB{
       int usb2 = 2;
       int usb3 = 1;
       int getTotalPorts(){
           // accessing the variable model of the outer classs
           if(MotherBoard.this.model.equals("MSI")) {
               return 4;
           }
           else {
               return usb2 + usb3;
           }
```

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

        // create an object of the static nested class
        MotherBoard.USB usb = new MotherBoard.USB();
        System.out.println("Total Ports = " + usb.getTotalPorts());
    }
}
```

QUESTION BANK MODULE-2

1. Write the general form of a class in Java. Explain class definition and member access with suitable program.
2. Explain two step procedure to create objects (class instance) with suitable program.
3. Explain assignment of objects with suitable example.
4. Explain general form of method definition in Java. Explain method definitions with suitable programs for:
    a. Methods with no parameters and no return value
    b. Methods with no parameters and return value
    c. Methods with parameters and return value
5. Explain the role of constructors with an example.
6. Explain with a program (a) default constructor (b) parameterized constructor
7. Explain the role of 'this' in parameter and instance variable name collision handling with suitable programming example.
8. Develop a Java program to define a class '2DBox' with data members 'length and width. Define a class member methods to calculate the area of 2DBox and print box dimensions. Develop main method to illustrate the creation of objects, member access and use of methods suitably.
9. Develop a Java program to define a class 'Customer' with data members like Name, Employment and Age. Define a constructor and methods to change employment and age. Develop main method to illustrate object creation, member access and invocation of methods suitably.
10. Explain Java support for compile time polymorphism by taking suitable programming example.
11. With a program, explain the effect of automatic type conversion in method overloading.
12. Explain constructor overloading with suitable class definition and program.
13. Explain with suitable program:

    a) Objects as method parameters

    b) Objects as constructor parameters
14. Explain passing of primitive types and passing of objects as arguments to methods with suitable programs.
15. Explain returning of objects from methods with a program.
16. Explain how recursion is implemented win Java? What are the benefits of recursion in programming?
17. Explain with a program, Java support for encapsulation through visibility modifiers.
18. Explain the different roles of keyword 'static' with programs.
19. Explain different roles of keyword 'final' with programs.
20. Explain the role of static nested classes and inner classes with programs.
21. Develop a recursive version of generating, storing and display of Fibonacci series (Hint: use array).