

OOPs Concepts

Topics

- 1. Data hiding**
- 2. Abstraction**
- 3. Encapsulation**
- 4. Tightly encapsulated class**
- 5. Is-A relationship**
- 6. Has-A relationship**
- 7. Overloading**
- 8. Method signature**
- 9. Overriding**
- 10. Difference between overloading and overriding**
- 11. Polymorphism**
- 12. Coupling**
- 13. Cohesion**
- 14. Type-casting**
- 15. Static Control Flow**
- 16. Instance Control flow**
- 17. Constructor**
- 18. Singleton Classes**

Data Hiding

Outside person can't access our internal data directly or our internal data should not go out directly. This OOP feature is nothing but data hiding. After validation or authentication, outside person can access our internal data.

Example 1: After providing proper username and password, we can able to access our gmail inbox information.

Example 2: Even though we are valid customer of the bank. We can able to access our account information and we can't access others account information.

By declaring data member (Variable) as private we can achieve data hiding.

```
public class Account {  
    private double balance;  
    public double getBalance () {  
        //Validation  
        return balance;  
    }  
}
```

The main advantage of data hiding is security.

Note: It is highly recommended to declare data member (Variable) as private.

Abstraction

Hiding internal implementation and just highlight the set of services what we are offering is the concept of abstraction.

Example: Through Bank ATM GUI screen bank people are highlighting the set of services what we are offering without highlighting internal implementation.

The main advantages of abstraction are:

1. Security
2. Enhancement is easy
3. Improve easiness
4. Easy maintenance

By using interface and abstract class we can implement abstraction.

Encapsulation

The process of binding data and corresponding method into a single unit is called encapsulation.

```
public class Student {  
    //data member  
    +  
    //methods (behavior)  
}
```

If any component follows data hiding and abstraction, such type of component is said to be encapsulated Component.

Encapsulation = data hiding + abstraction

The main advantages of encapsulation are:

1. Security
2. Enhancement will be easy.
3. Improves maintainability of application.

The main advantage of encapsulation is we can achieve security, but the main disadvantage of encapsulation is it increases length of the code and slows down its execution.

Tightly encapsulated class

A class is said to be tightly encapsulated if and only if each variable declared as private whether class contains corresponding getter and setter method or not and whether these methods are public or not these things we are not required to check.

```
public class Account {  
    private double balance;  
    public double getBalance() {  
        return balance;  
    }  
}
```

Q) Which of the following classes are tightly encapsulated?

```
public class A {  
    private int x=10;  
} //tightly encapsulated  
public class A extends B {  
    int y=20;  
} //not tightly encapsulated  
public class C extends A {  
    private int z=30;  
} //tightly encapsulated
```

Q) Which of the following classes are tightly encapsulated?

```
public class A {  
    int x=10;  
} //No  
public class A extends B {  
    private int y=10;  
} //No  
public class C extends A {  
    private int z=30;  
} //No
```

Note: If the parent class is not tightly encapsulated that child class is not tightly encapsulated.

Is-a relationship

- It is also known as inheritance.
- The main advantage of Is-a relationship is code reusability.
- By using extends keywords we can implement Is-a relationship.

```
public class P {  
    public void m1 () {  
        System.out.println ("Parent");  
    }  
}  
public class C extends P {  
    public void m2 () {  
        System.out.println ("Child");  
    }  
}  
public class Test {  
    public static void main (String[] args) {  
        P p = new P ();  
    }  
}
```

```

    p.m1 (); // Correct
    p.m2 (); //CE: cannot find symbol, Symbol: method m2 (), location: class P

    C c = new C ();
    c.m1 (); // Correct
    c.m2 (); // Correct

    P p1 = new C ();
    p1.m1 (); // Correct
    p1.m2 (); //CE: cannot find symbol, Symbol: method m2 (), location: class P

    C c1 = new P (); // CE: incompatible types, found: P, Required: C
}
}

```

Conclusions:

1. Whatever method parent has, by default available to the child and hence on child reference we can call both parent method and child method.
2. Whatever methods child has by default not available to parent and hence on parent reference we can't call child specific methods.
3. Parent reference can be used to hold child object, but by using that reference we can't call child specific methods, but we can call the methods present in parent class.
4. Child reference can't be used to hold parent object.

Without inheritance

```

class VLoan {
    //300 method
}
class HLoan {
    //300 method
}
class PLoan {
    //300 method
}
//Total 900 methods

```

With inheritance

```

class Loan {
    //250 common method
}
class VLoan extends Loan {
    //50 specific method
}
class HLoan extends Loan {
    //50 specific method
}
class PLoan extends Loan {
    //50 specific method
}
//Total 400 methods

```

Note: Most common methods which are applicable for any type of child, we must define in parent class. The specific methods which are applicable for a particular child we have to define in child class.

Total Java API is implemented based on inheritance concept. The most common method which is applicable for any Java object is defined in object class and hence every class in Java is child class of the object either directly or indirectly. So that object class method by default available to every java class without rewriting due to this object class acts as root for all java classes. Throwable classes define the most common method which is required for every exception and error classes. Hence this class acts as root for Java exception hierarchy.

Multiple Inheritance

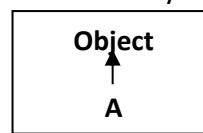
A java class can't extend more than one class at a time. Hence Java won't provide support for multiple Inheritances in classes.

```
public class A extends B,C {
} //Compile-time Error
```

Note:

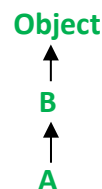
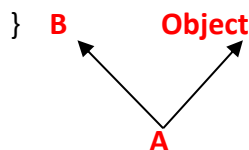
1. If our class doesn't extend any other class then only our class is direct child class of object.

Class A {
}



2. If our class extends any other class then our class is indirect child class of object.

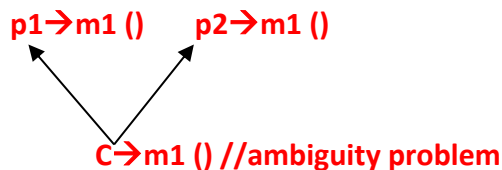
Class A extends B {
} **B** **Object**



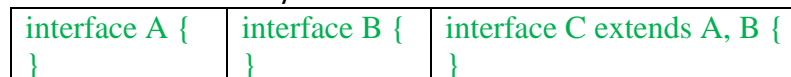
Note: Either directly or indirectly java won't provide support for multiple inheritance with respect to classes.

Q) Why Java won't provide support for multiple inheritance?

Ans: There may be a chance of ambiguity problem hence Java won't provide support for multiple Inheritances.

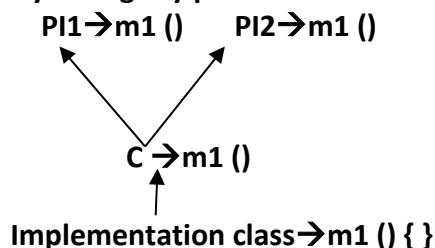


Class can't extend any number of classes simultaneously, but interface can extend s any number of interfaces simultaneously.



Hence Java provides support of for multiple inheritances with respect to interfaces.

Q) Why ambiguity problem won't be there in interfaces?



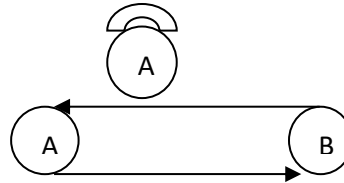
Even though multiple method declaration is available but implementation is unique and hence there is no chance of ambiguity problem in interfaces.

Note: Strictly speaking, through interfaces we won't get any inheritance.

Cyclic Inheritance

Not allowed in Java.

```
public class A extends A {
} //CE: Cyclic inheritance involving A
public class A extends B {
}
public class B extends A {
} //CE: Cyclic inheritance involving A
```



Has-a relationship

- Has-a relationship is also known as composition or Aggregation.
- There is no specific keyword to implement has – a relation but most of the time we are depending on new keyword.
- The main advantage of has-a relationship is reusability of the code.

```
public class Car {
    Engine e = new Engine ();
} Car Has-a engine reference.
```

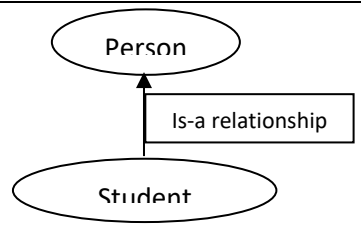
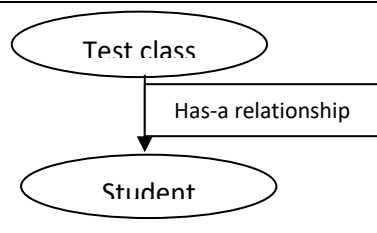
Composition Vs Aggregation

Composition	Aggregation
Without existing container object, there is no chance of existing contained object then container and contained objects are strongly associated and this Strong Association is nothing but Composition.	Without existing container object if there is a chance of existing contained object then container and contained objects are weakly associated and this weak association is nothing but Aggregation.
Example: University consists of several departments, without existing university there is no chance of existing department. Hence university and department are strongly associated, and this strong association is nothing but composition.	Example: Department consist of several professors without existing department there may be a chance of existing professors objects. Hence department and professor objects are weakly associated, and this weak association is nothing but aggregation.

- 1) In composition, Objects are strongly associated whereas in aggregation objects are weakly associated.
- 2) In composition, container object holds directly contained objects where as in-aggregation container object holds just references of contained objects.

Is-a relationship Vs Has-a relationship

Is-a relationship	Has-a relationship
If we want total functionality of a class automatically then we should go for Is-a relationship.	If we want part of the functionality then we should go for Has-a relationship.
<pre>public class Person { } public class Student extends Person {</pre>	<pre>public class Test { //100 method }</pre>

}	public class Student extends Test { Test t = new Test (); t.m1 (); }
 <pre> classDiagram class Person class Student Student -- > Person : Is-a relationship </pre>	 <pre> classDiagram class Testclass class Student Testclass --> Student : Has-a relationship </pre>

Method Signature

- In java, method signature consists of method names followed by argument types.

public static int m1 (int i, float f); → m1 (int, float)

- Return type is not part of method signature in Java.
- Compiler will use method signature to resolve method calls.

```

public class Test {
    public void m1 (int i) {
    }
    public void m2 (String i) {
    }
    public static void main (String... args) {
        Test t = new Test ();
        t.m1 (10);
        t.m2 ("Dhruv");
        t.m3 (10.5); // CE: cannot find Symbol, Symbol: method m3 (double),
location: class Test
    }
}

```

Within a class 2 methods with same signature is not allowed.

```

public class Test {
    public void m1 (int i) {
    }
    public void m1 (int x) {
        return s;
    } // CE: m1 (int) is already defined in Test
}

```

Overloading

Two methods are said to be overloaded if and only if both methods having same name but different argument types.

In c language overloading concept is not available hence we can't declare multiple methods with same name but different argument types.

If there is a change in argument type, compulsory we should go for new method name, which increases complexity of programming.

- abs (int i) → abs (10); (C)
- labs (long l) → labs (10l); (C)
- fabs (float f) → fabs (10.56); (C)

But in java ***we can declare multiple methods with same name but different argument types. Such types of methods are called overloaded methods.***

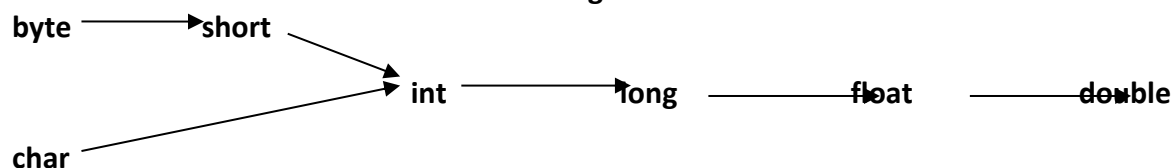
- abs (int i);
- abs (long l);
- abs (float f);

Having overloading concept in Java reduces complexity in programming.

```
public class Test {
    public void m1 () {
        System.out.println ("no-arg");
    }
    public void m1 (int i) {
        System.out.println ("int-arg");
    }
    public void m1 (double l) {
        System.out.println ("double-arg");
    }
    public static void main (String...strings args) {
        Test t = new Test ();
        t.m1 (); //no-arg
        t.m1 (10); //int-arg
        t.m1 (10.5); //double-arg
    }
}
```

- It is also known as compile-time polymorphism/static binding/ early-binding.
- In overloading, method resolution always takes care by compiler based on reference type.

Case 1: Automatic Promotion in Overloading



While resolving overloaded methods if exact matched method is not available then we won't get any CTE immediately. First, it will promote argument to the next level and check weather matched method is available or not. If matched method is available, then it will be considered and if matched method is not available then compiler promotes argument once again to the next level. The process will be continued until all possible promotions; still if the matched method is not available then we will get CTE. The above are all possible promotions in **overloading. The process is called Automatic promotion in overloading.**

```
public class Test {
    //Overloaded method
    public void m1 (int i) {
        System.out.println("int-arg");
    }
    public void m1 (float f) {
        System.out.println("float-arg");
    }
    public static void main (String... args) {
        Test t = new Test();
        t.m1 (10); //int-arg
        t.m1 (10.5f); //float-arg
        t.m1 ('a'); //int-arg
    }
}
```



```

        t.m1 (101); //float-arg
        t.m1 (10.5); //CE: cannot find Symbol, Symbol: method m1(double) location: class Test
    }
}

```

Case 2:

```

public class Test {
    //Overloaded method
    public void m1 (String s) {
        System.out.println("String-version");
    }
    public void m1 (Object o) {
        System.out.println("Object version");
    }
    public static void main (String... args) {
        Test t = new Test();
        t.m1 (new Object()); //object version
        t.m1 ("dhruv"); //String version
        t.m1 (null); //String version
    }
}

```

Note: While resolving overloaded methods, compiler will always gives precedence for child type argument when compared with parent type argument.

Case 3:

```

public class Test {
    //Overloaded method
    public void m1 (String s) {
        System.out.println("String-version");
    }
    public void m1 (StringBuffer sb) {
        System.out.println("StringBuffer version");
    }
    public static void main (String... args) {
        Test t = new Test();
        t.m1 ("dhruv"); // String version
        t.m1 (new StringBuffer("dhruv")); //StringBuffer version
        t.m1 (null); //CE: reference to m1 () is ambiguous
    }
}

```

Case 4:

```

public class Test {
    //Overloaded method
    public void m1 (int i, float f) {
        System.out.println("int-float version");
    }
    public void m1 (float f, int i) {
        System.out.println("float-int version");
    }
    public static void main (String... args) {
        Test t = new Test();
        t.m1 (10,10.5f); //int-float version
        t.m1 (10.5f,10); //float-int version
        t.m1 (10,10); //CE: reference to m1 () is ambiguous
        t.m1 (10.5f,10.5f); //CE: cannot find symbol symbol: method m1 (float, float) location: class Test
    }
}

```

Case 5:

```

public class Test {
    //Overloaded method
    public void m1 (int x) {

```

```

        System.out.println("General method");
    }
    public void m1 (int... i) {
        System.out.println("Var-arg method");
    }
    public static void main (String... args) {
        Test t = new Test();
        t.m1 (); //Var-arg method
        t.m1 (10, 20); //Var-arg method
        t.m1 (10); //General method
    }
}

```

In general var-arg method will get least priority i.e. if no other method matched then only var-arg method will get the chance it is exactly same as default case inside switch.

Case 6:

```

public class Animal {
}
public class Monkey extends Animal {
}
class Test {
    public void m1 (Animal a) {
        System.out.println("Animal version");
    }
    public void m1 (Monkey m) {
        System.out.println("Monkey version");
    }
    public static void main (String[] args) {
        Test t = new Test ();

        Animal a = new Animal();
        t.m1(a); //Animal version

        Monkey m = new Monkey ();
        t.m1(m); //Monkey version

        Animal a1 = new Monkey (); //parent reference child object
        t.m1(a1); //Animal version
    }
}

```

Note: In overloading, method resolution always takes care by compiler based on reference type. In overloading run time object won't play any role.

Overriding

Whatever methods parent has, by default available to child through inheritance. If child class not satisfied with parent class implementation, then child can redefine that method based on its requirement. This process is called overriding.

The parent class method which is overridden is called overridden method and child class method which is overriding is called overriding method.

```

public class P {
    public void property () {
        System.out.println("Cash + Land + Gold");
    }
    public void marry () {
        System.out.println("Subba Laxmi");
    }
}
public class C extends P {
    public void marry () {
        System.out.println("NayanTara");
    }
}

```

```

    }
}
class Test {
    public static void main (String[] args) {
        P p = new P ();
        p.marry (); // parent method

        C c = new C ();
        c.marry(); //child method

        P p1 = new C();
        p1.marry(); //child method
    }
}

```

In overriding, method resolution always takes care by JVM based on runtime object and hence overriding is also considered as run-time polymorphism or dynamic polymorphism or late-binding.

Rules for overriding

1. In overriding, method names and argument types must be matched i.e. method signature must be same.
2. In overriding, return types must be same, but this rule is valid till 1.4 version. From 1.5 version onwards child of parent return types is allowed i.e. co-variant return types.

Example: *//It is invalid in 1.4 version. But from 1.5 version it is valid*

```

public class P {
    public object m1 () {
        return null;
    }
}
public class C extends P {
    public String m1 () {
        return null;
    }
}

```

Parent class return types	Child class method return types
Object	Object/String/StringBuffer
Number	Number/Integer/double/float
String	Object
double	int

Co-variant return type concept is applicable for object types, but not for primitive types.

3. Parent class private methods not available to the child and hence overriding concept is not applicable for private methods.

Based on our requirement we can define exactly same private method in child class it is valid but not overriding.

```

public class P {
    private void m1 () {
    }
}
public class C extends P {
    private void m1 () {
    }
}
//It is valid, but not overriding.

```

4. We can't override parent class final method in child classes, if we are trying to override, we will get CE.

```
public class P {  
    public final void m1 () {  
    }  
}  
  
public class C extends P {  
    public void m1 () {  
    }  
}  
//CE: m1 () in C cannot override m1 () in P overridden method is final
```

5. Parent class abstract method we should override in child class to provide implementation.

```
abstract class P {  
    public abstract void m1 ();  
}  
  
public class C extends P {  
    public void m1 () {  
    }  
}
```

6. We can override non-abstract method as abstract.

```
public class P {  
    public void m1 () {  
    }  
}  
  
abstract class C extends P {  
    public abstract void m1 ();  
}
```

The main advantage of this approach is we can stop the availability of parent method implementation to the next level child classes.

7. In overriding, **abstract**, **synchronized**, **native**, **strictfp** modifier won't keep any restrictions.

Parent method → Child method
Final → non-final
Non-final → final
Abstract → non-abstract
Non-abstract → abstract
synchronized → non-synchronized
non-synchronized → synchronized
native → non-native
non-native → native
Strictfp → non-strictfp
non-strictfp → Strictfp

8. In overriding we can't reduce scope of access modifier, but we can increase the scope.

```
public class P {  
    public void m1 () {  
    }  
}  
  
public class C extends P {  
    void m1 ();  
}  
//CE: m1 () in C cannot override m1 () in P, attempting to assign weaker  
access privileges was public
```

private < default < protected < public
--

Parent class method	Child class method
public	public
protected	protected /public
default	default /protected /public
private	Overriding concept not applicable to private

9. If child class method throws any checked exception compulsory parent class method should throw the same checked exception or its parent otherwise, we will get CTE. But there is no restriction for unchecked exception.

```
public class P {
    public void m1 () throws IOException {
    }
}
public class C extends P {
    public void m1 () throws EOFException, InterruptedException {
    }
}
//CE: m1 () in c cannot override m1 () in P, overridden method doesn't throw
Java.lang.InterruptedException
```

Example:

1. P: public void m1 () throws Exception
C: public void m1 ()
2. P: public void m1 ()
C: public void m1 () throws Exception
3. P: public void m1 () throws Exception
C: public void m1 () throws IOException
4. P: public void m1 () throws IOException
C: public void m1 () throws Exception
5. P: public void m1 () throws IOException
C: public void m1 () throws FileNotFoundException, EOFException
6. P: public void m1 () throws IOException
C: public void m1 () throws EOFException, InterruptedException
7. P: public void m1 () throws IOException
C: public void m1 () throws ArithmeticException, NullPointerException, ClassCastException

Overriding w.r.t static methods

Case 1: We can't override a static method as non-static method otherwise we will get CE.

```
public class P {
    public static void m1 () {
    }
}
public class C extends P {
    public void m1 () {
    }
}
//CE: m1 () in c cannot override m1 () in P, overridden method is static.
```

Case 2: we can't override a non-static method as static.

```
public class P {
    public void m1 () {
    }
}
public class C extends P {
    public static void m1 () {
    }
}
//CE: m1 () in c cannot override m1 () in P, overriding method is static
```

Case 3: If both parent and child class methods are static then we won't get any CE. It seems overriding concept applicable for static method. But it is not overriding, and it is method-hiding.

```
public class P {
    public static void m1 () {
    }
}

public class C extends P {
    public static void m1 () {
    }
}

//It is method hiding. But not overriding
```

Method hiding

All rules of method hiding are exactly same as overriding except the following differences.

Method hiding	Overriding
Both methods should be static.	Both parent and child class methods should be non-static.
Compiler is responsible for method resolution based on reference type.	JVM is responsible for method resolution based on run-time object.
It is also known as compile-time polymorphism or static polymorphism or early binding.	It is also known as run-time run-time polymorphism or dynamic polymorphism or late-binding.

Example: *//It is method hiding. but not overriding.*

```
public class P {
    public static void m1 () {
        System.out.println("Parent");
    }
}

public class C extends P {
    public static void m1 () {
        System.out.println("Child");
    }
}

class Test {
    public static void main (String[] args) {
        P p = new P ();
        p.m1 (); // parent

        C c = new C ();
        c.m1 (); //child

        P p1 = new C ();
        p1.m1 (); // parent
    }
}
```

If both parent and child classes are non-static then it will become overriding. In this case

```
//output:    Parent
            Child
            Child
```

Overriding with respect to Var-arg method

We can override var-arg method with another var-arg method only. If we are trying to override with normal method, then it will become overloading but not overriding.

Example: *//It is overloading. but not overriding*

```
public class P {
    public void m1 (int... x) {
```

```

        System.out.println("Parent");
    }
}
public class C extends P {
    public void m1 (int x) {
        System.out.println("Child");
    }
}
class Test {
    public static void main (String[] args) {

        P p = new P ();
        p.m1 (10); // parent method

        C c = new C ();
        c.m1(10); //child method

        P p1 = new C();
        p1.m1(10); //Parent method
    }
}

```

In the above program, if we replace child method with var-arg method then it will become overriding. In this case the output:
parent
Child
Child

Overriding with respect to Variables

Variable resolution always takes care by compiler based on reference type irrespective of whether the variable is static or non-static. Overriding concept only applicable for methods but not for variables.

```

public class P {
    int x =10;
}
public class C extends P {
    int x =20;
}
class Test {
    public static void main (String[] args) {
        P p = new P ();
        System.out.println(p.x); // 10

        C c = new C ();
        System.out.println(c.x); // 20

        P p1 = new C();
        System.out.println(p1.x); // 10
    }
}

```

	P → non-static c → non-static	P → static c → non-static	P → non-static c → static	P → static c → static
P p = new P (); p.x;	10	10	10	10
C c = new C (); c.x;	20	20	20	20
P p1 = new C (); P1.x;	10	10	10	10

Differences between Overloading and Overriding

Property	Overloading	Overriding
Method names	Must be same.	Must be same
Argument types	Must be different [at least order].	Must be same [including order]
Method signature	Must be different.	Must be same
Return types	No rules/ restriction.	Must be same until 1.4V. From 1.5V onward, co-variant return types allowed.
Private, static n final method	Can be overloaded.	Cannot be overridden.
Access Modifier	No restrictions	The scope of access modifier cannot be reduced, but we can increase.
Throws clause	No restriction	If Child class method throws any checked exception, compulsory parent class should throw same checked exception or its parent but no restriction for unchecked exception.
Method resolution	Always taken care by compiler based on reference type.	Always takes care by JVM based on reference type object.
Also known as	Compile-time polymorphism, Static binding, Early binding	Run-time polymorphism, Dynamic polymorphism, Late binding

Q) If following method ***Public void m1 (int x) throws IOException*** are present in parent class then which of the following are true in child class.

1. **Public void m1 (int i);** → **overriding**
2. **Public static int m1 (long l);** → **overloading**
3. **Public static void m1 (int i);**
4. **Public void m1 (int i) throws Exception**
5. **Public static abstract void m1 (double d);** //CE: illegal combination of modifier.

Polymorphism

One name but multiple forms are the concept of polymorphism.

Example 1: method name is the same, but we can apply for different types of argument (**overloading**).

➤ **abs (int) abs (long) abs (float)**

Example 2: method signature is same, but in parent class one type of implementation and in the child class another type of implementation (**Overriding**).

```
public class P {  
    marry () {  
        System.out.println ("X");  
    }  
}  
  
public class C extends P {  
    marry () {  
        System.out.println("Y");  
    }  
}
```


Example 3: Uses of parent references to hold child object is the concept of polymorphism.

```
List l = new ArrayList/LinkedList/ Stack/Vector ();
P p = new C ();
p.m1 ();
p.m2 (); //CE: cannot find symbol, Symbol: method m2 () Location: class P
```

Parent class reference can be used to hold child object but by using that reference we can call only the methods available in parent class and we can't call child specific methods.

But by using child reference we can call both parent and child class methods.

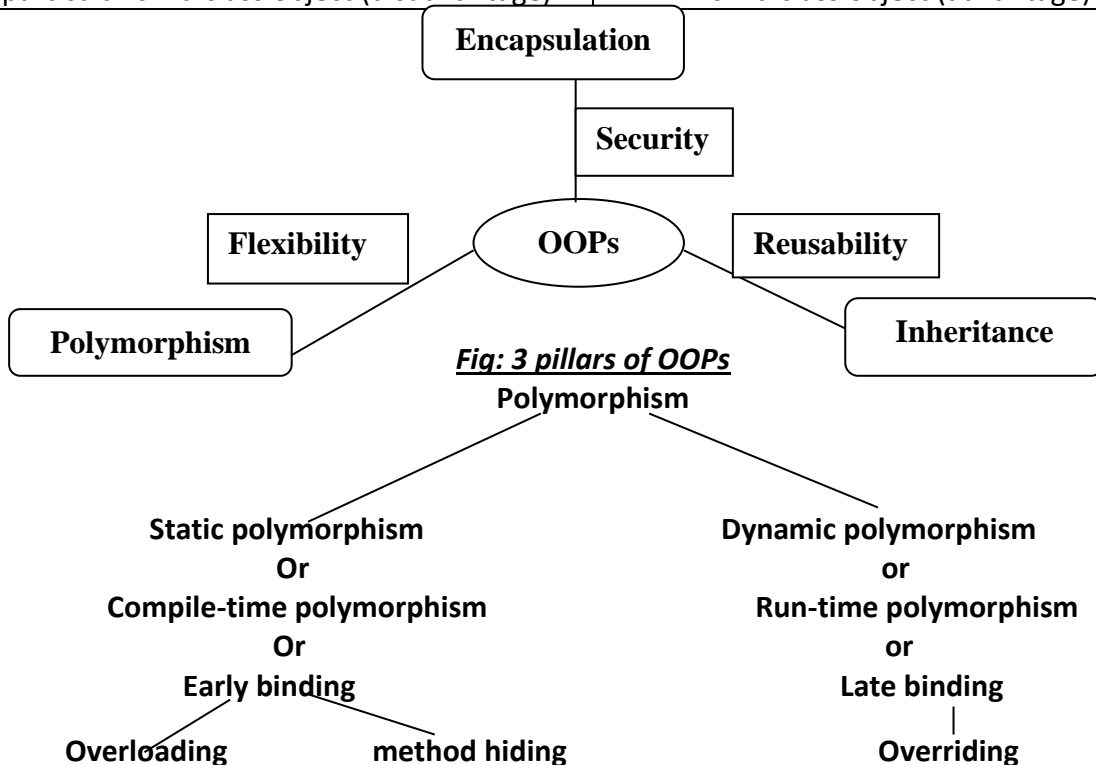
```
C c = new C ();
c.m1 ();
c.m2 ();
```

Q) When we should go for parent reference to hold object?

Ans: If we don't know exact run-time type of object then we should go for parent reference.

Example: first element in the ArrayList can be any type. It may be student object/customer object/ String object/ String buffer object. Hence the return type of get method is object, which can hold any object.

C c = new C();	P p = new C();
ArrayList l = new ArrayList ();	List l = new ArrayList ();
We can use this approach if we know exact run-time type of object.	We can use this approach if we don't know exact run-time type of object.
By using child reference, we can call both parent class and child class methods (advantage).	By using parent reference, we can call only methods available in parent class and we can't call child class methods (disadvantage).
We can use child reference to hold only particular child class object (disadvantage).	We can use parent reference to hold any child class object (advantage).



Coupling

The degree of dependency between the components is called coupling.

If dependency is more, then it is considered as tightly coupling. If dependency is less, then it is considered as loosely coupling.

```
public class A {
    static int i = B.j;
}
public class B {
    static int j = C.k;
}
public class C {
    static int k = D.m1();
}
public class D {
    public static int m1 () {
        return 10;
    }
}
```

The above components are said to be tightly coupled with each other because dependency between the components is more.

Tightly coupling is not a good programming practice because it has several serious disadvantages.

- Without affecting remaining components, we can't modify any component and hence enhancement will become difficult.
- It suppresses reusability.
- It reduces maintainability of the application.

Hence, we must maintain dependency between the components as less as possible i.e. loose coupling is a good programming practice.

Cohesion

For every component, a clear well-defined functionality is required then that component is said to be follow high cohesion.

High Cohesion is always a good programming practice because it has several advantages.

1. Without affecting remaining component, we can modify any component hence enhancement will become easy.
2. It promotes reusability of the code.
3. It improves maintainability of the application.

Note: Loosely coupling and high cohesion are good programming practice.

Object Typecasting

- We can use parent reference to hold child object.

Example: `Object o = new String ("durga");`

- We can use interface reference to hold implemented class object.

Example: `Runnable r = new Thread ();`
`Object o = new String ("durga");`
`StringBuffer sb = (StringBuffer) o; //RE`
`A b = (C) d;`
`A → class/ interface name`
`b → name of reference variable`
`C → class/ interface name`

d → reference variable name

Mantra 1: Compile-time checking 1

The type of 'd' and 'C' must have some relation either (child to parent or parent to child or same-type) otherwise we will get **CE: inconvertible type, Found: d, Required: C**

Example 1: Object o = new String ("durga");
 StringBuffer sb = (StringBuffer) o;

Example 2: String s = new String ("durga");
 StringBuffer sb = (StringBuffer) s; **//CE: inconvertible type**
Found: Java.lang.String Required:Java.lang.StringBuffer

Mantra 2: Compile-time checking 2

'C' must be either same or derived type of 'A' otherwise we will get **CE: incompatible types**
Found: C Required: A

Example 1: Object o = new String ("durga");
 StringBuffer sb = (StringBuffer) o;

Example 2: Object o = new String ("durga");
 StringBuffer sb = (String) o; **//CE: incompatible type Found:**
Java.lang.String Required: Java.lang.StringBuffer

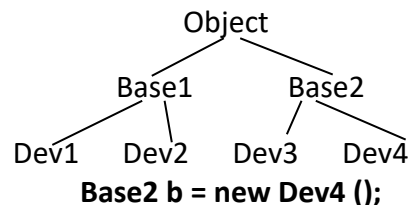
Mantra 3: Run-time checking 1

Run-time object type of 'd' must be either same or derived type of 'c' otherwise we will get **RE: ClassCastException.**

Example 1: Object o = new String ("durga");
 StringBuffer sb = (StringBuffer) o; **//RE: ClassCastException:**
Java.lang.String cannot cast to Java.lang.StringBuffer

Example 2: Object o = new String ("durga");
 Object o1 = (String) o;

Q)



Object o = (Base2) b;

Object o = (Base1) b; //CE: inconvertible type, found: Base2, Required: Base1

Object o = (Dev3) b; //RE: ClassCastException

Base2 b1 = (Base1) b; //CE: inconvertible type, found: Base2, Required: Base1

Base1 b1 = (Dev4) b; //CE: incompatible types, found: Dev4, Required: Base1

Base1 b1 = (Dev1) b; //CE: inconvertible types, found: Base2, Required: Dev1

Through type-casting, we are not creating any new object. For the existing object we are providing another type of reference variable, i.e. we are performing type-casting but not object casting.

Example 1: String s = new String ("dhruv");
 Object o = (Object) s;
 Object o = new String ("dhruv");

Example 2: Integer I = new Integer (10);
 Number n = (Number) I;
 Object o = (Object) n;
 Number n = new Integer (10);
 Object o = (Object) n;

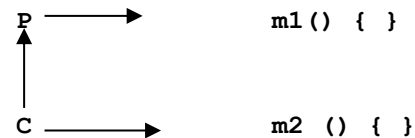
```
Object o = new Integer (10);
```

Note:

```
C c = new C ();
B b = new C (); → (B) C
A a = new C (); → (A) ((B) C)
```

Example 1:

```
C c = new C ();
c.m1 ();
c.m2 ();
((p)c).m1 ();
((p)c).m2 ();
```



Reason: parent reference can be used to hold child object but by using that reference we can't call child specific methods and we can call only methods available in parent class.

Example 2:

```
A → m1 () {Syso ("A");}
↑
B → m1 () {Syso ("B");}
↑
C → m1 () {Syso ("C");}
C c = new C ();
c.m1 () → C
((B)c).m1 () → C
((A)((B)c)).m1 () → C //It is overriding, method resolution always based on
run-time object.
```

Example 3:

```
A → static m1 () {Syso ("A");}
↑
B → static m1 () {Syso ("B");}
↑
C → static m1 () {Syso ("C");}
C c = new C ();
c.m1 () → C
((B)c).m1 () → B
((A)((B)c)).m1 () → A //Method Hiding, method resolutions always based on
reference-type.
```

Example 4:

```
A → int x =777;
↑
B → int x=888;
↑
C → int x =999;
C c = new C ();
System.out.println (c.x); //999
System.out.println ((B)c.x); //888
System.out.println ((A)((B)c.x); //777
```

Variable resolution always based on reference but not based on run-time object type.

Static control flow

Whenever we are executing a Java class the following sequence of steps will be executed as the part of static control flow.

1. Identification of static members. [1 – 6]
2. Execution of static variable assignment and static blocks from top to bottom. [7 - 12]
3. Execution of main method. [13-15]

```
public class Base {
    static int i =10; //1.static int i i = 0[RIWO] //7.i =10; i = 10[R&W]
    static { //2.static
        m1 (); //8.m1 ();
        System.out.println("First Static Block");//10.First Static Block
    }
    public static void main (String[] args) { //3.public static void main
        m1 (); //13.
        System.out.println("Main Method");//15.Main Method
    }
    public static void m1 () { //4.public static void m1
        System.out.println(j); //9.(j) //14.(j)
    }
    static { //5.static
        System.out.println("Second Static Block");//11.Second Static Block
    }
    static int j =20; //6.static int j J = 0[RIWO] //12.j=20 J = 20[R&W]
}
```

Output: 0
First Static Block
Second Static Block
20
Main Method

RIWO (Read indirectly Write only)

- Inside static block if we are trying to read a variable that is called direct read.
- If we are calling a method, within that method there is a read operation that is indirect read.

```
public class Base {
    static int i =10;
    static {
        m1 ();
        System.out.println(i); //Direct Read
    }
    public static void m1 () {
        System.out.println(i); //Indirect Read
    }
}
```

- If a variable is just identified by the JVM and original value not yet assigned, then the variable is said to be in Read Indirectly Write Only state (RIWO).
- If a variable is in RIWO state, then we can't perform direct read but we can perform indirect read. If we are trying to read directly then **CE: illegal forward reference.**

<pre>public class Test { static int x =10; static { Syso(x); } } //Output: 10 //RE: NoSuchMethodError: main</pre>	<pre>public class Test { static { Syso(x); } static int x =10; } //Output: x = 0[RIWO]</pre>	<pre>public class Test { static { m1 (); } public static void m1 () { Syso(x); } }</pre>
---	--	--

	<code>//CE: ILLegalForwardReference</code>	<code>} static int x =10; } //Output: 0 //CE: NoSuchMethodError: main</code>
--	--	---

Static Block

Static blocks will be executed at the time of class loading. Hence at the time of class loading, if we want to perform any activity, we must define that inside static class.

At the time of Java class loading the corresponding native library should be loaded. We must define this activity inside static block.

```
public class Test {
    static {
        System.loadLibrary("native library path");}}

```

After loading every database driver class, we must register driver class within driver manager but inside database driver class there is a static block to perform this activity and we are not responsible to register explicitly.

```
public class DBDriver {
    static {
        //Register the Driver with Driver Manager
    }
}

```

Within a class we can declare any no. of static blocks but all these static blocks will be executed from top to bottom.

Q 1) without writing main method is it possible to print some statement to the console?

Ans: Yes, by static block.

```
public class Test {
    static {
        System.out.println("Hello, I can print.");
        System.exit(0);
    }
}

```

Q 2) Without writing main method and static block, is it possible to print some statement to the console?

Ans: First way

```
public class Test {
    static int x = m1();
    public static int m1 () {
        System.out.println("Hello, I can print.");
        System.exit(0);
        return 1;
    }
} //Hello, I can print.

```

Second way

```
public class Test {
    static Test t = new Test();
    {
        System.out.println("Hello, I can print.");
        System.exit(0);
        return 1;
    }
}

```

Third Way

```
public class Test {
    static Test t = new Test();
}

```

```

Test () {
    System.out.println("Hello, I can print.");
    System.exit(0);
}
} //Hello, I can print.

```

Note: From 1.7V onwards, main method is mandatory to start a program execution. Hence from 1.7V onwards, without writing main method it is impossible to print some statement to console.

Static control Flow (Parent- Child relation)

1. Identification of static member from parent to child [1 to 11]
2. Execution of static variable assignment and static block only in parent to child [12 to 22]
3. Execution of only child class Main methods [23-25]

```

public class Base {
    static int i =10; //1.static int i i=0[RIWO] //12.int i =10;
    static { //2.static
        m1 (); //13.m1 ();
        System.out.println("Base Static Block");//15.Base Static Block
    }
    public static void main (String[] args) { //3.public static void main
        m1 ();
        System.out.println("Base Main");
    }
    public static void m1 () { //4.public static void m1
        System.out.println(j); //14. j
    }
    static int j =20; //5.static int j j= 0[RIWO] //16.int j =20;
}
public class Derived extends Base {
    static int x =100; //6.static int x x= 0[RIWO] //17.int x =100;
    static { //7.static
        m2 (); //18.m2 ();
        System.out.println("Derived 1st Static Block");//20.Derived 1st Static Block
    }
    public static void main (String... args) { //8.public static void main
        m2 (); //23.m2 ();
        System.out.println("Derived Main");//25.Derived Main
    }
    public static void m2 () { //9.public static void m2
        System.out.println(y); //19.(y); //24.(y);
    }
    static { //10.static
        System.out.println("Derived 2nd Static Block");//21 Derived 2nd Static Block
    }
    static int y =200; //11.static int y y= 0[RIWO] //22.int y =200;
}

```

```

Output:      Javac Derived
             0
             Base Static Block
             0
             Derived First Static Block
             Derived Second Static Block
             200
             Derived Main method

             Javac Base
             0
             Base Static Block
             20

```

Base Main method

Note: Whenever we are loading child class, automatically parent class will be loaded, but whatever loading parent class, child class won't be loaded (because parent class member by default available to the child class whereas child class member by default won't available to parent).

Instance Control Flow

Whenever we are executing a Java class, first static control flow will be executed. In the static control flow if we are creating an object the following sequence of events will be executed as a part of instance control flow.

1. Identification of instance member from top to bottom [3 to 8]
2. Execution of instance variable and instance block from top to bottom[9 to 14]
3. Execution of parent constructor [15]

```
public class Test {  
    int i =10; //3.int i //9.int i =10;  
    { //4.  
        m1 (); //10.m1 ();  
        System.out.println("First Instance Block");//12. Base Static Block  
    }  
    Test () { //5. Test  
        System.out.println("constructor"); //15. constructor  
    }  
    public static void main (String... args) { //1. public static void main  
        Test t = new Test(); ---1 //2. Test t = new Test();  
        System.out.println("Main Method");//16. Main Method  
    }  
    public void m1 () { //6. public void m1  
        System.out.println(j); //11. (j)  
    }  
    { //7.  
        System.out.println ("Second Instance Block");//13. Second Instance Block  
    }  
    int j =20; //8.int j //14.int j =20;  
}
```

Output: 0
First Instance block
Second Instance Block
Constructor
main

If we comment line 1 then output: main

Note:

1. Static control flow is 1-time activity which will perform at the time of class loading but instance control flow is not 1-time activity and it will perform for every object creation.
2. Object creation is the costliest operation, if there is no specific requirement then it is not recommended to create object

Instance control flow in parent to child relationship.

Whenever we are creating child class object the following sequence of events will be performed automatically as the part of instance control flow.

Sequence of Action

1. Identification of instance member from parent to child [4 to 14]
2. Execution of instance variable assignment & instance block only in parent class[15 to 19]

3. Execution of parent constructor [20]
4. Execution of instance variable assignment and instance block in child class [21-26]
5. Execution of child constructor [27]

```

public class Parent {
    int i =10; //4.int i //15. i=10;
    { //5.
        m1 (); //16.m1 ();
        System.out.println("Parent Instance Block");//18. Parent Instance Block
    }
    Parent () { //6.Parent ()
        System.out.println("Parent constructor");//20. Parent constructor
    }
    public static void main (String... args) { //1. public static void main
        Parent t = new Parent();
        System.out.println("Parent Main Method");
    }
    public void m1 () { //7. public void m1 ()
        System.out.println(j); //17. System.out.println(j)
    }
    int j =20; //8.int j //19.int j =20;
}

public class Child extends Parent {
    int x =100; //9.int x //21.int x =100;
    { //10.
        m2 (); //22. m2 ();
        System.out.println("Child 1st Instance Block");//24. Child 1st Instance Block
    }
    Child () { //11.Child ()
        System.out.println("Child constructor");//27. Child constructor
    }
    public static void main (String... args) { //2. public static void main
        Child c = new Child(); //3. Child c = new Child();
        System.out.println("Child Main");//28. Child Main
    }
    public void m2 () { //12. public void m2 ()
        System.out.println(y); //23. (y);
    }
    { //13.
        System.out.println("Child 2nd Instance Block");//25. Child 2nd Instance Block
    }
    int y =200; //14.int y //26.int y =200;
}

```

Javac Parent.Java -> parent.class/child.class

```

Java Child:  0
              Parent Instance Block
              Parent constructor
              0
              Child First Instance Block
              Child Second Instance Block
              Child Constructor
              Child main

```

Example 1:

```

public class Test {
    {
        System.out.println("First Instance Block");
    }
    static {
        System.out.println("First Static Block");
    }
}

```

```

Test () {
    System.out.println("Constructor");
}
public static void main (String [] args) {
    Test t1 = new Test();
    System.out.println("Main");
    Test t2 = new Test();
}
static {
    System.out.println("Second Static Block");
}
{
    System.out.println("Second Instance Block");
}
}
//Output: First Static Block
          Second Static Block
          First Instance Block
          Second Instance Block
          Constructor
          Main
          First Instance Block
          Second Instance Block
          Constructor

```

Example 2:

```

public class Initialization{
    private static String m1 (String message)
    {
        System.out.println(message);
        return message;
    }
    public Initialization () {
        m = m1("1");
    }
    {
        m = m1("2");
    }
    String m = m1("3");
    public static void main (String [] args) {
        Object o = new Instance();
    }
}
//Output: 2
          3
          1

```

Example 2:

```

public class Initialization {
    private static String m1 (String message)
    {
        System.out.println(message);
        return message;
    }
    static String m = m1("1");
    {
        m = m1("2");
    }
    static {
        String m = m1("3");
    }
    public static void main (String [] args) {
        Object o = new Initialization();
    }
}
//Output: 1

```

```

3
2
public class Test {
    int x =10;
    public static void main (String... args) {
        System.out.println(x); //CE: non-static variable x cannot be referenced from static context
    }
}

```

Note: From static area, we can't access instance members directly because while executing static area JVM may not identify instance members.

Q) In how many ways we can create an object in Java?

1. By using new operator
`Test t = new Test ();`
2. By using newInstance method
`Test t = (Test)Class.forName ("Test").newInstance();`
3. By using Factory method
`Runtime r = Runtime.getRuntime ();`
`DateFormat df = DateFormat.getInstance ();`
4. By using clone method
`Test t1 = new Test ();`
`Test t2 = (Test)t1.clone ();`
5. By using de-serialization
`FileInputStream fis = new FileInputStream ("abc.ser");`
`ObjectInputStream ois = new ObjectInputStream (fis);`
`Dog d2 = (Dog) ois.readobject ();`

Constructor

Once we create an object, compulsory we should perform initialization then only the object is in position to respond properly.

Whenever we are creating an object, some piece of the code will be executed automatically to perform initialization of the object. This piece of the code is nothing but constructor. Hence, the main purpose of constructor is to perform Initialization of object.

```

public class Student {
    String name;
    int rollNo;
    Student (String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }
    public static void main (String[] args) {
        Student s1 = new Student("durga", 101);
        Student s1 = new Student("ravi", 102);
    }
}

```

Note: purpose of constructor is to perform initialization of an object but not to create object.

Difference between Constructor and Instance block

Main purpose of constructor is to perform initialization of object but other than initialization if we want to perform any activity for every object creation then we should go for Instance block.

(Like updating one entry in the data base for every object creating or increasing count values for every object creation etc).

But constructor and Instance Block have their own different purpose and replacing one concept with another concept may not work always.

Both constructor and Instance Block will be executed for every object creation, but Instance block first followed by constructor.

```
public class Test {
    static int count = 0;
    {
        count++;
    }
    Test () {
    }
    Test (int i) {
    }
    Test (double d) {
    }
    public static void main (String[] args) {
        Test t1 = new Test();
        Test t2 = new Test(10);
        Test t3 = new Test(10.5);
        System.out.println ("No. of object created is: " +count);
    }
}
```

Rules for writing constructor

- Name of the class and name of the constructor must be matched.
- Return types concept not applicable for constructor even void also.

```
public class Test {
    void Test() {
        //It is a method but not constructor.
        //it is possible to use a method name as class-name but it's not recommended
    }
}
```

- Only applicable modifier is public, private, protected, default.

Default constructor

- Compiler is responsible to generate default constructor (but not JVM).
- Compiler will only generate default constructor if we don't write any constructor.

Prototype of default Constructor

- It is always no-arg constructor.
- Access Modifier of default constructor is exactly same as access modifier of class (applicable only for public and default).
- It contains only one line. **Super ();** //It is a no-argument call to super-class constructor.

Programmer's code	Compiler's code
<pre>public class Test { }</pre>	<pre>public class Test { Test () { super(); } }</pre>
<pre>public class Test { }</pre>	<pre>public class Test { public Test () { super(); } }</pre>

<pre>public class Test { void Test () { } }</pre>	<pre>public class Test { public Test () { super(); } void Test () { } }</pre>
<pre>public class Test { Test () { } }</pre>	<pre>public class Test { Test () { super(); } }</pre>
<pre>public class Test { Test (int i) { Super();}}}</pre>	<pre>public class Test { Test (int i) { super(); } }</pre>
<pre>public class Test { Test () { this(10);} Test(int i) { } }}</pre>	<pre>public class Test { Test() { this(10); } Test (int i) { super(); } }</pre>

Case 1: We can take super () or this () only in first line of constructor. If we are trying to take anywhere else we will get **CE**.

```
public class Test {
    Test() {
        System.out.println("dhruv");
        super();
    }
}
//CE: call to super must be first statement in constructor
```

Case 2: Within the constructor we can take either super () or this (), not both simultaneously.

```
public class Test {
    Test() {
        super();
        this();
    }
}
//CE: call to this () must be first statement in constructor
```

Case 3: We can use super () or this () only inside constructor. If we are trying to use outside of constructor, we will get **CE**.

```
public class Test {
    public void m1() {
        super();
        System.out.println("Hello");
    }
}
//CE: call to super () must be first statement in constructor
```

Case 4: super ()/ this ()

- only in constructors.
- only in first line.
- Only one but not both simultaneously.

Difference between super(), this() and super, this

super(), this()	super, this.
These are constructor calls to call super class and current class constructor.	These are keyword to refer super class and current class instance members.

We can use only in constructor as first line.	We can use anywhere except static Area.
We can use only once in constructor.	We can use any no. of time.

```
public class Test {
    public static void main(String... args) {
        System.out.println(super.hashCode());
    }
} //CE: non-static variable super cannot be referenced from static context
```

Overloaded Constructor

- Within a class we can declare multiple constructors and all these constructors having same name but different type of arguments. Hence, all these constructors are considered as overloaded constructors. Hence overloading concept applicable for constructors.

```
public class Test {
    Test () {
        this(10);
        System.out.println("no-arg");
    }
    Test (int i) {
        this(10.5);
        System.out.println("int-arg");
    }
    Test (double d) {
        System.out.println("double-arg");
    }
    public static void main(String... args) {
        Test t1 = new Test(); //double-arg/int-arg/no-arg
        Test t2 = new Test(10); //double-arg/int-arg
        Test t3 = new Test(10.5); //double-arg
        Test t4 = new Test(101); //double-arg
    }
}
```

- For constructors, Inheritance and overriding concepts are not applicable but overloading concept is applicable.
- Every class in java including abstract class can contain constructor but interface cannot contain constructor.

```
public class Test {
    Test () {
    }
}

abstract class Test {
    Test () {
    }
}

interface Test {
    Test () {
    }
} //CE
```

- Recursive method call is a Run-time Exception saying StackOverflowError.
- But in our program, if there is a chance of Recursive constructor invocation then they won't compile, and we will get CE.

<pre>public class Test { public static void m1() { m2(); } public static void m2() {</pre>	<pre>public class Test { Test() { this(10); } Test(int i) {</pre>
--	---

<pre> m1(); } public static void main (String... args) { m1(); System.out.println("hello"); } } //CE: StackOverflowError </pre>	<pre> this (); } public static void main (String... args) { System.out.println("hello"); } } //CE: Recursive Constructor call </pre>
---	--

<pre> public class P { P () { super (); } } public class C extends P { C () { super (); } } //true </pre>	<pre> public class P { P () { super (); } } public class C extends P { C () { super (); } } //true </pre>	<pre> public class P { P (int i) { super (); } } public class C extends P { C () { super (); } } //CE: cannot find symbol symbol: constructor P()location: class P </pre>
---	---	---

Note:

- If parent class contains any argument constructors, then while writing child class we must take special care w.r.t constructors.
- Whenever we are writing any argument constructor it is highly recommended to write no-arg constructor also.

<pre> public class P { P () throws Exception{ super (); } } public class C extends P { C () { super (); } } //CE: unreported Exception Java.io.IOException in default constructor </pre>	<pre> public class P { P () throws Exception{ super (); } } public class C extends P { C () throws Exception{ super (); } } </pre>
--	--

If a parent class constructor throws any checked Exception, compulsory child class constructor should throw same checked Exception or its parent otherwise code won't compile.

Q) Which of the following is valid?

1. The main purpose of constructor is to create an object.
2. The main purpose of constructor is to perform initialization of an object.
3. The name of the constructor need not be same as class name.
4. Return type concept applicable for constructors but only void.
5. We can apply any modifier for constructor.
6. Default constructor generated by JVM.
7. Compiler is responsible to generate default constructor.
8. Compiler will always generate default constructor.
9. If we are not writing no-arg constructor then compiler will generate default constructor.
10. Every no-arg constructor is always default constructor.
11. Default constructor is always no-arg constructor.
12. The first line inside every constructor should be either super () or this (), if we are not writing anything then compiler will generate this ().
13. For constructors, both overloading and overriding concepts are applicable
14. For constructors, inheritance concept applicable but not overriding.

- 15. Only concrete classes can contain constructor, but abstract classes cannot.
- 16. Interface can contain constructor.
- 17. Recursive constructor invocation is Runtime Exception.
- 18. If parent class constructor throws some checked Exception, then compulsory child class constructor should throw some checked Exception or its child.

Singleton classes

For any Java class, if we can create only one object. Such type of class is called Singleton class.

- **Example:** Runtime, BusinessDelegate, ServiceLocator

Advantage of Singleton class

- If several people have same requirement then it is not recommended to create a separate object for every requirement, we must create only one object, and we can reuse same object for every similar requirement. So that performance and memory utilization will be improved. This is core idea of singleton classes.

Example:

```
Runtime r1 = Runtime.getRuntime ();
Runtime r2 = Runtime.getRuntime ();
Runtime r3 = Runtime.getRuntime ();

...

Runtime r100000th = Runtime.getRuntime ();
```

Q) How to create our own singleton classes?

Approach 1: We can create our own singleton classes for this we have to use private constructor and private static variable and public factory method.

```
public class Test {
    private static Test t = new Test();
    private Test () {
    }
    private static Test getTest () {
        return t;
    }
}

Test t1 = Test.getTest();
Test t2 = Test.getTest();
```

Note: Runtime class is internally implemented by using this approach.

Approach 2: private constructor mainly uses to create our own singleton classes.

```
public class Test {
    private static Test t = null;
    private Test () {
    }
    public static Test getTest () {
        if(t == null) {
            t = new Test();
        }
        return t;
    }
}

Test t1 = Test.getTest();
Test t2 = Test.getTest();
```

Here class is not final, but we are not allowed to create child classes. How it is possible?
By declaring every constructor as private, we can restrict child class creation.


```
public class P {  
    private P () {  
    }  
} //For above class it is impossible to create child class
```