MCA FIRST SEMESTER- 2021
MCA20103-OBJECT ORIENTED PROGRAMMING USING JAVA
UNIT-3

1. METHODS IN JAVA:

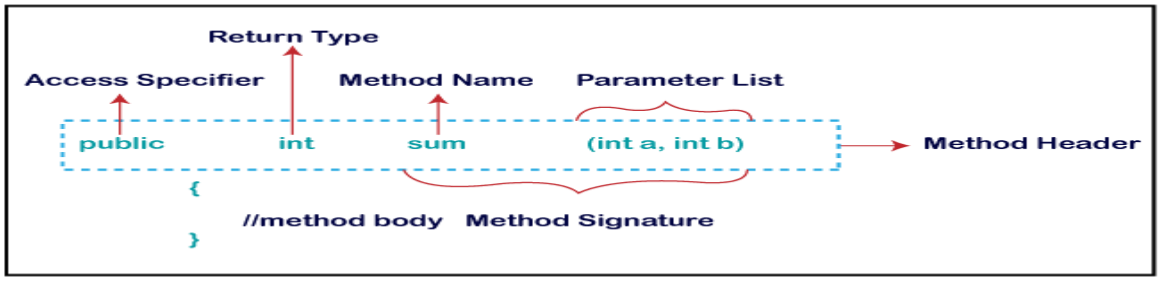
In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**.

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.
The most important method in Java is the **main()** method.

❖ **Method Declaration**

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.

Method Declaration



- ❖ **Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.
- ❖ **Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

  Public: The method is accessible by all classes when we use public specifier in our application.

  Private: When we use a private access specifier, the method is accessible only in the classes in which it is defined.

  Protected: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.

  Default: When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

- **Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
- **Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.
- **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
- **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Let's see an example of the predefined method.

Demo.java

```
class Demo

        {
        public static void main(String[] args)
        {
        // using the max() method of Math class
        System.out.println("The maximum number is: " + Math.max(9,7));
        }
        }
```

Output:

```
The maximum number is: 9
```

In the above example, we have used three predefined methods **main(), print(),** and **max()**. We have used these methods directly without declaration because they are predefined. The print() method is a method of **PrintStream** class that prints the result on the console. The max() method is a method of the **Math** class that returns the greater of two numbers.

❖ User-defined Method

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

Addition.java

```
class Addition
{
public static void main(String[] args)
{
int a = 19;
int b = 5;
//method calling
int c = add(a, b);   //a and b are actual parameters
System.out.println("The sum of a and b is= " + c);
}
//user defined method
public static int add(int n1, int n2)   //n1 and n2 are formal parameters
{
int s;
s=n1+n2;
return s; //returning the sum
}
}
```

Output:

```
The sum of a and b is= 24
```

2. Static Method:

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

Example of static method

Display.java

```
public class Display
{
public static void main(String[] args)
{
show();
static void show()
{
System.out.println("It is an example of static method.");
}
}
```

Output:

```
It is an example of a static method.
```

3. Instance Method:

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

InstanceMethodExample.java

```
public class InstanceMethodExample
{
public static void main(String [] args)
{
//Creating an object of the class
InstanceMethodExample obj = new InstanceMethodExample();
//invoking instance method
System.out.println("The sum is: "+obj.add(12, 13));
}
int s;
//user-defined method because we have not used static keyword
public int add(int a, int b)
{
s = a+b;
//returning the sum
return s;
```

```
        }
        }
```

Output:

```
The sum is: 25
```

4. this keyword in java:

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

Example:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Test it Now

Output:

```
111 ankit 5000
112 sumit 6000
```

5.Passing Primitive Data Types and objects to Methods:

❖ Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `String` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: fname is a **parameter**, while Liam, Jenny and Anja are **arguments**.

❖ Multiple Parameters

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

```
public class Main {
  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }

  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

6. Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

Syntax:

```
returntype methodname(){
 //code to be executed
 methodname();//calling same method
 }
```

Factorial Number example program:

```
public class RecursionExample3
 {
    static int factorial(int n)
     {
        if (n == 1)
        return 1;
       else
        return(n * factorial(n-1));
 }

public static void main(String[] args)
{
    System.out.println("Factorial of 5 is: "+factorial(5));
}
}
```

Output:

```
Factorial of 5 is: 120
```

Working of above program:

```
factorial(5)
   factorial(4)
      factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
      return 4*6 = 24
   return 5*24 = 120
```

7. RELATIONSHIP BETWEEN OBJECTS :

- **Association**

Association refers to the relationship between multiple objects. It refers to how objects are related to each other and how they are using each other's functionality. Composition and aggregation are two types of association.

- **Composition**

The composition is the strong type of association. An association is said to composition if an Object owns another object and another object cannot exist without the owner object. Consider the case of Human having a heart. Here Human object contains the heart and heart cannot exist without Human.

- **Aggregation**

Aggregation is a weak association. An association is said to be aggregation if both Objects can exist independently. For example, a Team object and a Player object. The team contains multiple players but a player can exist without a team.

8. Java Inner Classes:

Java inner class or nested class is a class which is declared inside the class or interface.We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
//code
 class Java_Inner_class{
  //code
 }
}
```

- **Advantage of java inner classes**

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Example:
```
class OuterClass {
  int x = 10;
  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

9. super Keyword:

The `super` keyword refers to superclass (parent) objects. It is used to call superclass methods, and to access the superclass constructor.The most common use of the `super` keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.

Example:Using `super` to call the superclass of `Dog` (subclass):

```
class Animal { // Superclass (parent)
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Dog extends Animal { // Subclass (child)
  public void animalSound() {
    super.animalSound(); // Call the superclass method
    System.out.println("The dog says: bow wow");
  }
}

public class Main {
  public static void main(String args[]) {
    Animal myDog = new Dog(); // Create a Dog object
    myDog.animalSound(); // Call the method on the Dog object
  }
}

Output:
The animal makes a sound
The dog says: bow wow
```

10. INHERITANCE: **Definition:-** The technique of deriving a new class from the already existing class is called inheritance. The existing class is called base class or super class and new classs is called derived class or subclass.

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that we can create new classes that are built upon existing classes. When we inherit from an existing class, we can reuse methods and fields of the parent class. Moreover, we can add new methods and fields in our current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

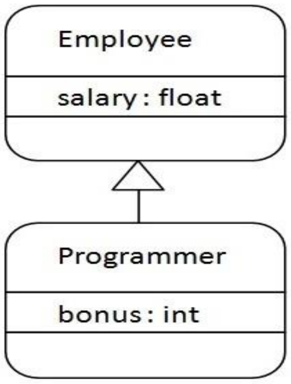
The syntax of Java Inheritance:-

```
 class Subclass-name extends Superclass-name
  {
  //methods and fields
  }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or super class, and the new class is called child or subclass.

Java Inheritance Example:-



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

Ex:-

```
 class Employee
{
 float salary=40000;
}
class Programmer extends Employee
{
 int bonus=10000;
 public static void main(String args[])
{
  Programmer p=new Programmer();
  System.out.println("Programmer salary is:"+p.salary);
  System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Output:-

```
Programmer salary is:40000.0
Bonus of Programmer is:10000
```

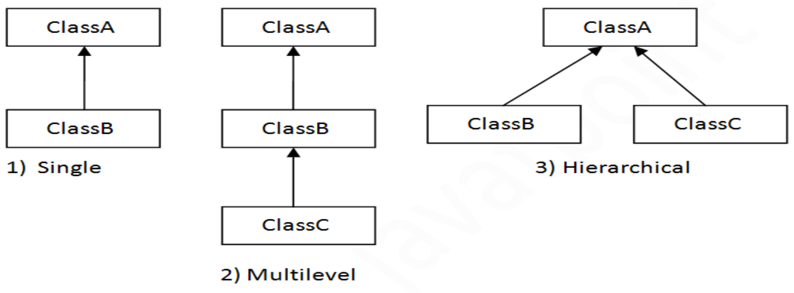
In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

❖ **Types of inheritance in java**

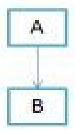
Inheritance allows the subclass to inherit all the variables and methods of their parent class. On the basis of class, there can be three types of inheritance in java:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

Java does not directly implements **multiple inheritance** and **hybrid inheritance** these concepts are implemented by using **interfaces.**



1. Single inheritance: Derivation of a class from only one base class is called single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.



(a) Single Inheritance

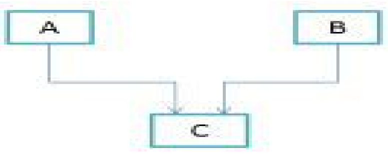
Ex:- **class** Animal{
 void eat(){System.out.println("eating...");}

```
 }
class Dog extends Animal{
 void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:- barking...
eating...

2.Multiple inheritance:

"**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.



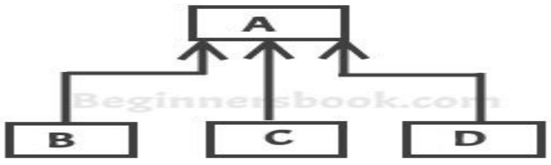
(b) Multiple Inheritance

Note 1: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

Note 2: Most of the new OO languages like **Small Talk, Java, C# do not support Multiple inheritance**. Multiple Inheritance is supported in C++.

3. Hierarchial inheritance:

When more than one classes inherit a same class then this is called hierarchical inheritance. For example class B, C and D extends a same class A. Lets see the diagram representation of this:



Hierarchical Inheritance

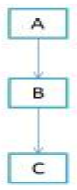
```
Ex: class A
{
   public void methodA()
   {
      System.out.println("method of Class A");
   }
}
class B extends A
{
   public void methodB()
   {
      System.out.println("method of Class B");
   }
}
class C extends A
{
  public void methodC()
  {
     System.out.println("method of Class C");
  }
}
class D extends A
{
  public void methodD()
  {
     System.out.println("method of Class D");
  }
}
class JavaExample
{
  public static void main(String args[])
  {
     B obj1 = new B();
     C obj2 = new C();
     D obj3 = new D();
     //All classes can access the method of class A
     obj1.methodA();
     obj2.methodA();
     obj3.methodA();
  }
}
```

Output:

```
method of Class A
method of Class A
method of Class A
```

4. Multilevel inheritance:

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A. For more details and example refer – Multilevel inheritance in Java.

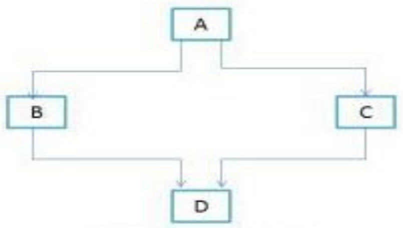


(d) Multilevel Inheritance

```
Class X
{
   public void methodX()
   {
     System.out.println("Class X method");
   }
}
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
   public void methodZ()
   {
     System.out.println("class Z method");
   }
   public static void main(String args[])
   {
     Z obj = new Z();
     obj.methodX(); //calling grand parent class method
     obj.methodY(); //calling parent class method
     obj.methodZ(); //calling local method
  }
}
```

5.Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



(e) Hybrid Inheritance

Example program:

```
class C
{
   public void disp()
   {
        System.out.println("C");
   }
}

class A extends C
{
   public void disp()
   {
        System.out.println("A");
   }
}

class B extends C
{
   public void disp()
   {
        System.out.println("B");
   }

}

class D extends A
{
   public void disp()
   {
        System.out.println("D");
   }
   public static void main(String args[]){

        D obj = new D();
        obj.disp();
   }
}
```

Output:

```
D
```

11. POLYMORPHISM:

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by **method overloading** and **method overriding.**

Method overloading:

Method overloading means writing to two or more methods with the same name and with the different signature is called method overloading.

To create and overloaded method we have to provide different method definition with the same name with different parameter list. The difference may be is number of arguments or type of arguments. That is each parameter list should be unique.

Method overloading is used when objects are required to perform similar tasks but using different input parameters when we call a method in an object java searches for the method name first and then the number and type of parameters to decide which method to execute this process is known as polymorphism.

Eg:

```
class Arith
{
int add( int x, int y)
{
return x+y;
}
float add(float x,float y)
{
return x+y;
}
}
class Test
{
public static void mainh (String args[ ])
{
Arith a=new Arith();
int m=a.add(3,4);
float n=a.add(4.5,3.6);
System.out.println("integer addition="+m);
System.out.println("float addition="+n);
}
}
```

Output:

```
Integer addition=7
Floating addition=8.1
```

Method overriding

Writing two or more methods in super and sub classes such that the methods have same name, same signature is called method over riding.

Over riding is possible by defining a method in subclass that has the same name, same parameters list and same return type as the method in the super class when that methods defining in the subclass is invoked and executed instead of super class method.

Eg:

```
import java.lang.Math;
class one
{
void calc(int x)
{
System.out.println(square  value=”+(x*x));
}
class two extends one
{
void calc(int x)
{
System.out.println(“square root=”+(math.sqrt(x));
}
}
class Test
{
public static void main(String args[ ])
{
two t=new two();
t.calc(49);
}
}
```

Output:

```
Square root=7
```

12. Final Methods and Final classes:

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant). Final variables are nothing but constants. We cannot change the value of a final variable once it is initialized. Lets have a look at the below code:

```
class Demo{

   final int MAX_VALUE=99;
   void myMethod(){
      MAX_VALUE=101;
   }
   public static void main(String args[]){
      Demo obj=new  Demo();
      obj.myMethod();
   }
}
```

We got a compilation error in the above program because we tried to change the value of a final variable "MAX_VALUE".

Blank final variable

A final variable that is not initialized at the time of declaration is known as **blank final variable**. We **must initialize the blank final variable in constructor** of the class otherwise it will throw a compilation error (Error: variable MAX_VALUE might not have been initialized).

This is how a blank final variable is used in a class:

```
class Demo{
   //Blank final variable
   final int MAX_VALUE;

   Demo(){
      //It must be initialized in constructor
      MAX_VALUE=100;
   }
   void myMethod(){
      System.out.println(MAX_VALUE);
   }
   public static void main(String args[]){
      Demo obj=new  Demo();
      obj.myMethod();
   }
}
```

Output:

```
100
```

Whatistheuseofblankfinalvariable?

Lets say we have a Student class which is having a field called Roll No. Since Roll No should not be changed once the student is registered, we can declare it as a final variable in a class but we cannot initialize roll no in advance for all the students(otherwise all students would be having same roll no). In such case we can declare roll no variable as blank final and we initialize this value during object creation like this:

```
class StudentData{
   //Blank final variable
   final int ROLL_NO;

   StudentData(int rnum){
      //It must be initialized in constructor
      ROLL_NO=rnum;
   }
   void myMethod(){
      System.out.println("Roll no is:"+ROLL_NO);
   }
   public static void main(String args[]){
      StudentData obj=new  StudentData(1234);
      obj.myMethod();
   }
}
```

Output:

```
Roll no is:1234
```

) final method

A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it?

Example:

```
class XYZ{
   final void demo(){
      System.out.println("XYZ Class Method");
   }
}

class ABC extends XYZ{
   void demo(){
      System.out.println("ABC Class Method");
   }

   public static void main(String args[]){
      ABC obj= new ABC();
      obj.demo();
   }
}
```

The above program would throw a compilation error, however we can use the parent class final method in sub class without any issues. Lets have a look at this code: This program would run fine as we are not overriding the final method. That shows that final methods are inherited but they are not eligible for overriding.

```
class XYZ{
   final void demo(){
      System.out.println("XYZ Class Method");
   }
}

class ABC extends XYZ{
   public static void main(String args[]){
      ABC obj= new ABC();
      obj.demo();
   }
}
```

Output:

```
XYZ Class Method
```

3) final class

In Java, the final class cannot be inherited by another class. For example,We cannot extend a final class. Consider the below example:

```
final class XYZ{
}

class ABC extends XYZ{
   void demo(){
      System.out.println("My Method");
   }
   public static void main(String args[]){
      ABC obj= new ABC();
      obj.demo();
   }
}
```

Output:

```
The type ABC cannot inherit the final class XYZ // compilation error
```

Points to Remember:

1) A constructor cannot be declared as final.
2) Local final variable must be initializing during declaration.
3) All variables declared in an interface are by default final.
4) We cannot change the value of a final variable.
5) A final method cannot be overridden.
6) A final class not be inherited.
7) If method parameters are declared final then the value of these parameters cannot be changed.
8) It is a good practice to name final variable in all CAPS.

13. Type Casting:

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  `byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  `double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

❖ Widening Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. Widening casting is done automatically when passing a smaller size type to a larger size type: It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

Example:

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

❖ Narrowing Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error. Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

14. ABSTRACT CLASSES:

Data **abstraction** is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

❖ Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the `abstract` keyword to declare an abstract class. For example,

Ex:
```
abstract class Language {

  // method of abstract class
  public void display() {
    System.out.println("This is Java Programming");
  }
}

class Main extends Language {

  public static void main(String[] args) {

    // create an object of Main
    Main obj = new Main();

    // access method of abstract class
    // using object of Main class
    obj.display();
  }
}
```

Output

```
This is Java programming
```
In the above example, we have created an abstract class named *Language*. The class contains a regular method `display()`.We have created the Main class that inherits the abstract class. Notice the statement,

```
obj.display();
```
Here, *obj* is the object of the child class *Main*. We are calling the method of the abstract class using the object *obj*.

❖ Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method.

For example,

```
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}

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

    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }
}
```

Output

```
Bark bark
I can eat.
```

In the above example, we have created an abstract class *Animal*. The class contains an abstract method `makeSound()` and a non-abstract method `eat()`.

We have inherited a subclass *Dog* from the superclass *Animal*. Here, the subclass *Dog* provides the implementation for the abstract method `makeSound()`.

Key Points to Remember

- We use the abstract keyword to create abstract classes and methods.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We cannot create objects of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- We can access the static attributes and methods of an abstract class using the reference of the abstract class.