**Core Java**

1.Oops concept in java with example.

Ans:-

* [1.1 Abstraction](https://www.journaldev.com/12496/oops-concepts-java-example" \l "abstraction)
* [1.2 Encapsulation](https://www.journaldev.com/12496/oops-concepts-java-example" \l "encapsulation)
* [1.3 Polymorphism](https://www.journaldev.com/12496/oops-concepts-java-example#polymorphism)
* [1.4 Inheritance](https://www.journaldev.com/12496/oops-concepts-java-example#inheritance)
* [1.5 Association](https://www.journaldev.com/12496/oops-concepts-java-example#association)
* [1.6 Aggregation](https://www.journaldev.com/12496/oops-concepts-java-example#aggregation)
* [1.7 Composition](https://www.journaldev.com/12496/oops-concepts-java-example#composition)
  1. Abstraction

Abstraction is the concept of hiding the internal details and describing things in simple terms. For example, a method that adds two integers. The method internal processing is hidden from outer world. There are many ways to achieve abstraction in object oriented programming, such as encapsulation and inheritance.

A java program is also a great example of abstraction. Here java takes care of converting simple statements to machine language and hides the inner implementation details from outer world.

* 1. Encapsulation

Encapsulation is the technique used to implement abstraction in object oriented programming. Encapsulation is used for access restriction to a class members and methods.

Access modifier keywords are used for encapsulation in object oriented programming. For example, encapsulation in java is achieved using private, protected and public keywords.

* 1. Polymorphism

Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism – compile time polymorphism and runtime polymorphism.

Compile time polymorphism is achieved by method overloading. For example, we can have a class as below.

public class Circle {

public void draw(){

System.out.println("Drwaing circle with default color Black and diameter 1 cm.");

}

public void draw(int diameter){

System.out.println("Drwaing circle with default color Black and diameter"+diameter+" cm.");

}

public void draw(int diameter, String color){

System.out.println("Drwaing circle with color"+color+" and diameter"+diameter+" cm.");

}

}

Here we have multiple draw methods but they have different behavior. This is a case of method overloading because all the methods name is same and arguments are different. Here compiler will be able to identify the method to invoke at compile time, hence it’s called compile time polymorphism.

Runtime polymorphism is implemented when we have “IS-A” relationship between objects. This is also called as method overriding because subclass has to override the superclass method for runtime polymorphism. If we are working in terms of superclass, the actual implementation class is decided at runtime. Compiler is not able to decide which class method will be invoked. This decision is done at runtime, hence the name as runtime polymorphism or dynamic method dispatch.

package com.journaldev.test;

public interface Shape {

public void draw();

}

package com.journaldev.test;

public class Circle implements Shape{

@Override

public void draw(){

System.out.println("Drwaing circle");

}

}

package com.journaldev.test;

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Drawing Square");

}

}

Shape is the superclass and there are two subclasses Circle and Square. Below is an example of runtime polymorphism.

Shape sh = new Circle();

sh.draw();

Shape sh1 = getShape(); //some third party logic to determine shape

sh1.draw();

In above examples, java compiler don’t know the actual implementation class of Shape that will be used at runtime, hence runtime polymorphism

* 1. Inheritance

Inheritance is the object oriented programming concept where an object is based on another object. Inheritance is the mechanism of code reuse. The object that is getting inherited is called superclass and the object that inherits the superclass is called subclass.

We use extends keyword in java to implement inheritance. Below is a simple example of inheritance in java.

package com.journaldev.java.examples1;

class SuperClassA {

public void foo(){

System.out.println("SuperClassA");

}

}

class SubClassB extends SuperClassA{

public void bar(){

System.out.println("SubClassB");

}

}

public class Test {

public static void main(String args[]){

SubClassB a = new SubClassB();

a.foo();

a.bar();

}

}

* 1. Association.

Association is the OOPS concept to define the relationship between objects. Association defines the multiplicity between objects. For example Teacher and Student objects. There is one to many relationship between a teacher and students. Similarly a student can have one to many relationship with teacher objects. However both student and teacher objects are independent of each other.

* 1. Aggregation.

Aggregation is a special type of association. In aggregation, objects have their own life cycle but there is an ownership. Whenever we have “HAS-A” relationship between objects and ownership then it’s a case of aggregation.

* 1. Composition.

Composition is a special case of aggregation. Composition is a more restrictive form of aggregation. When the contained object in “HAS-A” relationship can’t exist on it’s own, then it’s a case of composition. For example, House has-a Room. Here room can’t exist without house.

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

state: represents data (value) of an object.

behavior: represents the behavior (functionality) of an object such as deposit, withdraw etc.

identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

Q.)Class A has constructor with one parameter and class B extends class A and does not have any constructor whether it will compile or not.

Ans-No.

Q.)A class have two overloaded methods with one int I and second Integer I arguments which one will be called. Method(5)

**public** **class** Test {

**public** **void** get(**int** i){

System.***out***.println("int i");

}

**public** **void** get(Integer f){

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

}

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

// **TODO** Auto-generated method stub

Test t = **new** Test();

t.get(5);

}

}

o/p int i

Q.)A class have two overloaded methods with int I and flaot I arguments which one will be called if we call method(5).

**public** **class** Test {

**public** **void** get(**float** f){

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

}

**public** **void** get(**int** i){

System.***out***.println("int i");

}

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

// **TODO** Auto-generated method stub

Test t = **new** Test();

t.get(5);

}

}

o/p int i

Q.)List<Animal> list = new ArrayList<Dog>();------will compile or not.

No. Class.

Q.)why java does not multiple inheritance ?

Ans

Q.)what is generic in java.

Q.)print name without semicolon.

Q.)what is purpose of default method in interface of java1.8

Q.)diffrenece between java1.8 and 1.7

Q.)new features in java 1.8

2.Overloading Concept in java.

If a class has multiple methods having same name but different in type signature, it is known as Method Overloading. Where type signature is

Number of parameter

Type of parameter

Order of parameter.

One of them must be involved in order to for overloading.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Advantage of method overloading

Method overloading increases the readability of the program.

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

class Adder{

static int add(int a,int b){return a+b;}

static double add(int a,int b){return a+b;}

}

class TestOverloading3{

public static void main(String[] args){

System.out.println(Adder.add(11,11));//ambiguity

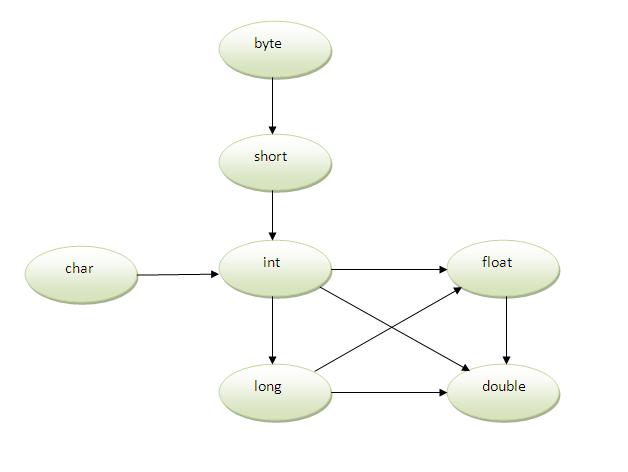
}}

Output:

Compile Time Error: method add(int,int) is already defined in class Adder.

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

1. **class** OverloadingCalculation1{
2. **void** sum(**int** a,**long** b){System.out.println(a+b);}
3. **void** sum(**int** a,**int** b,**int** c){System.out.println(a+b+c);}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation1 obj=**new** OverloadingCalculation1();
7. obj.sum(20,20);//now second int literal will be promoted to long
8. obj.sum(20,20,20);
10. }
11. }

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

class OverloadingCalculation2{

void sum(int a,int b){System.out.println("int arg method invoked");}

void sum(long a,long b){System.out.println("long arg method invoked");}

public static void main(String args[]){

OverloadingCalculation2 obj=new OverloadingCalculation2();

obj.sum(20,20);//now int arg sum() method gets invoked

}

}

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

1. **class** OverloadingCalculation3{
2. **void** sum(**int** a,**long** b){System.out.println("a method invoked");}
3. **void** sum(**long** a,**int** b){System.out.println("b method invoked");}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation3 obj=**new** OverloadingCalculation3();
7. obj.sum(20,20);//now ambiguity
8. }
9. }

Compile time error.

# Method Overriding in Java

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

### Usage of Java Method Overriding

* Method overriding is used to provide specific implementation of a method that is already provided by its super class.
* Method overriding is used for runtime polymorphism

#### Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

### Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

1. **class** Vehicle{
2. **void** run(){System.out.println("Vehicle is running");}
3. }
4. **class** Bike **extends** Vehicle{
6. **public** **static** **void** main(String args[]){
7. Bike obj = **new** Bike();
8. obj.run();
9. }
10. }

Output:Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

1. **class** Vehicle{
2. **void** run(){System.out.println("Vehicle is running");}
3. }
4. **class** Bike2 **extends** Vehicle{
5. **void** run(){System.out.println("Bike is running safely");}
7. **public** **static** **void** main(String args[]){
8. Bike2 obj = **new** Bike2();
9. obj.run();
10. }

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.

Java method overriding example of bank

### Can we override static method?

No, static method cannot be overridden. It can be proved by runtime polymorphism

### Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

### Can we override java main method?

No, because main is a static method.

|  |  |
| --- | --- |
| overloading | *overriding* |
| In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

🡪Covariant return type

When we are changing the return type while overriding then it is called Covariant return type.in this case changing the return type can be allowed on only Non-primitive type not on primitive data type.

For example.

1. **class** A{
2. A get(){**return** **this**;}
3. }
5. **class** B1 **extends** A{
6. B1 get(){**return** **this**;}
7. **void** message(){System.out.println("welcome to covariant return type");}
9. **public** **static** **void** main(String args[]){
10. **new** B1().get().message();
11. }
12. }

Output:welcome to covariant return type

2nd example

Here it is compile time error

**package** com.hcl.demo;

**class** A{

**public** **int** get(**int** a){

System.***out***.println("in A");

**return** a;

}

}

**class** B **extends** A{

**public** **long** get(**int** a){

System.***out***.println("in B");

**return** a;

}

}

**public** **class** Test4 {

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

// **TODO** Auto-generated method stub

}

}

Here it will show return type is incompatible with A.get().

**\*\*Instance Initializer block** is used to initialize the instance data member. It run each time when object of the class is created.

**Rule:-**

1.instance Initiliazer block will be created each time whenever object is created.

2.instance Initializer blow will be executed after parent class constructor invoked.

3.It execute in the same fashion in which order has been written.

**class** A{

**int** b;

{sysout(b);}

public A(){

sysout(“in A”);

}

}

**public** **class** Test4 {

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

// **TODO** Auto-generated method stub

A a = **new** A();

System.***out***.println(a.b);

}

}

Output

9

In A

Instance Initializer block will be added at the time of class object creation.

**class** A{

**int** b=100;

**public** A(){

System.***out***.println("In A"+b);

}

{System.***out***.println(b);}

}

In above code when we will create an object then also memory space will be created for B object then what is difference between 2nd last and 1st code.

In this case

If we want to execute some code while creation of an object of class.

Or we have requirement for fill array of element while initialization of an object of class.in this case we use Instance Initializer block.

In summary,

Whenever an object of class created all instance initialize blocks will be added inside the comstructor and executed first .but in case of parent/child relationship first parent class constructor will be executed then subclass constructor will be executed so priority of Intance initialize block depends on constructor of super and sub class.

## Program of instance initializer block that is invoked after super()

1. **class** A{
2. A(){
3. System.out.println("parent class constructor invoked");
4. }
5. }
6. **class** B2 **extends** A{
7. B2(){
8. **super**();
9. System.out.println("child class constructor invoked");
10. }
12. {System.out.println("instance initializer block is invoked");}
14. **public** **static** **void** main(String args[]){
15. B2 b=**new** B2();
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=B2)

Output:parent class constructor invoked

instance initializer block is invoked

child class constructor invoked

Q.)Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

1. **class** Bike{
2. **final** **void** run(){System.out.println("running...");}
3. }
4. **class** Honda2 **extends** Bike{
5. **public** **static** **void** main(String args[]){
6. **new** Honda2().run();
7. }
8. }

Output

running..

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

1. **class** Bike10{
2. **final** **int** speedlimit;//blank final variable
4. Bike10(){
5. speedlimit=70;
6. System.out.println(speedlimit);
7. }
9. **public** **static** **void** main(String args[]){
10. **new** Bike10();
11. }
12. }

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

1. **class** A{
2. **static** **final** **int** data;//static blank final variable
3. **static**{ data=50;}
4. **public** **static** **void** main(String args[]){
5. System.out.println(A.data);
6. }
7. }

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

1. **class** Bike11{
2. **int** cube(**final** **int** n){
3. n=n+2;//can't be changed as n is final
4. n\*n\*n;  //can't be changed as n is
5. Return
6. }
7. **public** **static** **void** main(String args[]){
8. Bike11 b=**new** Bike11();
9. b.cube(5);
10. }
11. }