UNIT-2

I. Program Structure in Java:

1. Classes and Objects: Introduction
2. Class Declaration and Modifiers
3. Class Members
4. Declaration of Class Objects
5. Assigning One Object to Another
6. Access Control for Class Members
7. Accessing Private Members of Class
8. Constructor Methods for Class
9. Overloaded Constructor Methods
10. Nested Classes
11. Final Class and Methods
12. Passing Arguments by Value and by Reference
13. Keyword this.

II. Methods:

1. Introduction
2. Defining Methods
3. Overloaded Methods
4. Overloaded ConstructorMethods
5. Class Objects as Parameters in Methods
6. Access Control
7. Recursive Methods
8. Nesting of Methods
9. Overriding Methods
10. Attributes Final and Static.

I. Program Structure in Java:

1. Classes and Objects: Introduction

Classes:

- A class is defined as collection of similar objects.
- Class is a blueprint or template for creating objects.
- Classes are user-defined data types and behave like the built-in types of a programming language.
- In the real-world, classes are invisible only objects are visible.

Example:

- Car is an object representing a class called Vehicle.
- We can see the object called care but we cannot see the class called Vehicle.

Syntax:

Vehicle car;

- It will create an object car belonging to the class Vehicle.

Objects:

- Objects are basic run-time entities in an object-oriented system.

(or)

Any real world entity is called an object.

(or)

Objects are the combination of data and methods.

Example: Person, Place, bank account, …., so on.

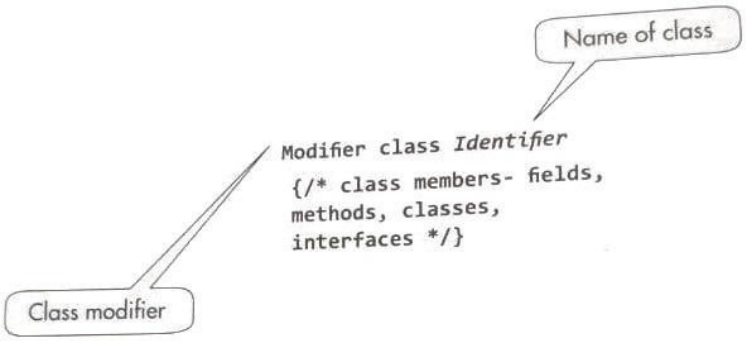
- In the real-world only objects are visible but classes are invisible.
- The most important benefits of an objects are
  - Modularity
  - Reusability
- The properties of objects are two types
  - visible
  - invisible
- Let man is an object, then visible properties are eyes, ears, hands, legs,…so on and invisible properties are name, blood group,…. so on.
- Every object contains three basic elements
  - Identity (name)
  - State (variables)
  - Behavior (methods)

Object = Data + Methods

2. Class Declaration and Modifiers

A class declaration starts with the Access modifier. It is followed by keyword class, which is followed by the name or identifier. The body of class is enclosed between a pair of braces { }.

Syntax:



Example:

```
public class MyFarm
{ double length;      // instance variable
double width;         // instance variable
}
```

- The class name starts with an upper-case letter, whereas variable names may start with lower-case letters.
- In the case of names consisting of two or more words as in MyFarm, the other words for with a capital letter for both classes and variables. In multiword identifiers, there is no blank space between the words
- The class names should be simple and descriptive.
- Class names should start with an upper-case letter and should be nouns. For example, it could include names such as vehicles, books, and symbols.
- It should have both upper and lower-case letters with the first letter capitalized
- Acronyms and abbreviations should be avoided

Class modifiers:

- Class modifiers are used to control the access to class and its inheritance characteristics.
- Java consists of packages and the packages consist of sub-packages and classes Packages can also be used to control the accessibility of a class
- These modifiers can be grouped as (a) access modifiers and (b) non-access modifiers. Table 5.l gives a description of the various class modifiers.

Modifier	Type	Description
(No modifier)	Access Modifier	The class is package-private (accessible only within its own package).
public	Access Modifier	The class is accessible from any other class in all packages.
—	—	—
final	Non-Access Modifier	The class cannot be extended (no subclass can be created).
abstract	Non-Access Modifier	The class cannot be instantiated directly; may contain abstract methods.
strictfp	Non-Access Modifier	Ensures consistent floating-point calculations across all platforms
private	Not allowed for top-level classes	Only applicable to **nested classes**; accessible only within the enclosing class.
protected	Not allowed for top-level classes	Only applicable to **nested classes/members**, not top-level classes.

Examples:

1. A class **without modifier.**

```
class Student
{
      /* class body*/
}
```

2. A class with modifier

```
public class Student
{
      /* class body*/
}
```

(or)

```
private class Student
{
      /* class body*/
}
```

(or)

```
protected class Student
{
      /* class body*/
}
```

(or)

```
final class Student
{
      /* class body*/
}
```

(or)

```
abstract class Student
{
      /* class body*/
}
```

3. Class Members

The class members are declared in the body of a class. These may comprise fields (variables in a class). methods, nested classes, and interfaces. The members of a class comprise the members declared in the class as well as the members inherited from a super class. The scope of all the members extends to the entire class body.

The fields comprise two types of variables

1. **Non Static variables :** These include *instance* and *local variables* and vanes in scope and value.

   (a) Instance variables: These variables are individual to an object and an object keeps a copy of these variables in its memory.

   (b) Local variables: These are local in scope and not accessible outside their scope.

2. **Class variables (Static Variables) :** These variables are also qualified as static variables. The values of these variables are common to all the objects of the class. **The class keeps only one copy of these variables and all the objects share the same copy.** As class variables belong to the whole class, these are also called class variables.

Example:

```
class CustomerId
{
      static int  count=0; // static variable
      int id; // instance variable
      CustomerId() // Constructor
      {
          count++;
          id = count ;
      }
      int getId() // Method
      {
          return id;
      }
      int localVar()
      {
        int a=10; //Local variable

         return a;
      }
}

class Application
{
   public static void main(String[] args)
   {
      CustomerId obj = new CustomerId();
      System.out.println("Customer Id = " + obj.getId());
      System.out.println("Local Variable = " + obj.localVar());

   }
}
```

Output:

```
C:\>javac Application.java

C:\>java Application
Customer Id = 1
Local Variable = 10
```

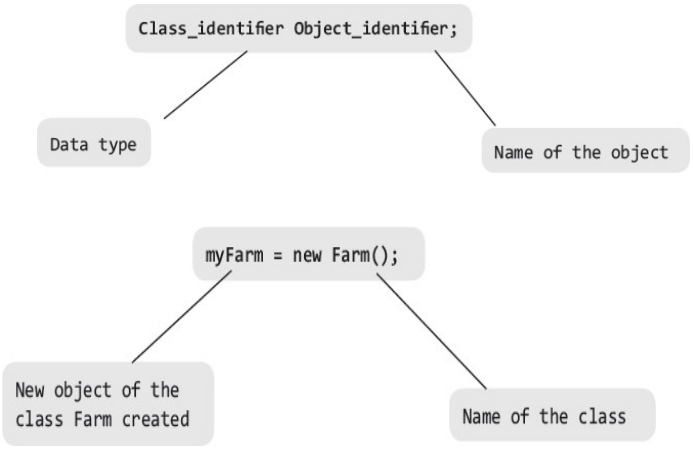
4. Declaration of Class Objects

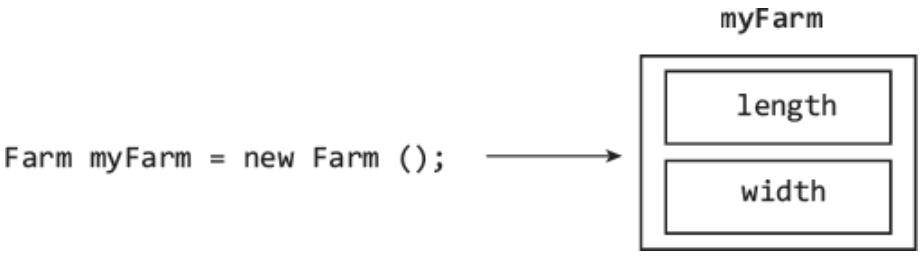
Creating an object is also referred to as instantiating an object.

- Objects in java are created dynamically using the new operator.
- The new operator creates an object of the specified class and returns a reference to that object.

Syntax: (creating an object)

className objectReference=new className();





Example:

```
Farm myFarm = new Farm();
```

5. Assigning One Object to Another

Java provides the facility to assign one object to another object

Syntax:

new_Object = old_object;

all the properties of old_object will be copied to new object.

Example:

```
class Farm
{
   double length;
   double width;
   double area()
   {
            return length*width;
   }
}
public class FarmExel
{
   public static void main (String args[])
   {
     Farm farm1 = new Farm(); //defining an object of Farm
     Farm farm2 = new Farm(); //defining new object of Farm

     farm1.length = 20.0;
     farm1.width = 40.0;

     System.out.println("Area of form1= " + farm1.area());

     farm2 = farm1;  // Object Assignment

     System.out.println("Area of form2 = " + farm2.area());

   }
}
```

Output:

```
C:\>javac FarmExel.java

C:\>java FarmExel
Area of form1= 800.0
Area of form2 = 800.0
```

6. Access Control for Class Members

In Java, There are three access specifiers are permitted:

- public
- protected
- private

The coding with access specifiers for variables is illustrated as

```
Access_specifier type identifier;
```

Detials of Access specifiers are as follows.

Table 5.2 Access specifiers and their effect

Access specifier	*Access*
No access specifier	Access is permitted to any other class belonging to the same package.
public	Access is permitted to any other class in any Java package.
protected	Access is permitted from subclass of this class in any package and to any class in the same package. The access is not permitted to any other class in another package.
private	Access is permitted only to members of the same class.

7. Accessing Private Members of Class

- Private members of a class, whether they are instance variables or methods, can only be accessed by other members of the same class
- Any outside code cannot directly access them because they are private. However, interface public method members may be defined to access the private members
- The code other than class members can access the interface public members that pass on the values.

Example:

```
public class Farm
{
      private double length;  // private member data
      private double width;  // private member data

      //definition of public methods
      public double area() {return length*width;}
      public void setSides(double l, double w)
                  { length=l; width = w;      }
      public double getLength(){return length;}
      public double getWidth(){return width;}
}
```

```
class FarmExe3
{
      public static void main (String args[])
      {

      Farm farm1 =new Farm();
      double farmArea;

      farm1.setSides(50.0,20.0);
      farmArea = farm1.area();

      System.out.println("Area of farm1 = "+ farmArea);
      System.out.println("Length of farm1 = "+ farm1.getLength());
      System.out.println("Width of farm1 = "+ farm1.getWidth());

      }
}
```

Output

```
C:\>javac PrivateMembers.java

C:\>java FarmExe3
Area of farm1 = 1000.0
Length of farm1 = 50.0
Width of farm1 = 20.0
```

In the above program, the two object variables `length` and `width` are declared private. The first thing is to assign values to these variables for an object. This is done by defining a public method `setSides()`, which is invoked by the class object for entering values that are passed to length and width variables The method setsides may be defined as

```
public void setsides (int 1, int w){length = 1; width = w;}
```

The class also defines another method `area()` to which the values are passed for calculation of area when the method area() is invoked by the object. For obtaining values of **`length`** and **`width`** by outside code, two public methods are defined as

```
public double getLength(){return length;} //Function for getting length
public double getWidth(){return width;}   // Function for getting width
```

These methods may be invoked by objects of the class to obtain the values of variables as follows:

```
farm1.getLength()

farm1.getWidth()
```

8. Constructor Methods for Class

- A constructor is a special method of the class and it is used to initialize an object whenever the object is created.
- A Constructor is a special method because,
  - Class name and Constructor name both must be same
  - Doesn't contain any return type
  - Automatically executed when object is created.
- Constructors are two types
  i. Default Constructor (without arguments)
  ii. Parameterized Constructor (with arguments)

Example:

```
class Perimeter
 {
  Perimeter()  // default Constructor
     {
       System.out.println("No parameters");
      }
   Perimeter(double r)  // Parameterized Constructor
    {
      System.out.println("Perimeter of the Circle="+(2*3.14*r));
    }
  Perimeter(int l, int b) // Parameterized constructor
    {
      System.out.println("Perimeter of the Rectangle="+(2*(l+b)));
    }
 }
class ConstructorDemo
 {
   public static void main(String args[])
    {
      Perimeter p1=new Perimeter();
      Perimeter p2=new Perimeter(10);
      Perimeter p3=new Perimeter(10,20);
    }
 }
Output
   E:\>javac ConstructorDemo.java

   E:\>java ConstructorDemo
   No parameters
   Perimeter of the Circle=62.800000000000004
   Perimeter of the Rectangle=60
```

9. Overloaded Constructor Methods

Like other methods, the constructors may also be overloaded. **The name of all the overloaded constructor methods same as the name of the class, but parameters have to be different either in number of type a order of parameters in each definition**.

Example:

```
class Perimeter
 {
  Perimeter()
     {
       System.out.println("No parameters");
      }
   Perimeter(double r)  //Constructor Overloading
    {
      System.out.println("Perimeter of the Circle="+(2*3.14*r));
    }
  Perimeter(int l, int b) // Constructor Overloading
    {
      System.out.println("Perimeter of the Rectangle="+(2*(l+b)));
    }
 }
class ConstructorDemo
 {
   public static void main(String args[])
    {
      Perimeter p1=new Perimeter();
      Perimeter p2=new Perimeter(10);
      Perimeter p3=new Perimeter(10,20);
    }
 }
Output
   C:\>javac ConstructorDemo.java

   C:\>java ConstructorDemo
   No parameters
   Perimeter of the Circle=62.800000000000004
   Perimeter of the Rectangle=60
```

10. Nested Classes

A nested class is one that is declared entirely in the body of another class or interface. The class, which is nested, exists only long as the enveloping class exists. Therefore, the scope of inner class is limited to the scope of enveloping class. There are four types of nested class.

Nested static class is like any other static member of the enveloping class.

i. Member Inner Class.
ii. Anonymous Class
iii. Local Class
iv. Static Nested Class

i. Member Inner Class.

A class which is declared within class is called Member inner class.

The inner class has access to all the members of the enveloping class including the members declared public, protected or private.

Example:

```
class Outer
{
      double outer_x;
      double outer_y;
      Outer (double a, double b)
      {
         outer_x = a;
         outer_y = b;
      }
      double outer_add()
      {
         return outer_x+outer_y;
      }
         void outer_display()
      {
      Inner in = new Inner();
      in.inner_display();
      }

      class Inner  // Inner Class
      {
            void inner_display()
            {
               System.out.println("x+y = "+ outer_add());
            }
      }
}
```

```
class NestedClassDemo
{
      public static void main (String args[])
      {
            Outer obj =new Outer(10,20);
            obj.outer_display();
      }
}
```

Output:

```
C:\>javac NestedClassDemo.java

C:\>java NestedClassDemo
x+y = 30.0
```

ii. Anonymous Class

- **Anonymous classes are inner classes without a name.**
- It is defined inside another class. Because class has no name it cannot have a constructor method and its objects cannot be declared outside the class.
- Therefore, an anonymous class must be defined and initialized in a single expression.
- An anonymous class may be used where the class has to be used only once.
- An anonymous class extends a super class or implements an interface, but keywords extend or implements do not appear in its definition. On the other hand, the names of super class and interface do appear.
- An anonymous class is defined by operator new followed by class name it extends, argument list for the constructor of super class, and then the anonymous class body.

Example:

```
abstract class Person
{
   abstract void display(); //abstract method
}

class AnonymousClass
{
   public static void main (String args[])
   {
    Person obj = new Person() { // Creating an object of Anonymous class
    void display()
    {
     System.out.println("In display() method ");
    }
   }; // anonymous class closes

  obj.display(); // Calling anonymous class method
 }
}
```

Output:

```
– – – –
C:\>javac AnonymousClass.java

C:\>java AnonymousClass
In display() method
```

iii. Local Class

- **A local class is declared in a block or a method**, and hence, their scope is limited to the block of method. The general properties of such classes are as follows
    - These classes can refer to local variables or parameters, which are declared final
    - These are not visible outside the block in which they are declared and hence, the access modifiers such as public, private, or protected do not apply to local classes.

Example:

Example:

```
class LocalClassDemo
{
  public static void main (String args[])
  {
      class Local     // Local class defined
      {
            int x;
            Local(int a) { x =a; }
            public  void display()
            {
              System.out.println("x = "+ x);
            }
      }

      Local localObj = new Local(10);
      localObj.display();
  }
}
```

Output

```
C:\>javac LocalClassDemo.java

C:\>java LocalClassDemo
x = 10
```

iv. Static Nested Class

- The main benefit of Static Nested classes is that their reference is not attached to outer class reference.
- Object may be accessed directly.
- These classes cannot access non-static variables and methods. They can access only static variables and methods
- Static nested class can be referred by its class name.

Example:

```
class Outer
{
   static double outer_x;
   static double outer_y;
   Outer (double a, double b)
   {
      outer_x = a;
      outer_y = b;
   }
   static double outer_add()
   {
      return outer_x+outer_y;
   }
         static void outer_display()
   {
   Inner in = new Inner();
   in.inner_display();
   }

   static class Inner  // Static Inner Class
   {
         void inner_display()
         {
            System.out.println("x+y = "+ outer_add());
         }
   }
}

class StaticNestedClass
{
   public static void main (String args[])
   {
         Outer obj =new Outer(10,20);
         obj.outer_display();
   }
}
```

Output:

```
C:\>javac StaticNestedClass.java

C:\>java StaticNestedClass
x+y = 30.0
```

11. Final Class and Methods

- A final class is a class that is declared as a **final** which cannot have a subclass

Example:

```
final class A
{
   int a;
        A(int x) {a=x;}
   void display()
   {
     System.out.println("a = "+ a);
   }
}

class B extends A
{
   int b;
        B(int x,int y)
   {
       super(x);
       this.b=y;
   }
   void display()
   {
     System.out.println("b = "+ b);
   }
}

class FinalClass
{
   public static void main (String args[])
   {
         A objA= new A(10);
         B objB= new B(100,200);

         objA.display();
         objB.display();
   }
}
```

Output:

```
C:\>javac FinalClass.java
FinalClass.java:11: error: cannot inherit from final A
class B extends A
                ^
1 error
```

12. Passing Arguments by Value and by Reference

Arguments are the variables which are declared in the method prototype to receive the values as a input to the Method(Function).
Example:

```
int add(int a, int b) // method prototype
{
   //Body of the method add
  return a+b;
}
```

Here **a** and **b** are called as arguments. (also called as **formal arguments**)

Arguments are passed to the method from the method calling

Ex:
int x=10,y=20;
add(x , y); // method calling

Here x and y are **actual arguments**.

Arguments can be passed in two ways

i. Call by value
ii. Call by reference

i. <u>Call by value</u>

In call by value **actual arguments are copied in to formal arguments**.
Example:

```
class Swap
{
   int a,b;
   void setValues(int p, int q)
   {
      a=p;
      b=q;
   }
   void swapping()
   {
         int temp;
         temp =a;
         a=b;
         b=temp;
   }
   void display()
   {
     System.out.println("In Swap Class: a= "+a+" b= "+b);
   }
```

```
}
class CallByValue
{
  public static void main (String args[])
  {
   int x=10,y=20;
   System.out.println("Before Swap : x= "+x+ " y="+y);
   Swap obj =new Swap();
   obj.setValues(x,y);
   obj.swapping();
   obj.display();
   System.out.println("After Swap : x= "+x+ " y="+y);
  }
}
```

Output:

```
C:\>javac CallByValue.java

C:\>java CallByValue
Before Swap : x= 10 y=20
In Swap Class: a= 20 b= 10
After Swap : x= 10 y=20
```

ii. Call by reference

In call by reference the **object will be passed to the method as an argument**. At that time the actual and formal arguments are same.

That means any occurs in actual arguments will be reflected in the formal arguments.

Example:

```
class Swap
{
      int a,b;
      void setValues(Swap objSwap)
      {
         a = objSwap.a;
         b = objSwap.b;
      }
      void swapping()
      {
            int temp;
            temp =a;
            a=b;
            b=temp;
      }
      void display()
      {
        System.out.println("In Swap Class: a= "+a+" b= "+b);
      }
}
```

```
class CallByReference
{
  public static void main (String args[])
  {
Swap obj =new Swap();
obj.a=10;
obj.b=20;
System.out.println("Before Swap : obj.a = "+ obj.a+" obj.b="+ obj.b);

obj.setValues(obj); // call by reference
obj.swapping();
obj.display();
System.out.println("After Swap: obj.a = "+ obj.a+ " obj.b="+ obj.b);
  }
}
```

Output:

```
C:\1. JAVA\PPT Programs>javac CallByReference.java

C:\1. JAVA\PPT Programs>java CallByReference
Before Swap : obj.a = 10 obj.b=20
In Swap Class: a= 20 b= 10
After Swap : obj.a = 20 obj.b=10
```

13. Keyword this.

The keyword **this** provides reference to the current object.

Example:

```
class Add
{
      int a,b;
      void setValues(int a, int b)
      {
         this.a = a;
         this.b = b;
      }

      void add()
      {
        System.out.println("Sum = "+ (a+b) );
      }
}
class ThisKeyword
{
  public static void main (String args[])
  {
```

```
      Add obj= new Add();
      obj.setValues(10,20);
      obj.add();
 }
}
```

Output:

C:\ >javac ThisKeyword.java

C:\ >java ThisKeyword

Sum = 30

II. Methods

1. Introduction

A method in Java represents an action on data or behaviour of an object. In other programming languages, **the methods are called functions** or procedures.

A method is an encapsulation of declarations and executable statements meant to execute desired operations.

A few types of actions and behaviour of Methods are as follows.

1. It could involve carrying out computation on data presented to method.
2. The action may simply be rearranging the elements of an object. for example, sorting arrays.
3. The action may comprise finding or searching elements in the list.
4. The action may simply be the initialization of an object.
5. Methods may create images, voice, and multimedia as well as display.
6. Methods may define how an object will communicate with other objects.
7. It may simply answer an enquiry.
8. A method may tell whether an action is permissible or not.

- In Java, a method must be defined inside a class and an interface.
- An interface represents an encapsulation of constants, classes, interfaces, and one or more abstract methods that are implemented by a class.
- A method cannot be defined inside another method, but it can be defined inside a local class

2. Defining Methods

A method definition comprises two components:

1. ***Header*** that includes modifier, type, identifier, or name of method and a list of parameters.
   - The parameter list is placed in a pair of parentheses.
2. ***Body*** that is placed in braces ({ }) and consists of declarations and executable statement and other expressions.

Method definition:

```
Modifier return_type  method_name (datatype Parameter_Name,…)
{
  /*Statements --
    Body of the method*/
}
```

Modifier description is as follows.

Table 6.1 Brief description of non-access modifiers

Non-access modifiers	*Description*
static	With this modifier, a method may be called without an object of class. However, class reference is needed.
final	Method declared final cannot be modified (overridden) in a subclass.
native	It is used only for methods. It indicates that the method is implemented in a platform-dependent language like C.
transient	A variable is declared transient if it is desired that it should not be part of the persistent state of the object, that is, the transient variable value is not stored, whereas non-transient variables are stored when an object is stored in persistent memory.
synchronized	It is applied to make method thread safe. When it is used, it ensures that the method can be accessed by only one thread at a time.
volatile	It indicates to the compiler that a variable can be modified by the other thread.

Example:

```
class Add
{
      int a,b;
      void setValues(int x, int y) // method with two arguments
      {
         a = x;
         b = y;
      }

      void add() // method without arguments
      {
        System.out.println("Sum = "+ (a+b) );
      }
}
class MethodDemo
{
  public static void main (String args[])
  {
      Add obj= new Add();
      obj.setValues(10,20); // method calling
      obj.add();
 }
}
```

Output:

C:\ >javac MethodDemo.java

C:\ >java MethodDemo

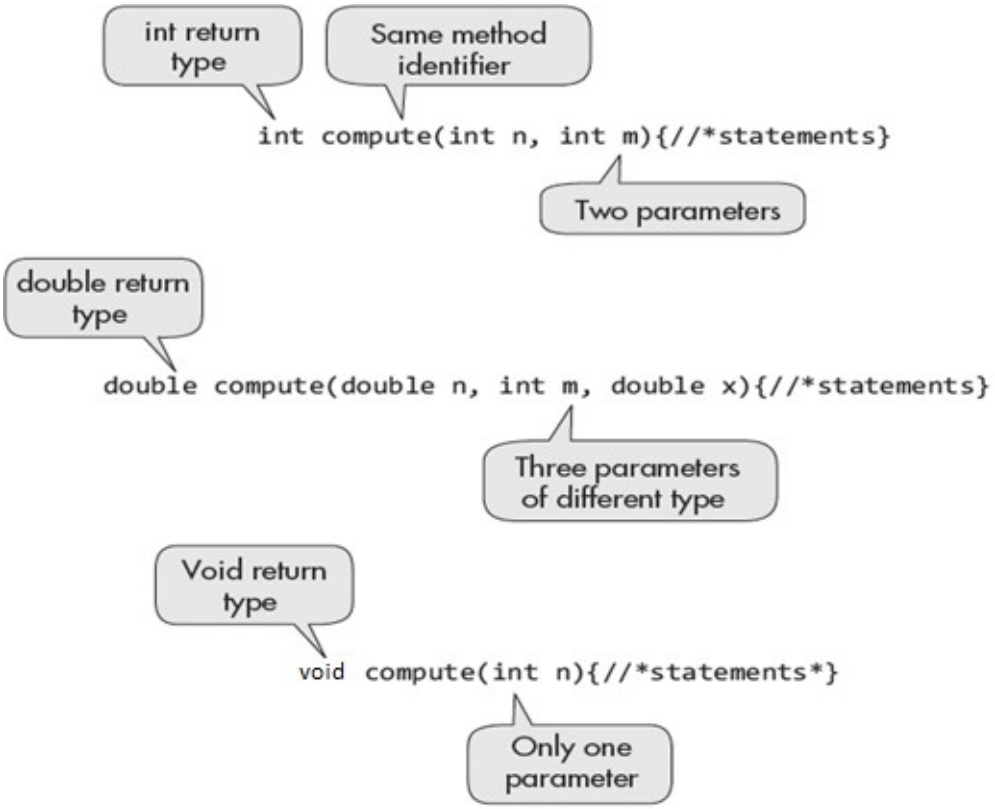
Sum = 30

3. Overloaded Methods

Methods with the same name and scope are permitted provided they have different signatures that include the following:

i. Number of parameters
ii. Data types of parameters
iii. Their order in the parameter list

The compiler executes the version of the method whose parameters match with the arguments. For example, the following types of declarations in the scope are permissible:



Example:

```
class Add
{
int a,b;
void setValues(int a, int b) // method with two arguments
{
   this.a = a;
   this.b = b;
}
void add() // method without arguments
{
  System.out.println("In add() method Sum = "+ (a+b) );
}
//Method overloding - integer datatype arguments
void add(int a, int  b)
{
 System.out.println("In add(int, int) Method- sum= "+ (a+b) );
}

//Method overloding - double datatype arguments
void add(double a, double b)
{
 System.out.println("In add(double, double) MethodSum = "+ (a+b) );
}
}

class MethodOverload
{
  public static void main (String args[])
  {
  Add obj= new Add();
  obj.setValues(10,20); // method calling
  obj.add(); // calling method without arguments
  obj.add(15,30);// calling method with integer datatype arguments
  obj.add(10.3, 30.4); // calling method with double datatype arguments
  }
}
```

```
C:\>javac MethodOverload.java

C:\>java MethodOverload
In add() method Sum = 30
In add(int, int) Method- sum= 45
```

```
In add(double, double) MethodSum = 40.7
```

4. Overloaded ConstructorMethods

A constructor method is automatically called whenever a new object of the class is constructed. It **creates and initializes the Object**.

A constructor method has the **same name as the name of class** to which it belongs. **It has no type and it does not return any value**. **It only initializes the object**.

The **constructor method may also be overloaded** by changing the number of default values. Therefore, constructors with different parameters may be declared. For the remaining parameters, it will pick up default values when these are not specified in the object definition.

Example:

```
class AddDemo
{
 int a,b;
 AddDemo() // Constructor without arguments
 {
   a=10;
   b=20;
 }

 // Constructor Overloading with arguments
 AddDemo(int x, int y)
 {
   a = x;
   b = y;
 }

void add() // method without arguments
{
 System.out.println("a = " + a + ", b = "+ b+ ":  Sum = "+ (a+b) );
}
}

class ConstructorOverload
{
  public static void main (String args[])
  {
  AddDemo obj1= new AddDemo(); //calling constructor without arguments
  obj1.add();

  AddDemo obj2= new AddDemo(150,60); //calling constructor with arguments
  obj2.add();
 }
}
```

Output:

```
C:\>javac ConstructorOverload.java

C:\>java ConstructorOverload
a = 10, b = 20:  Sum = 30
a = 150, b = 60:  Sum = 210
```

5. Class Objects as Parameters in Methods

Objects can be passed as parameters to the Methods just like primitive data types. It is called as Call by Reference.

Example:

```
class AddDemo
{
 int a,b;

 void add(AddDemo obj2) // method with Object as an
argument
 {
   System.out.println("Sum  =  "+  (obj2.a  +  obj2.b)
);
 }
}
class ObjectAsParameters
{
  public static void main (String args[])
  {
 AddDemo obj1= new AddDemo();
 obj1.a=180;
 obj1.b=50;
 obj1.add(obj1);
 }
}
```

Output:

```
C:\>javac ObjectAsParameter.java

C:\>java ObjectAsParameter
Sum = 230
```

6. Access Control

Java supports access control at the class level and at the level of class members. At the class level, the following two categories are generally used:

i. **default case no modifier applied :** In the default case, when no access specifier is applied, the class can be accessed by other classes only in the same package
ii. **public :** A class declared public may be accessed by any other class in any package.

In a class, Java supports the information hiding mechanism so that the user of a class does not get to know how the process is taking place. A class contains data members and method members or a nested class.

To access any of the members data method, or nested class-can be controlled by the following modifiers.

i. private
ii. protected

iii. public
iv. default case-no modifier specified

i. **private :** The private members can o**nly be accessed by the** other members (methods) of the **same class**. No other code outside the class can access them.
Ex:

```
private int x;
private int getx()
{
   return x;
}
```

ii. **protected :** The protected members can accessed by own class and **derived class only**.

```
protected int x;
protected int getx()
{
   return x;
}
```

iii. public : The public members can accessed **by all the classes**.
Ex:

```
public int x;
public int getx()
{
   return x;
}
```

iv. **default case (** no modifier specified): The default members can accessed by **all the classes within the package only.**

```
int x;
int getx()
{
   return x;
}
```

7. Recursive Methods

A Method which is calling itself is called as Recursive Method.

Example: Recursive method to find factorial of a given number.

```
class Fact
{
      int factorial (int n)
      {
         if(n<2)
           return n;
         else
           return  n*(factorial(n-1));
      }

}

class FactDemo
{
   public static void main(String[] args)
   {
     Fact obj =new Fact();
     int n=5;
     int res = obj.factorial(n);
     System.out.println("Factorial of " + n + " = " +res);
   }
}
```

Output:

```
C:\>javac FactDemo.java

C:\>java FactDemo
Factorial of 5 = 120
```

8. Nesting of Methods

A method calling in another method with in the class is called as Nesting of Methods.

Example:

```
class Rectangle
{
 void perimeter(int l, int w)
 {
   System.out.println("Length ="+l+", Width= "+w);
   System.out.println("Perimeter = " + (l+w));
 }

 void area(int l, int w)
 {
    perimeter(l,w); // Nesting of Method
    System.out.println("Area = " + (l*w));
 }
}

class RectangleDemo
{
   public static void main(String[] args)
   {
        Rectangle obj = new Rectangle();
        obj.area(5,4);
   }
}
```

Output:

```
C:\ >javac RectangleDemo.java

C:\ >java RectangleDemo
Length =5, Width= 4
Perimeter = 9
Area = 20
```

9. Overriding Methods

Inheritance is an OOP property that allows us to derive a new class (subclass) from an existing class (superclass). The subclass inherits the attributes and methods of the superclass.

Now, if the same method is defined in both the superclass and the subclass, then the method of the subclass class overrides the method of the superclass. This is known as method overriding.

Example:

```
class Animal {
  public void displayInfo() {
    System.out.println("I am an animal.");
  }
}

class Dog extends Animal
{
  public void displayInfo()
 {
    System.out.println("I am a dog.");
  }
}

class Main
```

```
{
  public static void main(String[] args)
{
    Dog d1 = new Dog();
    d1.displayInfo();
  }
}
```

Output;

Iam a Dog

Java Overriding Rules

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
- We cannot override the method declared as final and static

10. Attributes Final and Static

Attribute Final:

Final Variable:

The value of a variable declared final cannot be changed in the program. It makes the variable a constant. A few examples of declarations are as follows:

final double PI = 3.14159; // The value of PI cannot be changed in its

scope final int M = 900; // The value of M cannot be changed in its scope

final double X = 7.5643; // The value of x cannot be changed in its scope.

- As mentioned in the comments, the values of PI, M, and x cannot be changed in their respective scopes.

Final Method:

- The attribute final may be used for **methods** as well as for classes. These are basically connected with inheritance of classes.
- When final keyword is used with Java method, it becomes the final method.
- A final method cannot be overridden in a sub-class.

Final Class:

- A Java class with final modifier is called final class A final class cannot be sub-classed or inherited. Several classes in Java are final including String, Integer, and other wrapper classes.

- There are certain important points to be noted when using final keyword in Java
  - New value cannot be reassigned to a variable defined as final in Java.
  - Final keyword can be applied to a member variable, local variable, method, or class.
  - Final member variable must be initialized at the time of declaration.
  - Final method cannot be overridden in Java
  - Final class cannot be inheritable in Java
  - Final is different from finally keyword, which is used on Exception handling in Java

Example- 1:

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

        int n =10; // Normal variable  final int f =
        20; // final variable

        System.out.println("n = "+ n);
        System.out.println("f = "+ f);

        n = 50; // Now the value 50 is assigned to variable n
        f = 60; // Error : f is final variable can not be changed
        }
}
```

Output:

```
C:\>javac Final.java
Final.java:13: error: cannot assign a value to final variable f
            f = 60; // Error : f is final variable can not be changed
            ^
1 error
C:\>
```

Example-2:

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

        int n =10; // Normal variable  final int f =
        20; // final variable

        System.out.println("n = "+ n);
        System.out.println("f = "+ f);

        n = 50; // Now the value 50 is assigned to variable n System.out.println("n = "+ n);}
}
C:\>javac Final.java

C:\>java Final n =
```

10
f = 20
n = 50

III. Static Variables and Methods

Static Variables:

- The static variables are class variables. Only one copy of such variables is kept in the memory and all the objects share that copy.
- The static variables are accessed through class reference**,** whereas the instance variables are accessed through class object reference
- The variables in a class may be modified by modifier static.
- The non-static variables declared in a class are instance variables Each object of the class keeps a copy of the values of these variables.

Static Methods:

- The static methods are similar to class methods and can be invoked without any reference of object of class, however, class reference (name of class) is needed, as in the following example The method like sqrt() is declared as static method in Math class and is called

```
Math.sqrt(5); // Finds square root of 5
```

The static method is called using the method name that is preceded by the class name; in this case. Math and period ().

Program 3.18: illustration of using static methods of class Math

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

  System.out.println("The Square root root of 16 = "+ Math.sqrt(16)); System.out.println("The cubroot
  root of 27 = "+ Math.cbrt(27));
  //printing five random variables for(int i =1;
  i<=5;i++)
          System.out.println("Random Number "  + i + " = " +
                                        (int)(100 *Math.random()));
          }
}
```

```
E:\>javac StaticMethods.java

E:\>java StaticMethods
The Square  root root of 16 = 4.0 The cubroot
root of 27 = 3.0
 Random  Number 1  =  77
 Random  Number 2  =  69
 Random  Number 3  =  83
 Random  Number 4  =  2
 Random  Number 5  =  66

E:\>java StaticMethods
The Square root root of 16 = 4.0 The cubroot
root of 27 = 3.0
 Random  Number 1  =    67
 Random  Number 2  =    31
 Random  Number 3  =    10
 Random  Number 4  =    13
 Random  Number 5  =    40
```