Inheritance in Java Programming with examples

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Child Class:

The class that extends the features of another class is known as child class, sub class or derived class.

Parent Class:

The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

Inheritance allows us to reuse of code, it improves reusability in your java application.

Note: The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.

This means that the data members(instance variables) and methods of the parent class can be used in the child class as.

If you are finding it difficult to understand what is class and object then refer the guide that I have shared on object oriented programming: OOPs Concepts

Lets back to the topic:

Syntax: Inheritance in Java

To inherit a class we use extends keyword. Here class XYZ is child class and class ABC is parent class. The class XYZ is inheriting the properties and methods of ABC class.

```
class XYZ extends ABC
{
}
```

Inheritance Example

In this example, we have a base class `Teacher` and a sub class `PhysicsTeacher`. Since class `PhysicsTeacher` extends the designation and college properties and work() method from base class, we need not to declare these properties and method in sub class.
Here we have collegeName, designation and work() method which are common to all the teachers so we have declared them in the base class, this way the child classes like `MathTeacher`, `MusicTeacher` and `PhysicsTeacher` do not need to write this code and can be used directly from base class.

```
class Teacher {
   String designation = "Teacher";
   String collegeName = "Beginnersbook";
   void does(){
         System.out.println("Teaching");
   }
}

public class PhysicsTeacher extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
         PhysicsTeacher obj = new PhysicsTeacher();
         System.out.println(obj.collegeName);
         System.out.println(obj.designation);
         System.out.println(obj.mainSubject);
         obj.does();
   }
}
```

Output:

```
Beginnersbook
Teacher
Physics
Teaching
```

Based on the above example we can say that `PhysicsTeacher` **IS-A** `Teacher`. This means that a child class has IS-A relationship with the parent class. This is inheritance is known as **IS-A relationship** between child and parent class

Note:
The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

```
class Teacher {
   private String designation = "Teacher";
   private String collegeName = "Beginnersbook";
   public String getDesignation() {
         return designation;
   }
   protected void setDesignation(String designation) {
         this.designation = designation;
   }
   protected String getCollegeName() {
         return collegeName;
   }
   protected void setCollegeName(String collegeName) {
         this.collegeName = collegeName;
   }
   void does(){
         System.out.println("Teaching");
   }
}

public class JavaExample extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
         JavaExample obj = new JavaExample();
         /* Note: we are not accessing the data members
          * directly we are using public getter method
          * to access the private members of parent class
          */
         System.out.println(obj.getCollegeName());
         System.out.println(obj.getDesignation());
         System.out.println(obj.mainSubject);
         obj.does();
   }
}
```

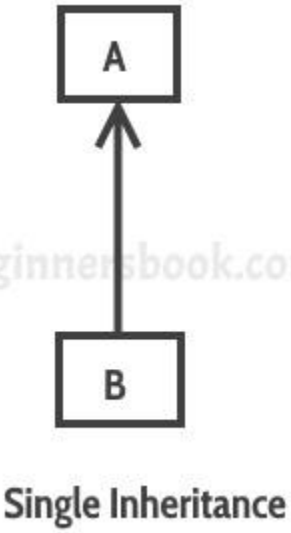
The output is:

```
Beginnersbook
Teacher
Physics
Teaching
```

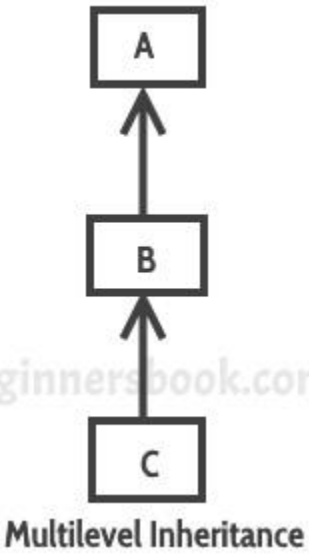
The important point to note in the above example is that the child class is able to access the private members of parent class through **protected methods** of parent class. When we make a instance variable(data member) or method **protected**, this means that they are accessible only in the class itself and in child class. These public, protected, private etc. are all access specifiers and we will discuss them in the coming tutorials.

Types of inheritance

To learn types of inheritance in detail, refer: Types of Inheritance in Java.
Single Inheritance: refers to a child and parent class relationship where a class extends the another class.

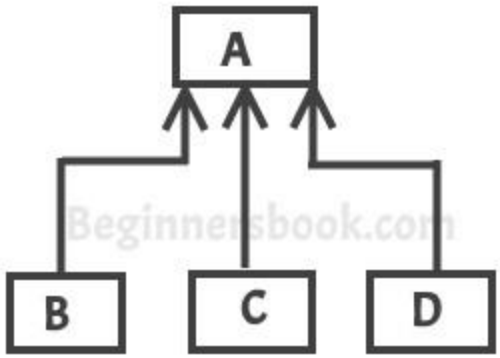


Multilevel inheritance: refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.



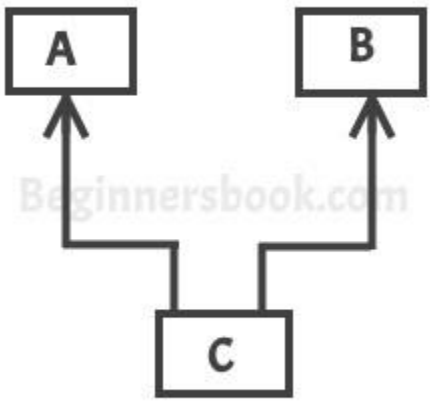
Hierarchical inheritance: refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D

extends the same class A.



Hierarchical Inheritance

Multiple Inheritance: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example class C extends both classes A and B. Java doesn't support multiple inheritance, read more about it here.



Multiple Inheritance

Hybrid inheritance: Combination of more than one types of inheritance in a single program. For example class A & B extends class C and another class D extends class A then this is a hybrid inheritance example because it is a combination of single and hierarchical inheritance.

Constructors and Inheritance

constructor of sub class is invoked when we create the object of subclass, it by default invokes the default constructor of super class. Hence, in inheritance the objects are constructed top-down. The superclass constructor can be called explicitly using the super keyword, but it should be first statement in a constructor. The super keyword refers to the superclass, immediately above of the calling class in the hierarchy. The use of multiple super keywords to access an ancestor class other than the direct parent is not permitted.

```
class ParentClass{
   //Parent class constructor
   ParentClass(){
         System.out.println("Constructor of Parent");
   }
}
class JavaExample extends ParentClass{
   JavaExample(){
         /* It by default invokes the constructor of parent class
          * You can use super() to call the constructor of parent.
          * It should be the first statement in the child class
          * constructor, you can also call the parameterized constructor
          * of parent class by using super like this: super(10), now
          * this will invoke the parameterized constructor of int arg
          */
         System.out.println("Constructor of Child");
   }
   public static void main(String args[]){
         //Creating the object of child class
         new JavaExample();
   }
}
```

Output:

```
Constructor of Parent
Constructor of Child
```

Inheritance and Method Overriding

When we declare the same method in child class which is already present in the parent class the this is called method overriding. In this case when we call the method from child class object, the child class version of the method is called. However we can call the parent class method using super keyword as I have shown in the example below:

```
class ParentClass{
   //Parent class constructor
   ParentClass(){
```

```
        System.out.println("Constructor of Parent");
    }
    void disp(){
        System.out.println("Parent Method");
    }
}
class JavaExample extends ParentClass{
    JavaExample(){
        System.out.println("Constructor of Child");
    }
    void disp(){
        System.out.println("Child Method");
        //Calling the disp() method of parent class
        super.disp();
    }
    public static void main(String args[]){
        //Creating the object of child class
        JavaExample obj = new JavaExample();
        obj.disp();
    }
}
```

The output is :

```
Constructor of Parent
Constructor of Child
Child Method
Parent Method
```

OOPs concepts – What is Aggregation in java?

Aggregation is a special form of association. It is a relationship between two classes like association, however its a **directional** association, which means it is strictly a **one way association.** It represents a **HAS-A** relationship.

Aggregation Example in Java

For example consider two classes `Student` class and `Address` class. Every student has an address so the relationship between student and address is a Has-A relationship. But if you consider its vice versa then it would not make any sense as an `Address` doesn't need to have a `Student` necessarily. Lets write this example in a java program.
`Student Has-A Address`

```
class Address
```

```
{
   int streetNum;
   String city;
   String state;
   String country;
   Address(int street, String c, String st, String coun)
   {
       this.streetNum=street;
       this.city =c;
       this.state = st;
       this.country = coun;
   }
}
class StudentClass
{
   int rollNum;
   String studentName;
   //Creating HAS-A relationship with Address class
   Address studentAddr;
   StudentClass(int roll, String name, Address addr){
       this.rollNum=roll;
       this.studentName=name;
       this.studentAddr = addr;
   }
   public static void main(String args[]){
       Address ad = new Address(55, "Agra", "UP", "India");
       StudentClass obj = new StudentClass(123, "Chaitanya", ad);
       System.out.println(obj.rollNum);
       System.out.println(obj.studentName);
       System.out.println(obj.studentAddr.streetNum);
       System.out.println(obj.studentAddr.city);
       System.out.println(obj.studentAddr.state);
       System.out.println(obj.studentAddr.country);
   }
}
```

Output:

```
123
Chaitanya
55
Agra
UP
India
```

The above example shows the **Aggregation** between Student and Address classes. You can see that in Student class I have declared a property of type Address to obtain student address. Its a typical example of Aggregation in Java.

Why we need Aggregation?

To maintain code re-usability. To understand this lets take the same example again. Suppose there are two other classes College and Staff along with above two

classes `Student` and `Address`. In order to maintain Student's address, College Address and Staff's address we don't need to use the same code again and again. We just have to use the reference of Address class while defining each of these classes like:

```
Student Has-A Address (Has-a relationship between student and address)
College Has-A Address (Has-a relationship between college and address)
Staff Has-A Address (Has-a relationship between staff and address)
```

Hence we can improve code re-usability by using Aggregation relationship.

So if I have to write this in a program, I would do it like this:

```
class Address
{
   int streetNum;
   String city;
   String state;
   String country;
   Address(int street, String c, String st, String coun)
   {
        this.streetNum=street;
        this.city =c;
        this.state = st;
        this.country = coun;
   }
}
class StudentClass
{
   int rollNum;
   String studentName;
   //Creating HAS-A relationship with Address class
   Address studentAddr;
   StudentClass(int roll, String name, Address addr){
        this.rollNum=roll;
        this.studentName=name;
        this.studentAddr = addr;
   }
   ...
}
class College
{
   String collegeName;
   //Creating HAS-A relationship with Address class
   Address collegeAddr;
   College(String name, Address addr){
        this.collegeName = name;
        this.collegeAddr = addr;
   }
   ...
}
class Staff
```

```
{
   String employeeName;
   //Creating HAS-A relationship with Address class
   Address employeeAddr;
   Staff(String name, Address addr){
        this.employeeName = name;
        this.employeeAddr = addr;
   }
   ...
}
```

As you can see that we didn't write the Address code in any of the three classes, we simply created the HAS-A relationship with the Address class to use the Address code. The dot dot(…) part in the above code can be replaced with the public static void main method, the code in it would be similar to what we have seen in the first example.

OOPs concepts – What is Association in java?

In this article we will discuss **Association in Java**. Association establishes relationship between two separate **classes** through their **objects**. The relationship can be one to one, One to many, many to one and many to many.

Association Example

```
class CarClass{
   String carName;
   int carId;
   CarClass(String name, int id)
   {
         this.carName = name;
         this.carId = id;
   }
}
class Driver extends CarClass{
   String driverName;
   Driver(String name, String cname, int cid){
         super(cname, cid);
         this.driverName=name;
   }
}
class TransportCompany{
   public static void main(String args[])
   {
         Driver obj = new Driver("Andy", "Ford", 9988);
         System.out.println(obj.driverName+" is a driver of car Id: "+obj.carId);
   }
```

```
}
```
Output:

```
Andy is a driver of car Id: 9988
```
In the above example, there is a one to one relationship(**Association**) between two classes: `CarClass`and `Driver`. Both the classes represent two separate entities.

Association vs Aggregation vs Composition

Lets discuss **difference between Association, Aggregation and Composition**:

Although all three are related terms, there are some major differences in the way they relate two classes. **Association** is a relationship between two separate classes and the association can be of any type say one to one, one to may etc. It joins two entirely separate entities.

Aggregation is a special form of association which is a unidirectional one way relationship between classes (or entities), for e.g. Wallet and Money classes. Wallet has Money but money doesn't need to have Wallet necessarily so its a one directional relationship. In this relationship both the entries can survive if other one ends. In our example if Wallet class is not present, it does not mean that the Money class cannot exist.

Composition is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other. For e.g. Human and Heart. A human needs heart to live and a heart needs a Human body to survive. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then its a composition. Heart class has no sense if Human class is not present.

Super keyword in java with example

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

The super keyword refers to the objects of immediate parent class. Before learning super keyword you must have the knowledge of inheritance in Java so that you can understand the examples given in this guide.

The use of super keyword

1) To access the data members of parent class when both parent and child class have member with same name
2) To explicitly call the no-arg and parameterized constructor of parent class
3) To access the method of parent class when child class has overridden that method.

Now lets discuss them in detail with the help of examples:

1) How to use super keyword to access the variables of parent class

When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.

Lets take an example to understand this: In the following program, we have a data member `num` declared in the child class, the member with the same name is already present in the parent class. There is no way you can access the `num` variable of parent class without using super keyword. .

```
//Parent class or Superclass or base class
class Superclass
{
   int num = 100;
}
//Child class or subclass or derived class
class Subclass extends Superclass
{
   /* The same variable num is declared in the Subclass
    * which is already present in the Superclass
    */
    int num = 110;
    void printNumber(){
        System.out.println(num);
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
    }
}
```

Output:
110

Accessing the num variable of parent class:
By calling a variable like this, we can access the variable of parent class if both the classes (parent and child) have same variable.

```
super.variable_name
```

Let's take the same example that we have seen above, this time in print statement we are passing `super.num` instead of num.

```
class Superclass
{
   int num = 100;
}
class Subclass extends Superclass
{
   int num = 110;
   void printNumber(){
        /* Note that instead of writing num we are
         * writing super.num in the print statement
         * this refers to the num variable of Superclass
         */
        System.out.println(super.num);
   }
   public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
   }
}
```

Output:
100
As you can see by using super.num we accessed the num variable of parent class.

2) Use of super keyword to invoke constructor of parent class

When we create the object of sub class, the new keyword invokes the constructor of child class, which implicitly invokes the constructor of parent class. So the order to execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed. It happens because compiler itself adds super()(this invokes the no-arg constructor of parent class) as the first statement in the constructor of child class.

Let's see an example to understand what I have explained above:

```
class Parentclass
{
   Parentclass(){
        System.out.println("Constructor of parent class");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
        /* Compile implicitly adds super() here as the
         *  first statement of this constructor.
         */
        System.out.println("Constructor of child class");
   }
   Subclass(int num){
        /* Even though it is a parameterized constructor.
         * The compiler still adds the no-arg super() here
         */
        System.out.println("arg constructor of child class");
   }
   void display(){
        System.out.println("Hello!");
   }
   public static void main(String args[]){
        /* Creating object using default constructor. This
         * will invoke child class constructor, which will
         * invoke parent class constructor
         */
        Subclass obj= new Subclass();
        //Calling sub class method
        obj.display();
        /* Creating second object using arg constructor
         * it will invoke arg constructor of child class which will
         * invoke no-arg constructor of parent class automatically
         */
        Subclass obj2= new Subclass(10);
        obj2.display();
   }
}
```

Output:

```
Constructor of parent class
Constructor of child class
Hello!
Constructor of parent class
arg constructor of child class
Hello!
```

Parameterized super() call to invoke parameterized constructor of parent class

We can call super() explicitly in the constructor of child class, but it would not make any sense because it would be redundant. It's like explicitly doing

something which would be implicitly done otherwise.
However when we have a constructor in parent class that takes arguments then we can use parameterized super, like `super(100);` to invoke parameterized constructor of parent class from the constructor of child class.
Let's see an example to understand this:

```
class Parentclass
{
   //no-arg constructor
   Parentclass(){
         System.out.println("no-arg constructor of parent class");
   }
   //arg or parameterized constructor
   Parentclass(String str){
         System.out.println("parameterized constructor of parent class");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
        /* super() must be added to the first statement of constructor
         * otherwise you will get a compilation error. Another important
         * point to note is that when we explicitly use super in constructor
         * the compiler doesn't invoke the parent constructor automatically.
         */
         super("Hahaha");
         System.out.println("Constructor of child class");

   }
   void display(){
         System.out.println("Hello");
   }
   public static void main(String args[]){
         Subclass obj= new Subclass();
         obj.display();
   }
}
```

Output:

```
parameterized constructor of parent class
Constructor of child class
Hello
```

There are few important points to note in this example:
1) super()(or parameterized super must be the first statement in constructor otherwise you will get the compilation error: "Constructor call must be the first statement in a constructor"
2) When we explicitly placed super in the constructor, the java compiler didn't call the default no-arg constructor of parent class.

3) How to use super keyword in case of method overriding

When a child class declares a same method which is already present in the parent class then this is called method overriding. We will learn method overriding in the next tutorials of this series. For now you just need to remember this: When a child class overrides a method of parent class, then the call to the method from child class object always call the child class version of the method. However by using super keyword like this: super.method name you can call the method of parent class (the method which is overridden). In case of method overriding, these terminologies are used: Overridden method: The method of parent class Overriding method: The method of child class Lets take an example to understand this concept:

```
class Parentclass
{
   //Overridden method
   void display(){
         System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   //Overriding method
   void display(){
         System.out.println("Child class method");
   }
   void printMsg(){
         //This would call Overriding method
         display();
         //This would call Overridden method
         super.display();
   }
   public static void main(String args[]){
         Subclass obj= new Subclass();
         obj.printMsg();
   }
}
```

Output:

```
Child class method
Parent class method
```

What if the child class is not overriding any method: No need of super

When child class doesn't override the parent class method then we don't need to use the super keyword to call the parent class method. This is because in this case we have only one version of each method and child class has access to the

parent class methods so we can directly call the methods of parent class without using super.

```
class Parentclass
{
   void display(){
         System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   void printMsg(){
         /* This would call method of parent class,
          * no need to use super keyword because no other
          * method with the same name is present in this class
          */
         display();
   }
   public static void main(String args[]){
         Subclass obj= new Subclass();
         obj.printMsg();
   }
}
```

Output:

```
Parent class method
```

Method Overloading in Java with examples

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

let's get back to the point, when I say argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:
1. Number of parameters.
For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters.
For example:

```
add(int, int)
add(int, float)
```

3. Sequence of Data type of parameters.
For example:

```
add(int, float)
add(float, int)
```

Invalid case of method overloading:
When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```

Method overloading is an example of Static Polymorphism. We will discuss polymorphism and types of it in a separate tutorial.

Points to Note:
1. Static Polymorphism is also known as compile time binding or early binding.
2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Method Overloading examples

As discussed in the beginning of this guide, method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Lets see examples of each of these cases.

Argument list is also known as parameter list

Example 1: Overloading - Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
class Sample
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
       obj.disp('a');
       obj.disp('a',10);
   }
}
```

Output:

```
a
a 10
```

In the above example – method `disp()` is overloaded based on the number of parameters – We have two methods with the name `disp` but the parameters they have are different. Both are having different number of parameters.

Example 2: Overloading - Difference in data type of parameters

In this example, method disp() is overloaded based on the data type of parameters – We have two methods with the name disp(), one with parameter of char type and another method with the parameter of int type.

```
class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
```

```
    }
    public void disp(int c)
    {
        System.out.println(c );
    }
}

class Sample2
{
    public static void main(String args[])
    {
         DisplayOverloading2 obj = new DisplayOverloading2();
         obj.disp('a');
         obj.disp(5);
    }
}
```

Output:

```
a
5
```

Example3: Overloading - Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverloading3
{
   public void disp(char c, int num)
   {
        System.out.println("I'm the first definition of method disp");
   }
   public void disp(int num, char c)
   {
        System.out.println("I'm the second definition of method disp" );
   }
}
class Sample3
{
   public static void main(String args[])
   {
        DisplayOverloading3 obj = new DisplayOverloading3();
        obj.disp('x', 51 );
        obj.disp(52, 'y');
   }
}
```

Output:

```
I’m the first definition of method disp
I’m the second definition of method disp
```

Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

What it has to do with method overloading?
Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.
Lets take an example to see what I am talking here:

```
class Demo{
   void disp(int a, double b){
         System.out.println("Method A");
   }
   void disp(int a, double b, double c){
         System.out.println("Method B");
   }
   public static void main(String args[]){
         Demo obj = new Demo();
         /* I am passing float value as a second argument but
          * it got promoted to the type double, because there
          * wasn't any method having arg list as (int, float)
          */
         obj.disp(100, 20.67f);
   }
}
```

Output:

```
Method A
```

As you can see that I have passed the float value while calling the disp() method but it got promoted to the double type as there wasn’t any method with argument list as (int, float)

But this type promotion doesn’t always happen, lets see another example:

```
class Demo{
   void disp(int a, double b){
         System.out.println("Method A");
   }
   void disp(int a, double b, double c){
         System.out.println("Method B");
   }
   void disp(int a, float b){
```

```
        System.out.println("Method C");
    }
    public static void main(String args[]){
        Demo obj = new Demo();
        /* This time promotion won't happen as there is
         * a method with arg list as (int, float)
         */
        obj.disp(100, 20.67f);
    }
}
```

Output:

```
Method C
```

As you see that this time type promotion didn't happen because there was a method with matching argument type.

Type Promotion table:

The data type on the left side can be promoted to the any of the data type present in the right side of it.

```
byte → short → int → long
short → int → long
int → long → float → double
float → double
long → float → double
```

Lets see few Valid/invalid cases of method overloading

Case 1:

```
int mymethod(int a, int b, float c)
int mymethod(int var1, int var2, float var3)
```

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

```
int mymethod(int a, int b)
int mymethod(float var1, float var2)
```

Result: Perfectly fine. Valid case of overloading. Here data types of arguments are different.

Case 3:

```
int mymethod(int a, int b)
int mymethod(int num)
```

Result: Perfectly fine. Valid case of overloading. Here number of arguments are different.

Case 4:

```
float mymethod(int a, float b)
float mymethod(float var1, int var2)
```

Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

```
int mymethod(int a, int b)
float mymethod(int var1, int var2)
```

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn't matter while overloading a method.

Guess the answers before checking it at the end of programs:

Question 1 – return type, method name and argument list same.

```
class Demo
{
   public int myMethod(int num1, int num2)
   {
        System.out.println("First myMethod of class Demo");
        return num1+num2;
   }
   public int myMethod(int var1, int var2)
   {
        System.out.println("Second myMethod of class Demo");
        return var1-var2;
   }
}
class Sample4
{
   public static void main(String args[])
   {
        Demo obj1= new Demo();
        obj1.myMethod(10,10);
        obj1.myMethod(20,12);
   }
}
```

Answer:

It will throw a compilation error: More than one method with same name and argument list cannot be defined in a same class.

Question 2 – return type is different. Method name & argument list same.

```
class Demo2
{
   public double myMethod(int num1, int num2)
   {
       System.out.println("First myMethod of class Demo");
       return num1+num2;
   }
   public int myMethod(int var1, int var2)
   {
       System.out.println("Second myMethod of class Demo");
       return var1-var2;
   }
}
class Sample5
{
   public static void main(String args[])
   {
       Demo2 obj2= new Demo2();
       obj2.myMethod(10,10);
       obj2.myMethod(20,12);
   }
}
```

Answer:
It will throw a compilation error: More than one method with same name and argument list cannot be given in a class even though their return type is different. **Method return type doesn't matter in case of overloading.**

Method overriding in java with example

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The `Boy` class extends `Human` class. Both the classes have a common method `void eat()`. Boy class is giving its own

implementation to the `eat()` method or in other words it is overriding the `eat()` method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
class Human{
   //Overridden method
   public void eat()
   {
       System.out.println("Human is eating");
   }
}
class Boy extends Human{
   //Overriding method
   public void eat(){
       System.out.println("Boy is eating");
   }
   public static void main( String args[]) {
       Boy obj = new Boy();
       //This will call the child class version of eat()
       obj.eat();
   }
}
```

Output:

```
Boy is eating
```

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object. This process in

which call to the overridden method is resolved at runtime is known as dynamic method dispatch. Lets see an example to understand this:

```
class ABC{
   //Overridden method
   public void disp()
   {
         System.out.println("disp() method of parent class");
   }
}
class Demo extends ABC{
   //Overriding method
   public void disp(){
         System.out.println("disp() method of Child class");
   }
   public void newMethod(){
         System.out.println("new method of child class");
   }
   public static void main( String args[]) {
         /* When Parent class reference refers to the parent class object
          * then in this case overridden method (the method of parent class)
          *  is called.
          */
         ABC obj = new ABC();
         obj.disp();

         /* When parent class reference refers to the child class object
          * then the overriding method (method of child class) is called.
          * This is called dynamic method dispatch and runtime polymorphism
          */
         ABC obj2 = new Demo();
         obj2.disp();
   }
}
```

Output:

```
disp() method of parent class
disp() method of Child class
```

In the above example the call to the disp() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).

Note: In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class. In the above example the object `obj2` is calling the `disp()`. However if you try to call the `newMethod()` method (which has been newly declared in Demo class) using obj2 then you would give compilation error with the following message:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: The method xyz() is undefined for the type ABC
```

Rules of method overriding in Java

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class). The data types of the arguments and their sequence should exactly match.
2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier,because all of these three access modifiers are more restrictive than public.
   For e.g. This is **not allowed** as child class disp method is more restrictive(protected) than base class(public)

```
3.  class MyBaseClass{
4.      public void disp()
5.      {
6.          System.out.println("Parent class method");
7.      }
8.  }
9.  class MyChildClass extends MyBaseClass{
10.     protected void disp(){
11.         System.out.println("Child class method");
12.     }
13.     public static void main( String args[]) {
14.         MyChildClass obj = new MyChildClass();
15.         obj.disp();
16.     }
    }
```

   Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: Cannot reduce the visibility of the inherited method from MyBaseClass
```

   However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.

```
class MyBaseClass{
    protected void disp()
    {
        System.out.println("Parent class method");
    }
}
class MyChildClass extends MyBaseClass{
    public void disp(){
        System.out.println("Child class method");
    }
    public static void main( String args[]) {
        MyChildClass obj = new MyChildClass();
        obj.disp();
    }
```

```
}
```

Output:

```
Child class method
```

17. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
18. Overriding method (method of child class) can throw unchecked exceptions, regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.
19. Binding of overridden methods happen at runtime which is known as dynamic binding.
20. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is a abstract class.

Super keyword in Method Overriding

The super keyword is used for calling the parent class method/constructor. `super.myMethod()` calls the myMethod() method of base class while `super()` calls the constructor of base class. Let's see the use of super in method Overriding.
As we know that we we override a method in child class, then call to the method using child class object calls the overridden method. By using super we can call the overridden method as shown in the example below:

```
class ABC{
   public void myMethod()
   {
         System.out.println("Overridden method");
   }
}
class Demo extends ABC{
   public void myMethod(){
         //This will call the myMethod() of parent class
         super.myMethod();
         System.out.println("Overriding method");
   }
   public static void main( String args[]) {
         Demo obj = new Demo();
         obj.myMethod();
```

```
    }
}
```

Output:

```
Class ABC: mymethod()
Class Test: mymethod()
```

As you see using super keyword, we can access the overriden method.

Polymorphism in Java with example

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

Polymorphism is one of the OOPs feature that allows us to perform a single action in different ways. For example, lets say we have a class `Animal` that has a method `sound()`. Since this is a generic class so we can't give it a implementation like: Roar, Meow, Oink etc. We had to give a generic message.

```
public class Animal{
   ...
   public void sound(){
      System.out.println("Animal is making a sound");
   }
}
```

Now lets say we two subclasses of Animal class: `Horse` and `Cat` that extends (see Inheritance) `Animal` class. We can provide the implementation to the same method like this:

```
public class Horse extends Animal{
...
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
}
```

and

```
public class Cat extends Animal{
...
    @Override
    public void sound(){
        System.out.println("Meow");
    }
}
```

As you can see that although we had the common action for all subclasses `sound()` but there were different ways to do the same action. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways). It would not make any sense to just call the generic sound() method as each Animal has a different sound. Thus we can say that the action this method performs is based on the type of object.

What is polymorphism in programming?

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations. As we have seen in the above example that we have defined the method sound() and have the multiple implementations of it in the different-2 sub classes.
Which sound() method will be called is determined at runtime so the example we gave above is a **runtime polymorphism example**.

Types of polymorphism and method overloading & overriding are covered in the separate tutorials. You can refer them here:
1. Method Overloading in Java – This is an example of compile time (or static polymorphism)
2. Method Overriding in Java – This is an example of runtime time (or dynamic polymorphism)
3. Types of Polymorphism – Runtime and compile time – This is our next tutorial where we have covered the types of polymorphism in detail. I would recommend you to go though method overloading and overriding before going though this topic.

Lets write down the complete code of it:

Example 1: Polymorphism in Java

Runtime Polymorphism example:
Animal.java

```
public class Animal{
   public void sound(){
       System.out.println("Animal is making a sound");
   }
}
```

Horse.java

```
class Horse extends Animal{
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
    public static void main(String args[]){
        Animal obj = new Horse();
        obj.sound();
    }
}
```

Output:

```
Neigh
```

Cat.java

```
public class Cat extends Animal{
    @Override
    public void sound(){
        System.out.println("Meow");
    }
    public static void main(String args[]){
        Animal obj = new Cat();
        obj.sound();
    }
}
```

Output:

```
Meow
```

Example 2: Compile time Polymorphism

Method Overloading on the other hand is a compile time polymorphism example.

```
class Overload
{
    void demo (int a)
    {
        System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
        System.out.println ("a and b: " + a + "," + b);
    }
    double demo(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}
class MethodOverloading
{
    public static void main (String args [])
```

```
    {
        Overload Obj = new Overload();
        double result;
        Obj .demo(10);
        Obj .demo(10, 20);
        result = Obj .demo(5.5);
        System.out.println("O/P : " + result);
    }
}
```

Here the method demo() is overloaded 3 times: first method has 1 int parameter, second method has 2 int parameters and third one is having double parameter. Which method is to be called is determined by the arguments we pass while calling methods. This happens at ~~runtime~~ compile time so this type of polymorphism is known as compile time polymorphism.

Output:

```
a: 10
a and b: 10,20
double a: 5.5
O/P : 30.25
```

Types of polymorphism in java- Runtime and Compile time polymorphism

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

In the last tutorial we discussed Polymorphism in Java. In this guide we will see **types of polymorphism**. There are two types of polymorphism in java:
1) **Static Polymorphism** also known as compile time polymorphism
2) **Dynamic Polymorphism** also known as runtime polymorphism

Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.
Method Overloading: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters. We have already discussed Method overloading here: If you didn't read that guide, refer: Method Overloading in Java

Example of static Polymorphism

Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

```
class SimpleCalculator
{
    int add(int a, int b)
    {
         return a+b;
    }
    int  add(int a, int b, int c)
    {
         return a+b+c;
    }
}
public class Demo
{
   public static void main(String args[])
   {
           SimpleCalculator obj = new SimpleCalculator();
       System.out.println(obj.add(10, 20));
       System.out.println(obj.add(10, 20, 30));
   }
}
```

Output:

```
30
60
```

Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism. I have already discussed method overriding in detail in a separate tutorial, refer it: Method Overriding in Java.

Example

In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that

determines which version of the method would be called (not the type of reference).

To understand the concept of overriding, you should have the basic knowledge of inheritance in Java.

```
class ABC{
   public void myMethod(){
         System.out.println("Overridden Method");
   }
}
public class XYZ extends ABC{

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

Output:

```
Overriding Method
```

When an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time.
Since both the classes, child class and parent class have the same method `animalSound`. Which version of the method(child class or parent class) will be called is determined at runtime by JVM.

Few more overriding examples:

```
ABC obj = new ABC();
obj.myMethod();
// This would call the myMethod() of parent class ABC

XYZ obj = new XYZ();
obj.myMethod();
// This would call the myMethod() of child class XYZ

ABC obj = new XYZ();
obj.myMethod();
// This would call the myMethod() of child class XYZ
```

In the third case the method of child class is to be executed because which method is to be executed is determined by the type of object and since the object belongs to the child class, the child class version of myMethod() is called.

Static and dynamic binding in java

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

Association of method call to the method body is known as binding. There are two types of binding: **Static Binding** that happens at compile time and **Dynamic Binding** that happens at runtime. Before I explain static and dynamic binding in java, lets see few terms that will help you understand this concept better.

What is reference and object?

```
class Human{
....
}
class Boy extends Human{
   public static void main( String args[]) {
        /*This statement simply creates an object of class
         *Boy and assigns a reference of Boy to it*/
        Boy obj1 = new Boy();

        /* Since Boy extends Human class. The object creation
         * can be done in this way. Parent class reference
         * can have child class reference assigned to it
         */
        Human obj2 = new Boy();
   }
}
```

Static and Dynamic Binding in Java

As mentioned above, association of method definition to the method call is known as binding. There are two types of binding: Static binding and dynamic binding. Lets discuss them.

Static Binding or Early Binding

The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private and final methods is compile-time. **Why?** The reason is that the these method cannot be overridden and the type of the class is determined at the compile time. Lets see an example to understand this:

Static binding example

Here we have two classes Human and Boy. Both the classes have same method walk() but the method is static, which means it cannot be overriden so even though I have used the object of Boy class while creating object obj, the parent class method is called by it. Because the reference is of Human type (parent class). So whenever a binding of static, private and final methods happen, type of the class is determined by the compiler at compile time and the binding happens then and there.

```
class Human{
   public static void walk()
   {
        System.out.println("Human walks");
   }
}
class Boy extends Human{
   public static void walk(){
        System.out.println("Boy walks");
   }
   public static void main( String args[]) {
        /* Reference is of Human type and object is
         * Boy type
         */
        Human obj = new Boy();
        /* Reference is of HUman type and object is
         * of Human type.
         */
        Human obj2 = new Human();
        obj.walk();
        obj2.walk();
   }
}
```

Output:

```
Human walks
Human walks
```

Dynamic Binding or Late Binding

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object**determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.

Dynamic binding example

This is the same example that we have seen above. The only difference here is that in this example, overriding is actually happening since these methods are **not** static, private and final. In case of overriding the call to the overriden method is determined at runtime by the type of object thus late binding happens. Lets see an example to understand this:

```
class Human{
   //Overridden Method
   public void walk()
   {
       System.out.println("Human walks");
   }
}
class Demo extends Human{
   //Overriding Method
   public void walk(){
       System.out.println("Boy walks");
   }
   public static void main( String args[]) {
       /* Reference is of Human type and object is
        * Boy type
        */
       Human obj = new Demo();
       /* Reference is of HUman type and object is
        * of Human type.
        */
       Human obj2 = new Human();
       obj.walk();
       obj2.walk();
   }
}
```

Output:

```
Boy walks
Human walks
```

As you can see that the output is different than what we saw in the static binding example, because in this case while creation of object obj the type of the object is determined as a Boy type so method of Boy class is called. Remember the type of the object is determined at the runtime.

Static Binding vs Dynamic Binding

Lets discuss the **difference between static and dynamic binding in Java**.

1. Static binding happens at compile-time while dynamic binding happens at runtime.
2. Binding of private, static and final methods always happen at compile time since these methods cannot be overridden. When the method overriding is actually happening and the reference of parent type is assigned to the object of child class type then such binding is resolved during runtime.
3. The binding of overloaded methods is static and the binding of overridden methods is dynamic.

Abstract Class in Java with example

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

A class that is declared using "**abstract**" keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods. In this guide we will learn what is a abstract class, why we use it and what are the rules that we must remember while working with it in Java.

An abstract class can not be **instantiated**, which means you are not allowed to create an **object**of it. Why? We will discuss that later in this guide.

Why we need an abstract class?

Lets say we have a class `Animal` that has a method `sound()` and the subclasses(see inheritance) of it like `Dog`, `Lion`, `Horse`, `Cat` etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like `Lion` class will say "Roar" in this method and `Dog` class will say "Woof".

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the `Animal` class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Abstract class Example

```
//abstract parent class
abstract class Animal{
   //abstract method
   public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

   public void sound(){
         System.out.println("Woof");
   }
   public static void main(String args[]){
         Animal obj = new Dog();
         obj.sound();
   }
}
```

Output:

```
Woof
```

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//Declaration using abstract keyword
abstract class A{
   //This is abstract method
   abstract void myMethod();

   //This is concrete method with body
   void anotherMethod(){
      //Does something
   }
}
```

Rules

Note 1: As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Note 2: Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

Note 3: If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

Do you know? Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.

Interfaces on the other hand are used for 100% abstraction (See more about abstraction here).
You may also want to read this: Difference between abstract class and Interface in Java

Why can’t we create the object of an abstract class?

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen?There would be no actual implementation of the method to invoke.
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

Example to demonstrate that object creation of abstract class is not allowed

As discussed above, we cannot instantiate an abstract class. This program throws a compilation error.

```
abstract class AbstractDemo{
   public void myMethod(){
      System.out.println("Hello");
   }
   abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

   public void anotherMethod() {
         System.out.print("Abstract method");
   }
   public static void main(String args[])
   {
      //error: You can't create object of it
      AbstractDemo obj = new AbstractDemo();
      obj.anotherMethod();
   }
}
```

Output:

```
Unresolved compilation problem: Cannot instantiate the type AbstractDemo
```

Note: The class that extends the abstract class, have to implement all the abstract methods of it, else you have to declare that class abstract as well.

Abstract class vs Concrete class

A class which is not abstract is referred as **Concrete class**. In the above example that we have seen in the beginning of this guide, `Animal` is a abstract class and `Cat`, `Dog` & `Lion` are concrete classes.

Key Points:

1. An abstract class has no use until unless it is extended by some other class.
2. If you declare an **abstract method** in a class then you must declare the class abstract as well. you can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. It can have non-abstract method (concrete) as well.

I have covered the rules and examples of abstract methods in a separate tutorial, You can find the guide here: Abstract method in Java
For now lets just see some basics and example of abstract method.
1) Abstract method has no body.
2) Always end the declaration with a **semicolon**(;).
3) It must be overridden. An abstract class must be extended and in a same way abstract method must be overridden.
4) A class has to be declared abstract to have abstract methods.

Note: The class which is extending abstract class must override all the abstract methods.

Example of Abstract class and method

```
abstract class MyClass{
   public void disp(){
     System.out.println("Concrete method of parent class");
   }
   abstract public void disp2();
}

class Demo extends MyClass{
   /* Must Override this method while extending
    * MyClas
    */
   public void disp2()
   {
       System.out.println("overriding abstract method");
   }
   public static void main(String args[]){
       Demo obj = new Demo();
       obj.disp2();
   }
}
```

Output:

```
overriding abstract method
```

Abstract method in Java with examples

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

A method without body (no implementation) is known as abstract method. A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well. In the last tutorial we discussed Abstract class, if you have not yet checked it out

read it here: Abstract class in Java, before reading this guide.
This is how an abstract method looks in java:

```
public abstract int myMethod(int n1, int n2);
```

As you see this has no body.

Rules of Abstract Method

1. Abstract methods don't have body, they just have method signature as shown above.
2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn't need to have an abstract method compulsory.
3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

Example 1: abstract method in an abstract class

```
//abstract class
abstract class Sum{
   /* These two are abstract methods, the child class
    * must implement these methods
    */
   public abstract int sumOfTwo(int n1, int n2);
   public abstract int sumOfThree(int n1, int n2, int n3);

   //Regular method
   public void disp(){
         System.out.println("Method of class Sum");
   }
}
//Regular class extends abstract class
class Demo extends Sum{

   /* If I don't provide the implementation of these two methods, the
    * program will throw compilation error.
    */
   public int sumOfTwo(int num1, int num2){
         return num1+num2;
   }
   public int sumOfThree(int num1, int num2, int num3){
         return num1+num2+num3;
   }
   public static void main(String args[]){
         Sum obj = new Demo();
         System.out.println(obj.sumOfTwo(3, 7));
         System.out.println(obj.sumOfThree(4, 3, 19));
```

```
        obj.disp();
    }
}
```

Output:

```
10
26
Method of class Sum
```

Example 2: abstract method in interface

All the methods of an interface are public abstract by default. You cannot have concrete (regular methods with body) methods in an interface.

```
//Interface
interface Multiply{
    //abstract methods
    public abstract int multiplyTwo(int n1, int n2);

    /* We need not to mention public and abstract in interface
     * as all the methods in interface are
     * public and abstract by default so the compiler will
     * treat this as
     * public abstract multiplyThree(int n1, int n2, int n3);
     */
    int multiplyThree(int n1, int n2, int n3);

    /* Regular (or concrete) methods are not allowed in an interface
     * so if I uncomment this method, you will get compilation error
     * public void disp(){
     *    System.out.println("I will give error if u uncomment me");
     * }
     */
}

class Demo implements Multiply{
    public int multiplyTwo(int num1, int num2){
        return num1*num2;
    }
    public int multiplyThree(int num1, int num2, int num3){
        return num1*num2*num3;
    }
    public static void main(String args[]){
        Multiply obj = new Demo();
        System.out.println(obj.multiplyTwo(3, 7));
        System.out.println(obj.multiplyThree(1, 9, 0));
    }
}
```

Output:

```
21
0
```

Reference:
Abstract method javadoc

Interface in java with example programs

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

In the last tutorial we discussed abstract class which is used for achieving partial abstraction. Unlike abstract class an interface is used for full abstraction. Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user(See: Abstraction). In this guide, we will cover **what is an interface in java**, why we use it and what are rules that we must follow while using interfaces in Java Programming.

What is an interface in Java?

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body, see: Java abstract method). Also, the variables declared in an interface are public, static & final by default. We will cover this in detail, later in this guide.

What is the use of interface in Java?

As mentioned above they are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

Syntax:
Interfaces are declared by specifying a keyword "interface". E.g.:

```
interface MyInterface
{
   /* All the methods are public abstract by default
    * As you see they have no body
    */
   public void method1();
   public void method2();
}
```

Example of an Interface in Java

This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.

Do you know? class `implements` interface but an interface `extends` another interface.

```
interface MyInterface
{
   /* compiler will treat them as:
    * public abstract void method1();
    * public abstract void method2();
    */
   public void method1();
   public void method2();
}
class Demo implements MyInterface
{
   /* This class must have to implement both the abstract methods
    * else you will get compilation error
    */
   public void method1()
   {
         System.out.println("implementation of method1");
   }
   public void method2()
   {
         System.out.println("implementation of method2");
   }
   public static void main(String arg[])
   {
         MyInterface obj = new Demo();
         obj.method1();
   }
}
```

Output:

```
implementation of method1
```

You may also like to read: Difference between abstract class and interface

Interface and Inheritance

As discussed above, an interface can not implement another interface. It has to extend the other interface. See the below example where we have two interfaces Inf1 and Inf2. Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf2 as well as Inf1.

Learn more about inheritance here: Java Inheritance

```
interface Inf1{
   public void method1();
}
interface Inf2 extends Inf1 {
   public void method2();
}
public class Demo implements Inf2{
   /* Even though this class is only implementing the
    * interface Inf2, it has to implement all the methods
    * of Inf1 as well because the interface Inf2 extends Inf1
    */
    public void method1(){
        System.out.println("method1");
    }
    public void method2(){
        System.out.println("method2");
    }
    public static void main(String args[]){
        Inf2 obj = new Demo();
        obj.method2();
    }
}
```

In this program, the class Demo only implements interface Inf2, however it has to provide the implementation of all the methods of interface Inf1 as well, because interface Inf2 extends Inf1.

Tag or Marker interface in Java

An empty interface is known as tag or marker interface. For example Serializable, EventListener, Remote(java.rmi.Remote) are tag interfaces. These interfaces do not have any field and methods in it. Read more about it here.

Nested interfaces

An interface which is declared inside another interface or class is called nested interface. They are also known as inner interface. For example Entry interface in collections framework is declared inside Map interface, that's why we don' use it directly, rather we use it like this: Map.Entry.

Key points: Here are the key points to remember about interfaces:
1) We can't instantiate an interface in java. That means we cannot create the object of an interface

2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.

3) `implements` keyword is used by classes to implement an interface.

4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.

5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.

6) Interface cannot be declared as private, protected or transient.

7) All the interface methods are by default **abstract and public**.

8) Variables declared in interface are **public, static and final** by default.

```
interface Try
{
   int a=10;
   public int a=10;
   public static final int a=10;
   final int a=10;
   static int a=0;
}
```

All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

```
interface Try
{
      int x;//Compile-time error
}
```

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the

value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

```
class Sample implements Try
{
  public static void main(String args[])
  {
     x=20; //compile time error
  }
}
```

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A **class** can implement any **number of interfaces**.

13) If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
   public void aaa();
}
interface B
{
   public void aaa();
}
class Central implements A,B
{
   public void aaa()
   {
         //Any Code here
   }
   public static void main(String args[])
   {
         //Statements
    }
}
```

14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A
{
   public void aaa();
}
interface B
{
   public int aaa();
}
```

```
class Central implements A,B
{

   public void aaa() // error
   {
   }
   public int aaa() // error
   {
   }
   public static void main(String args[])
   {

   }
}
```

15) Variable names conflicts can be resolved by interface name.

```
interface A
{
    int x=10;
}
interface B
{
    int x=100;
}
class Hello implements A,B
{
    public static void Main(String args[])
    {
       /* reference to x is ambiguous both variables are x
        * so we are using interface name to resolve the
        * variable
        */
       System.out.println(x);
       System.out.println(A.x);
       System.out.println(B.x);
    }
}
```

Advantages of interface in java:

Advantages of using interfaces are as follows:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

Encapsulation in Java with example

BY CHAITANYA SINGH | FILED UNDER: OOPS CONCEPT

Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation. In this guide we will see how to do encapsulation in java program, if you are looking for a real-life example of encapsulation then refer this guide: OOPs features explained using real-life examples.

For other OOPs topics such as inheritance and polymorphism, refer OOPs concepts

Lets get back to the topic.

What is encapsulation?

The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.

However if we setup public getter and setter methods to update (for example `void setSSN(int ssn)`)and read (for example `int getSSN()`) the private data fields then the outside class can access those private data fields via public methods.

This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That's why encapsulation is known as **data hiding.** Lets see an example to understand this concept better.

Example of Encapsulation in Java

How to implement encapsulation in java:
1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
2) Have getter and setter methods in the class to set and get the values of the fields.

```
class EncapsulationDemo{
    private int ssn;
    private String empName;
    private int empAge;
```

```
    //Getter and Setter methods
    public int getEmpSSN(){
        return ssn;
    }

    public String getEmpName(){
        return empName;
    }

    public int getEmpAge(){
        return empAge;
    }

    public void setEmpAge(int newValue){
        empAge = newValue;
    }

    public void setEmpName(String newValue){
        empName = newValue;
    }

    public void setEmpSSN(int newValue){
        ssn = newValue;
    }
}
public class EncapsTest{
    public static void main(String args[]){
         EncapsulationDemo obj = new EncapsulationDemo();
         obj.setEmpName("Mario");
         obj.setEmpAge(32);
         obj.setEmpSSN(112233);
         System.out.println("Employee Name: " + obj.getEmpName());
         System.out.println("Employee SSN: " + obj.getEmpSSN());
         System.out.println("Employee Age: " + obj.getEmpAge());
    }
}
```

Output:

```
Employee Name: Mario
Employee SSN: 112233
Employee Age: 32
```

In above example all the three data members (or data fields) are private(see: Access Modifiers in Java) which cannot be accessed directly. These fields can be accessed via public methods only.
Fields empName, ssn and empAge are made hidden data fields using encapsulation technique of OOPs.

Advantages of encapsulation

1. It improves maintainability and flexibility and re-usability: for e.g. In the above code the implementation code of `void setEmpName(String name)` and `String getEmpName()` can be changed at any point of time. Since the implementation is purely hidden for outside classes they would still be accessing the private field empName using the same methods (`setEmpName(String name)` and `getEmpName()`). Hence the code can be maintained at any point of time without breaking the classes that uses the code. This improves the re-usability of the underlying class.
2. The fields can be made read-only (If we don't define setter methods in the class) or write-only (If we don't define the getter methods in the class). For e.g. If we have a field(or variable) that we don't want to be changed so we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.
3. User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call set method and to read a field call get method but what these set and get methods are doing is purely hidden from them.

Encapsulation is also known as "**data Hiding**".