**Java Concepts**

* **Comparable and Comparator**

Let’s see how we can sort primitive types or Object array and list with a simple program.

package com.journaldev.sort;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class JavaObjectSorting {

public static void main(String[] args) {

//sort primitives array like int array

int[] intArr = {5,9,1,10};

Arrays.sort(intArr);

System.out.println(Arrays.toString(intArr));

//sorting String array

String[] strArr = {"A", "C", "B", "Z", "E"};

Arrays.sort(strArr);

System.out.println(Arrays.toString(strArr));

//sorting list of objects of Wrapper classes

List<String> strList = new ArrayList<String>();

strList.add("A");

strList.add("C");

strList.add("B");

strList.add("Z");

strList.add("E");

Collections.sort(strList);

for(String str: strList) System.out.print(" "+str);

}

}

**Output:**

[1, 5, 9, 10]

[A, B, C, E, Z]

A B C E Z

* **Comparable Interface:**

Java Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object). It provides single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be rollno, name, age or anything else.

**public int compareTo(Object obj):** is used to compare the current object with the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

**public void p(List list):** is used to sort the elements of List. List elements must be of Comparable type.

**Example :**

package com.incredible.comparable;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

class Employee implements Comparable<Employee> {

int id;

String name;

int salary;

int age;

public Employee(int id, String name, int salary, int age) {

this.id = id;

this.name = name;

this.salary = salary;

this.age = age;

}

@Override

public int compareTo(Employee emp) {

if (age == emp.age)

return 0;

else if (age > emp.age)

return 1;

else

return -1;

}

}

public class ComparableDemo {

public static void main(String args[]) {

List<Employee> e = new ArrayList<Employee>();

e.add(new Employee(10, "Amit", 25000, 30));

e.add(new Employee(20, "Ajit", 65000, 28));

e.add(new Employee(30, "Amey", 55000, 2));

Collections.sort(e);

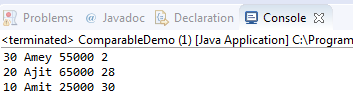
for(Employee e1:e){

System.out.println(e1.id+" "+e1.name+" "+e1.salary+" "+e1.age);

}

}

}



* **Comparator Interface:**

Java Comparator interface is used to order the objects of user-defined class.This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

**public int compare(Object obj1,Object obj2):** compares the first object with second object.

**public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

**Example :**

**package** com.incredible.comparator;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.List;

**class** Employee {

**int** id;

String name;

**int** salary;

**int** age;

**public** Employee(**int** id, String name, **int** salary, **int** age) {

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

**this**.age = age;

}

}

**class** AgeComparator **implements** Comparator {

@Override

**public** **int** compare(Object arg0, Object arg1) {

Employee e1 = (Employee) arg0;

Employee e2 = (Employee) arg1;

**if** (e1.age == e2.age)

**return** 0;

**else** **if** (e1.age > e2.age)

**return** 1;

**else**

**return** -1;

}

}

**class** NameComparator **implements** Comparator {

**public** **int** compare(Object o1, Object o2) {

Employee e1 = (Employee) o1;

Employee e2 = (Employee) o2;

**return** e1.name.compareTo(e2.name);

}

}

**public** **class** ComparatorDemo {

**public** **static** **void** main(String args[]) {

List<Employee> e = **new** ArrayList<Employee>();

e.add(**new** Employee(10, "Amit", 25000, 30));

e.add(**new** Employee(20, "Ajit", 65000, 28));

e.add(**new** Employee(30, "Amey", 55000, 2));

Collections.*sort*(e, **new** AgeComparator());

System.***out***.println("Sorting by Age....");

**for** (Employee e1 : e) {

System.***out***.println(e1.id + " " + e1.name + " " + e1.salary + " " + e1.age);

}

System.***out***.println("Sorting by Name....");

Collections.*sort*(e, **new** NameComparator());

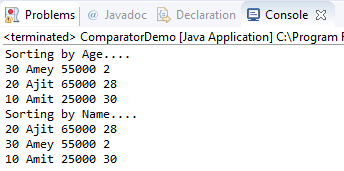
**for** (Employee e1 : e) {

System.***out***.println(e1.id + " " + e1.name + " " + e1.salary + " " + e1.age);

}

}

}



* **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, sees: Java abstract method). Also, the variables declared in an interface are public, static & final by default.

**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 interface 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();**

**}**

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

**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.

**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);**

**}**

**}**

* **Abstract Class in Java**

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.

An abstract class cannot 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?**

Let’s 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:**

An abstract class has no use until unless it is extended by some other class.

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.

It can have non-abstract method (concrete) as well.

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

* **Method overriding in java**

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**.

**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**

**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.

**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)

**class MyBaseClass{**

**public void disp()**

**{**

**System.out.println("Parent class method");**

**}**

**}**

**class MyChildClass extends MyBaseClass{**

**protected void disp(){**

**System.out.println("Child class method");**

**}**

**public static void main( String args[]) {**

**MyChildClass obj = new MyChildClass();**

**obj.disp();**

**}**

**}**

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

**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.

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.

Binding of overridden methods happen at runtime which is known as dynamic binding.

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.

* **Exception handling in Method overriding**

**Rule:** An overriding method (the method of child class) can throw any unchecked exceptions, regardless of whether the overridden method (method of base class) throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw those checked exceptions, which have less scope than the exception(s) declared in the overridden method.

Let’s understand the above explanation with the help of few examples:

**Scenario 1:** If base class doesn’t throw any exception but child class throws an unchecked exception.

In this example class Room is overriding the method color (). The overridden method is not throwing any exception however the overriding method is throwing an unchecked exception (NullPointerException). Upon compilation code ran successfully.

**class Building {**

**void color()**

**{**

**System.out.println("Blue");**

**}**

**}**

**class Room extends Building{**

**//It throws an unchecked exception**

**void color() throws NullPointerException**

**{**

**System.out.println("White");**

**}**

**public static void main(String args[]){**

**Building obj = new Room();**

**obj.color();**

**}**

**}**

Output:

White

**Scenario 2:** If base class doesn’t throw any exception but child class throws an checked exception

**import java.io.\*;**

**class Building {**

**void color()**

**{**

**System.out.println("Blue");**

**}**

**}**

**class Room extends Building{**

**void color() throws IOException**

**{**

**System.out.println("White");**

**}**

**public static void main(String args[]){**

**Building obj = new Room();**

**try{**

**obj.color();**

**}catch(Exception e){**

**System.out.println(e);**

**}**

**}**

**}**

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Exception IOException is not compatible with throws clause in Building.color()

The above code is having a compilation error: Because the overriding method (child class method) cannot throw a checked exception if the overridden method(method of base class) is not throwing an exception.

**Scenario 3:** When base class and child class both throws a checked exception

**import java.io.\*;**

**class Building {**

**void color() throws IOException**

**{**

**System.out.println("Blue");**

**}**

**}**

**class Room extends Building{**

**void color() throws IOException**

**{**

**System.out.println("White");**

**}**

**public static void main(String args[]){**

**Building obj = new Room();**

**try{**

**obj.color();**

**}catch(Exception e){**

**System.out.println(e);**

**}**

**}**

**}**

Output:

White

The code ran fine because color() method of child class is NOT throwing a checked exception with scope broader than the exception declared by color() method of base class.

**Scenario 4**: When child class method is throwing broder checked exception compared to the same method of base class

**package beginnersbook.com;**

**import java.io.\*;**

**class Building {**

**void color() throws IOException**

**{**

**System.out.println("Blue");**

**}**

**}**

**class Room extends Building{**

**void color() throws Exception**

**{**

**System.out.println("White");**

**}**

**public static void main(String args[]){**

**Building obj = new Room();**

**try{**

**obj.color();**

**}catch(Exception e){**

**System.out.println(e);**

**}**

**}**

**}**

Output:

Compilation error because the color() method of child class is throwing Exception which has a broader scope than the exception thrown by method color() of parent class.

* **Overloading vs Overriding in Java**

1. Overloading happens at [compile-time](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) while Overriding happens at [runtime](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/): The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime.
2. Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
3. The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
4. [Static binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overloaded methods and [dynamic binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overridden/overriding methods.
5. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
6. private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
7. Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type.
8. Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

### Overloading example

//A class for adding upto 5 numbers

class Sum

{

int add(int n1, int n2)

{

return n1+n2;

}

int add(int n1, int n2, int n3)

{

return n1+n2+n3;

}

int add(int n1, int n2, int n3, int n4)

{

return n1+n2+n3+n4;

}

int add(int n1, int n2, int n3, int n4, int n5)

{

return n1+n2+n3+n4+n5;

}

public static void main(String args[])

{

Sum obj = new Sum();

System.out.println("Sum of two numbers: "+obj.add(20, 21));

System.out.println("Sum of three numbers: "+obj.add(20, 21, 22));

System.out.println("Sum of four numbers: "+obj.add(20, 21, 22, 23));

System.out.println("Sum of five numbers: "+obj.add(20, 21, 22, 23, 24));

}

}

Output:

Sum of two numbers: 41

Sum of three numbers: 63

Sum of four numbers: 86

Sum of five numbers: 110

Here we have 4 versions of same method add. We are overloading the method add() here.

### Overriding example

package beginnersbook.com;

class CarClass

{

public int speedLimit()

{

return 100;

}

}

class Ford extends CarClass

{

public int speedLimit()

{

return 150;

}

public static void main(String args[])

{

CarClass obj = new Ford();

int num= obj.speedLimit();

System.out.println("Speed Limit is: "+num);

}

}

Output:

Speed Limit is: 150

* **Reflection API in Java**

Java Reflection is a process of examining or modifying the run time behavior of a class at run time.The java.lang.Class class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.The java.lang and java.lang.reflect packages provide classes for java reflection.

### Commonly used methods of Class class:

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public String getName() | returns the class name |
| 2) public static Class forName(String className)throws ClassNotFoundException | loads the class and returns the reference of Class class. |
| 3) public Object newInstance()throws InstantiationException,IllegalAccessException | creates new instance. |
| 4) public boolean isInterface() | checks if it is interface. |
| 5) public boolean isArray() | checks if it is array. |
| 6) public boolean isPrimitive() | checks if it is primitive. |
| 7) public Class getSuperclass() | returns the superclass class reference. |
| 8) public Field[] getDeclaredFields()throws SecurityException | returns the total number of fields of this class. |
| 9) public Method[] getDeclaredMethods()throws SecurityException | returns the total number of methods of this class. |
| 10) public Constructor[] getDeclaredConstructors()throws SecurityException | returns the total number of constructors of this class. |
| 11) public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException | returns the method class instance. |

* **Strings:**

String represents sequence of characters enclosed within the double quotes. “abc”, “JAVA”, “123”, “A” are some examples of strings. In many languages, strings are treated as character arrays. But In java, strings are treated as objects. To create and manipulate the strings, Java provides three classes.

1) java.lang.String (From JDK 1.0)

2) java.lang.StringBuffer (From JDK 1.5)

3) java.lang.StringBuilder (From JDK 1.5)

1) All these three classes are members of java.lang package and they are final classes. That means you can’t create subclasses to these three classes.

2) All three classes implement Serializable and CharSequence interface.

3) java.lang.String objects are immutable in java. That is, once you create String objects, you can’t modify them. Whenever you try to modify the existing String object, a new String object is created with modifications. Existing object is not at all altered. Where as java.lang.StringBuffer and java.lang.StringBuilder objects are mutable. That means, you can perform modifications to existing objects

4) Only String and StringBuffer objects are thread safe. StringBuilder objects are not thread safe. So whenever you want immutable and thread safe string objects, use java.lang.String class and whenever you want mutable as well as thread safe string objects then use java.lang.StringBuffer class.

5) In all three classes, toString() method is overrided. So. whenever you use reference variables of these three types, they will return contents of the objects not physical address of the objects.

6) hashCode() and equals() methods are overrided only in java.lang.String class but not in java.lang.StringBuffer and java.lang.StringBuilder classes.

7) There is no reverse() and delete() methods in String class. But, StringBuffer and StringBuilder have reverse() and delete() methods.

8) In case of String class, you can create the objects without new operator. But in case of StringBuffer and StringBuilder class, you have to use new operator to create the objects.

**How The Strings Are Stored In The Memory?**

While storing the string objects in the memory also, they are specially treated by the Java. After reading this article, you will come to know how they are specially treated in the memory.

We all know that JVM divides the allocated memory to a Java program into two parts. one is Stack and another one is heap. Stack is used for execution purpose and heap is used for storage purpose. In that heap memory, JVM allocates some memory specially meant for string literals. This part of the heap memory is called String Constant Pool.

Whenever you create a string object using string literal, that object is stored in the string constant pool and whenever you create a string object using new keyword, such object is stored in the heap memory.

For example, when you create string objects like below, they will be stored in the String Constant Pool.

**String s1 = "abc";**

**String s2 = "xyz";**

**String s3 = "123";**

**String s4 = "A";**

And when you create string objects using new keyword like below, they will be stored in the heap memory.

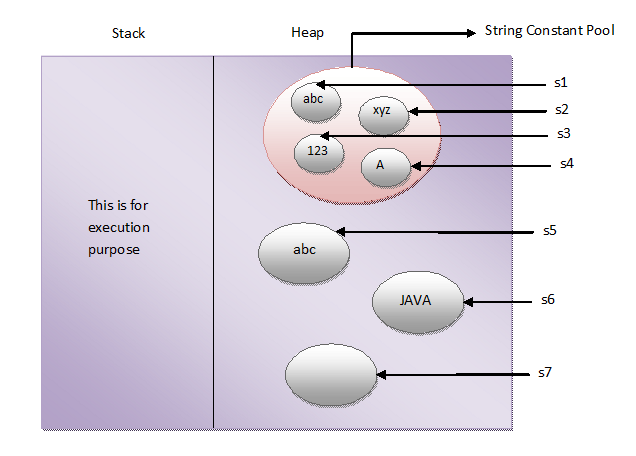
**String s5 = new String("abc");**

**char[] c = {'J', 'A', 'V', 'A'};**

**String s6 = new String(c);**

**String s7 = new String(new StringBuffer());**

This is how String Constant Pool looks like in the memory.



One more interesting thing about String Constant Pool is that, pool space is allocated to an object depending upon its content. There will be no two objects in the pool having the same content.

This is what happens when you create string objects using string literal,

“When you create a string object using string literal, JVM first checks the content of to be created object. If there exist an object in the pool with the same content, then it returns the reference of that object. It doesn’t create new object. If the content is different from the existing objects then only it creates new object.”

But, when you create string objects using new keyword, a new object is created whether the content is same or not.

This can be proved by using “==” operator. As “==” operator returns true if two objects have same physical address in the memory otherwise it will return false. In the below example, s1 and s2 are created using string literal “abc”. So, s1 == s2 returns true. Whereas s3 and s4 is created using new operator having the same content. But, s3 == s4 returns false.

**public class StringExamples**

**{**

**public static void main(String[] args)**

**{**

**//Creating string objects using literals**

**String s1 = "abc";**

**String s2 = "abc";**

**System.out.println(s1 == s2); //Output : true**

**//Creating string objects using new operator**

**String s3 = new String("abc");**

**String s4 = new String("abc");**

**System.out.println(s3 == s4); //Output : false**

**if**(s3.equals(s4))

System.***out***.println("true"); **//Output : true**

**}**

**}**

In simple words, there cannot be two string objects with same content in the string constant pool. But, there can be two string objects with the same content in the heap memory.

* **Inheritance (IS-A) vs. Composition (HAS-A) Relationship**

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the vimplementation of inheritance (IS-A relationship), or object composition (HAS-A relationship). Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition.

**IS-A Relationship:**

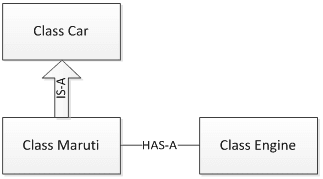
In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

It is a key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

**HAS-A Relationship:**

Composition(HAS-A) simply mean the use of instance variables that are references to other objects. For example Maruti has Engine, or House has Bathroom.

Let’s understand these concepts with an example of Car class.



[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php)

1. **package** relationships;
2. **class** Car {
3. // Methods implementation and class/Instance members
4. **private** String color;
5. **private** **int** maxSpeed;
6. **public** **void** carInfo(){
7. System.out.println("Car Color= "+color + " Max Speed= " + maxSpeed);
8. }
9. **public** **void** setColor(String color) {
10. **this**.color = color;
11. }
12. **public** **void** setMaxSpeed(**int** maxSpeed) {
13. **this**.maxSpeed = maxSpeed;
14. }
15. }

As shown above, Car class has a couple of instance variable and few methods. Maruti is a specific type of Car which extends Car class means Maruti IS-A Car.

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php)

1. **class** Maruti **extends** Car{
2. //Maruti extends Car and thus inherits all methods from Car (except final and static)
3. //Maruti can also define all its specific functionality
4. **public** **void** MarutiStartDemo(){
5. Engine MarutiEngine = **new** Engine();
6. MarutiEngine.start();
7. }
8. }

Maruti class uses Engine object’s start() method via composition. We can say that Maruti class HAS-A Engine.

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php)

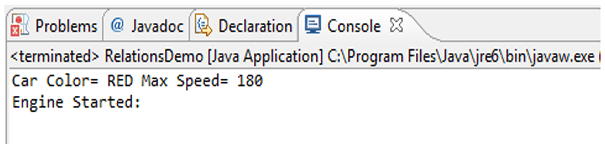
1. **package** relationships;
2. **public** **class** Engine {
3. **public** **void** start(){
4. System.out.println("Engine Started:");
5. }
6. **public** **void** stop(){
7. System.out.println("Engine Stopped:");
8. }
9. }

RelationsDemo class is making object of Maruti class and initialized it. Though Maruti class does not have setColor(), setMaxSpeed() and carInfo() methods still we can use it due to IS-A relationship of Maruti class with Car class.

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php)

1. **package** relationships;
2. **public** **class** RelationsDemo {
3. **public** **static** **void** main(String[] args) {
4. Maruti myMaruti = **new** Maruti();
5. myMaruti.setColor("RED");
6. myMaruti.setMaxSpeed(180);
7. myMaruti.carInfo();
8. myMaruti.MarutiStartDemo();
9. }
10. }

If we run RelationsDemo class we can see output like below.



**Comparing Composition and Inheritance**

* It is easier to change the class implementing composition than inheritance. The change of a superclass impacts the inheritance hierarchy to subclasses.
* You can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
* Composition is dynamic binding (run-time binding) while Inheritance is static binding (compile time binding)
* It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition) because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool.
* With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remains inside the same class.
  + **Don't use inheritance just to get code reuse** If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
  + **Don't use inheritance just to get at polymorphism** If all you really want is a polymorphism, but there is no natural is-a relationship, use composition with interfaces.

**Summary**

* IS-A relationship based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance.
* Has-a relationship is composition relationship which is a productive way of code reuse.
* **Aggregation in Java**

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

1. **class** Employee{
2. **int** id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

#### Address.java

1. **public** **class** Address {
2. String city,state,country;
4. **public** Address(String city, String state, String country) {
5. **this**.city = city;
6. **this**.state = state;
7. **this**.country = country;
8. }
10. }

#### Emp.java

1. **public** **class** Emp {
2. **int** id;
3. String name;
4. Address address;
6. **public** Emp(**int** id, String name,Address address) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.address=address;
10. }
12. **void** display(){
13. System.out.println(id+" "+name);
14. System.out.println(address.city+" "+address.state+" "+address.country);
15. }
17. **public** **static** **void** main(String[] args) {
18. Address address1=**new** Address("gzb","UP","india");
19. Address address2=**new** Address("gno","UP","india");
21. Emp e=**new** Emp(111,"varun",address1);
22. Emp e2=**new** Emp(112,"arun",address2);
24. e.display();
25. e2.display();
27. }
28. }

Output:111 varun

gzb UP india

112 arun

gno UP india

* [**Memory Management In Java – Stack And Heap**](http://javaconceptoftheday.com/memory-management-java/)

Whenever you trigger a java command, it divides allocated memory into two parts – Stack and Heap. Stack is used only for execution purpose. Heap is used for storage purpose. consider the following program and we will see how it uses stack and heap memory through diagram.

class StackAndHeapMemory

{

static void methodOne()

{

System.out.println("From Method One");

methodTwo();

}

static void methodTwo()

{

System.out.println("From Method Two");

}

public static void main(String[] args)

{

System.out.println("Main Method Started");

methodOne();

System.out.println("Main Method Ended");

}

}

When you trigger >java StackAndHeapMemory, java command divides allocated memory into two parts. one is stack and another one is heap. As already said, stack is used for execution purpose and heap is used for storage purpose. Here is the steps involved in execution of above program.

Step 1 :

First java command enters the stack memory for execution. First it checks whether the class StackAndHeapMemory is loaded in heap memory or not. If it is not loaded, loading operation of class StackAndHeapMemory starts.

Step 2 :

Randomly one object is created in the heap memory. It is also called Class Memory. After object creation, all static members are loaded into class memory. You know that execution of every java program start with main() method. So, java commands calls main() method for execution.

Step 3 :

main() method enters stack memory for execution. First statement in main() method (Line 16) is executed. It prints “Main Method Started” on the console. In the second statement (Line 17), it calls methodOne() for execution.

Step 4 :

methodOne() enters the stack for execution. First statement (Line 5) of methodOne() is executed first. It prints “From Method One” on the console. In the second statement (Line 6), it calls methodTwo() for execution.

Step 5 :

methodTwo() enters the stack for execution. In the methodTwo(), there is only one statement (Line 11). This statement is executed. It prints “From Method Two” on the console. There is no other statements in methodTwo(). So, methodTwo() leaves stack memory.

Step 6 :

Now, again control comes back to methodOne(). Second statement (Line 6) of methodOne() is already executed in step 4. There are no other statements left in methodOne(). So, methodOne() also leaves stack.

Step 7 :

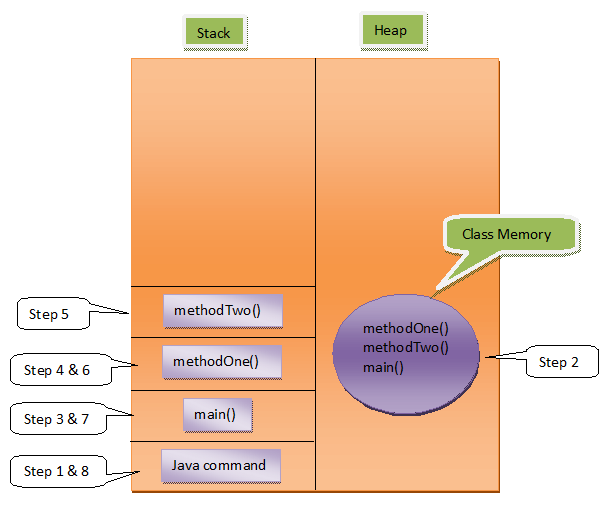
Now, control comes back to main() method. second statement (Line 17) of main() is already executed is in step 3. So, now third statement (Line 18) is executed. It prints “Main Method Ended” on the console. As there are no other statements left in main() method, it also leaves stack after clearing the heap memory.

Step 8 :

java command also leaves stack memory and gives back the allocated memory to OS.

Here is the diagrammatic representation of memory allocation of the above program.

Memory Management In Java



Output :

Main Method Started

From Method One

From Method Two

Main Method Ended

* **HashCode and equals methods**

HashTable, HashMap and HashSet are the Collection classes in java.util package that make use of hashing algorithm to store objects. In all these Collection classes except HashSet, objects are stored as key-value pairs. For the storage and the retrieval of any user-defined objects it is a good practice to override the following methods which is mentioned below,

* **hashCode()**
* **equals()**

These methods are available in the Object class and hence available to all java classes.Using these two methods, an object can be stored or retrieved from a **Hashtable, HashMap or HashSet**.

## hashCode() method

This method returns a **hashcode** value as an int for the object. Default implementation for hashcode() should be overridden in order to make searching of data faster. The implementation of **hashCode()** method for an user-defined object should be calculated based on the properties of the class which we wish to consider.

## equals() method

This method returns a boolean which specifies whether two objects are equal or not. The default implementation of equals() method given by the Object Class uses the ‘==’ operator to compare two object references, and returns true only if they refer to the same object. But, we can meaningfully re-define this equals() method to have en equality check based on our own criteria.  
Consider the following code, which defines two user defined classes Employee and EmployeeId which are supposed to be stored in a Map.

**Employee.java**

public class Employee {

private String name;

public Employee(String name) {

this.name = name;

}

public String toString() {

return name;

}

}

**EmployeeId.java**

public class EmployeeId {

private String id;

public EmployeeId(String id) {

this.id = id;

}

public String toString() {

return id;

}

}

The following class makes use of the above classes by storing it in a Map for later retrieval. We are adding Employee objects into the Map keyed with Employee Id.  
**HashCodeTest.java**

public class HashCodeTest {

public static void main(String[] args) {

Map<EmployeeId, Employee>

employees = new HashMap<EmployeeId, Employee>;

employees.put(new EmployeeId("111"), new Employee("Johny"));

employees.put(new EmployeeId("222"), new Employee("Jeny"));// Line A

employees.put(new EmployeeId("333"), new Employee("Jessie"));

Employee emp = employees.get(new EmployeeId("222"));// Line B

System.out.println(emp); // Line C

}

}

}

}

In Line B, we try to retrieve the Employee object who has Employee Id with a value of 222. We expect the output to be **‘Jeny’**, because the Employee with Employee Id (222) was already there in the Collection, but surprisingly, the output of the above code is **null**. The reason is that we did not override the equals() method for EmployeeId and Employee classes because the default implementation of equals() in the Object class considers the new EmployeeId("222") in the put statement and new EmployeeId("222") in the get statement as two different instances, and hence the call to get() in Line B returnsnull.  
Let us look at how the same code works when we provide our desired implementation for hashcode() and equals() methods. We basically override hashcode() here just to make the object to be searched fast.

**Employee.java**

public class Employee {

private String name;

public Employee(String name) {

this.name = name;

}

public String toString() {

return name;

}

@Override

public boolean equals(Object obj) {

if (obj == null) {

return false;

}

if (obj.getClass() != getClass()) {

return false;

}

Employee emp = (Employee) obj;

if (this.name == emp.name) {

return true;

}

return false;

}

@Override

public int hashCode() {

return name.hashCode();

}

}

**EmployeeId.java**

public class EmployeeId {

private String id;

public EmployeeId(String id) {

this.id = id;

}

public String toString() {

return id;

}

public boolean equals(Object obj) {

if (obj == null)

return false;

if (obj.getClass() != getClass()) {

return false;

}

EmployeeId empId = (EmployeeId) obj;

if (this.id == empId.id) {

return true;

}

return false;

}

@Override

public int hashCode() {

return id.hashCode();

}

}

Now, we get the desired output ‘Jeny’, because as per our implementation for the equals() method, the new EmployeeId("222") in the put statement and new EmployeeId("222") in the get statement are considered one and the same.