Object – Oriented Programming

Topics

Module 5: OOPs (Object Oriented Programming)	
1.	Data hiding, Abstraction, Encapsulation
2.	IS_A_Relationship (Inheritance)
3.	HAS_A_Relationship (Composition, Aggregation)
4.	Method signature
5.	Polymorphism - Overloading
6.	Polymorphism – Overriding
7.	Polymorphism – Method Hiding
8.	Static control flow
9.	Instance control flow
10.	Constructor
11.	Singleton class
12.	Coupling, Cohesion
13.	Type – casting

1. Data hiding, Abstraction, Encapsulation

Data hiding

- Outside person can't access our internal data directly or our internal data should not go out directly. This OOPs feature is nothing but **data hiding**.
- Only after validation/ Identification, outside person can access our internal data.
  E.g. After providing proper username & password, we can access our gmail inbox information.

- By declaring data member (variable) as private, we can achieve data hiding & it is highly recommended to declare data member (variables) as private.

- The main advantage of data hiding is Security.

After validation only we will get balance

```
public class Account {

    private double balance;
    //....

    private double getBalance () {
        // validations
        return balance;
    }
}
```

Abstraction

- Hiding internal implementation & just highlighting a set of services what we're offering, is the concept of Abstraction.
  E.g. Through Bank ATM GUI screen, bank people are highlighting the set of services what they're offering without highlighting internal implementation.

- The main advantages of Abstraction are: -
  a) We can achieve security because we're not highlighting our internal implementation.
  b) Without affecting outside person, we can able to perform any type of changes in our internal system & hence enhancement will become easy.
  c) It improves maintainability of the application.
  d) It improves easiness to use our system.

- By using interfaces (Fully abstraction) & abstract (Partial abstraction) classes, we can implement abstraction.

Encapsulation

- The process of binding data & corresponding methods into a single unit is nothing but encapsulation.
- If any component follows data hiding & abstraction, such type of component is said to be encapsulated component.

i.e. **Encapsulation = Data hiding + Abstraction**

```
class Student {

    // data members

        +

    // behaviour (methods)

}
```

E.g.

```
public class Account {
    private double balance; // Data hiding

    public double getBalance () {   // Abstraction
        return balance;
    }
}
```

- The main advantages of Encapsulation area
  a) We can achieve security.
  b) Enhancement will become easy.
  c) It improves maintainability of the application.

- The main disadvantage of encapsulation is it increases length of code & slows down execution.

Tightly Encapsulated class

- A class is said to be Tightly encapsulated if each & every variable inside the class is declared as private.
- If parent class is not Tightly encapsulated then no child class is Tightly encapsulated.

2. Is_A Relationship (also known as Inheritance)

- The main advantage of Is_A relationship is Code reusability.
- By using "**extends**" keyword, we can implement Is_A relationship.

- **Conclusions about Is_A relationship**
  a) Whatever methods parent has, by default available to the child & hence on the child reference we can call both parent & child class methods.
  b) Whatever methods child has, by default not available to the parent & hence on the parent reference we can't call child specific methods.
  c) 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.
  d) Parent reference can be used to hold child object but Child reference can't be used to hold parent object.

- The main advantage of Is_A relationship is Code reusability.
- The most common methods which are applicable for any type of child, we have to define those methods in parent class. Only the child specific methods are defined in child class.

- The most common methods applicable for any java objects are defined in Object class & hence every class in java is the child class of **Object class** either directly or indirectly so that Object class methods by default available to every java class without rewriting.

  Due to this, Object class acts as root for all java classes.

- **Throwable class** defines the most common methods required for every Exception or Error classes.

  Due to this,* Throwable *class acts as root for java Exception hierarchy.

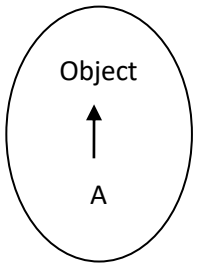
```
class P {
    public void m1 () {
        System.out.println("Parent");
    }
}
```

```
class C extends P {
    public void m2 () {
        System.out.println("Child");
    }
}
```

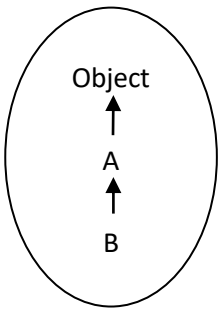
```
public class Test {
    public static void main (String [] args) {
        P p1 = new P ();
        p1.m1();    // Parent
        //p1.m2();    // Error : can't find symbol p1.m2()

        C c1 = new C ();
        c1.m1();    // Parent
        c1.m2();    // Child

        P p2 = new C ();
        p2.m1();    // Parent
        //p2.m2();    // Error : can't find symbol p2.m2()

        //C c2 = new P(); // Incompatible types : P can't be converted to C
    }
}
```

Multiple Inheritance

- A java class can't extend more than one class at a time; hence java won't provide support for multiple inheritance in classes (directly or indirectly).
- If our class doesn't extend any other class then only our class is direct child of Object class.
- If our class extends any other class then our class is indirect child class of Object.



Class A {}



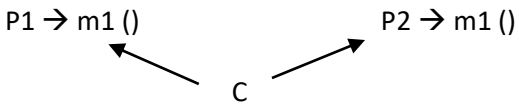
Class A extends B {}

- But interface can extend any number of interfaces simultaneously hence java provide support for multiple inheritance with respect to interfaces as there is no ambiguity problem in interfaces as there is multiple method declarations available but no definition hence we can define method as per our requirement.

  e.g. **interface C extends A, B {}**

- But strictly speaking, through interface we won't get any inheritance because we are getting only method declaration but code reusability means without rewriting that method, implementation is available but here only declaration available.

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 inheritance.



Which method to use c.m1 () ? → **Ambiguity Problem** (Diamond Ambiguity problem)

Cyclic Inheritance: - Not allowed in java. E.g. Class A extends A { }

3. Has_A Relationship (also known as Composition or Aggregation)

- There is no specific keyword to implement Has_A relationship but most of the times we are depending on new keyword.
- The main advantages of Has_A relationship is Code reusability.

Composition Vs Aggregation

No.	Composition	Aggregation
1.	Without existing container object, if there is no chance of existing contained objects then container & contained objects are **strongly associated** & this strong association is nothing but **Composition**.	Without existing container object, if there is chance of existing contained object then container & contained objects are **weakly associated** & this weak association is nothing but **Aggregation**.
2.	In composition, container object holds directly contained objects.	In aggregation, container object holds just references of contained objects.
3.	e.g. University consists of several departments. Without existing university, there is no chance of existing department; hence university & department are strongly associated & this is **Composition**.	e.g. Department consists of several Professors. Without existing department, there may be a chance of existing professor objects. Hence department & professor objects are weakly associated & this is **Aggregation**.

E.g. Car Has_A Engine reference

```
class Car {
    Engine e = new Engine ();
    //....
}

class Engine {
    //Engine specfic functionality
}
```

Is_A Relationship Vs Has_A Relationship

No.	Is_A Relationship (Inheritance)	Has_A Relationship
1.	If we want total functionality of a class automatically, then we should go for IS_A Relationship.	If we want part of the functionality of a class then we should go for HAS_A Relationship.
2.	Person class → Student class Complete functionality of Person class is required for Student class.	Test class → Demo class Partial functionality like a method or 2 of Test class is required for Demo class.

4. Method Signature

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

e.g. `public static int m1 (int i, float f)`

`m1 (int, float)`

- In C++, method signature consists of method name, argument types & return type as well but return type is not part of method signature in Java.
- Compiler will use method signature to resolve method calls.
- Within a class, 2 methods with same signature are not allowed.

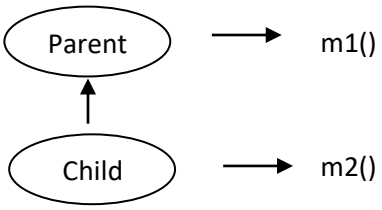
5. Polymorphism

- One name but multiple forms is the concept of polymorphism.
- Method name is same but we can apply different types of argument (**Overloading**)

  abs (int); abs (long); abs (float);
- Method signature is same but in parent class one type of implementation & in the child classes another type of implementation (**Overriding**)
- Usage of parent reference to hold child object is the concept of polymorphism.

Note: Parent class reference can be used to hold child object but by using that reference we can call only the parent class methods not child specific methods but using child reference we can call, both parent & child class methods.

e.g. Parent p = new Child ();
p.m1 (); // correct
p.m2 (); // wrong
Child c = new Child ();
c.m1 (); // correct
c.m2 (); // correct

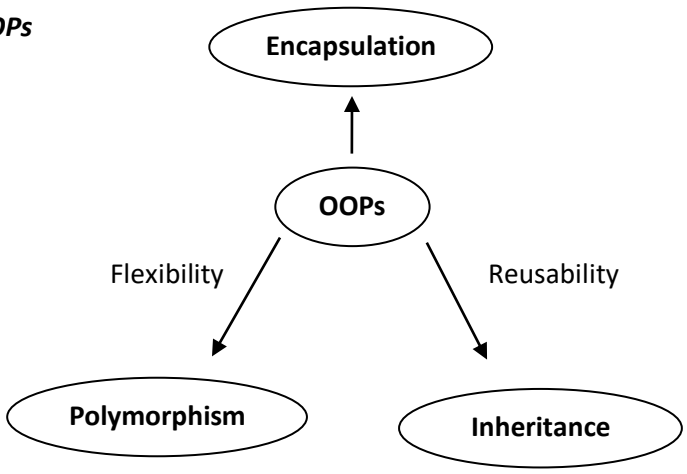


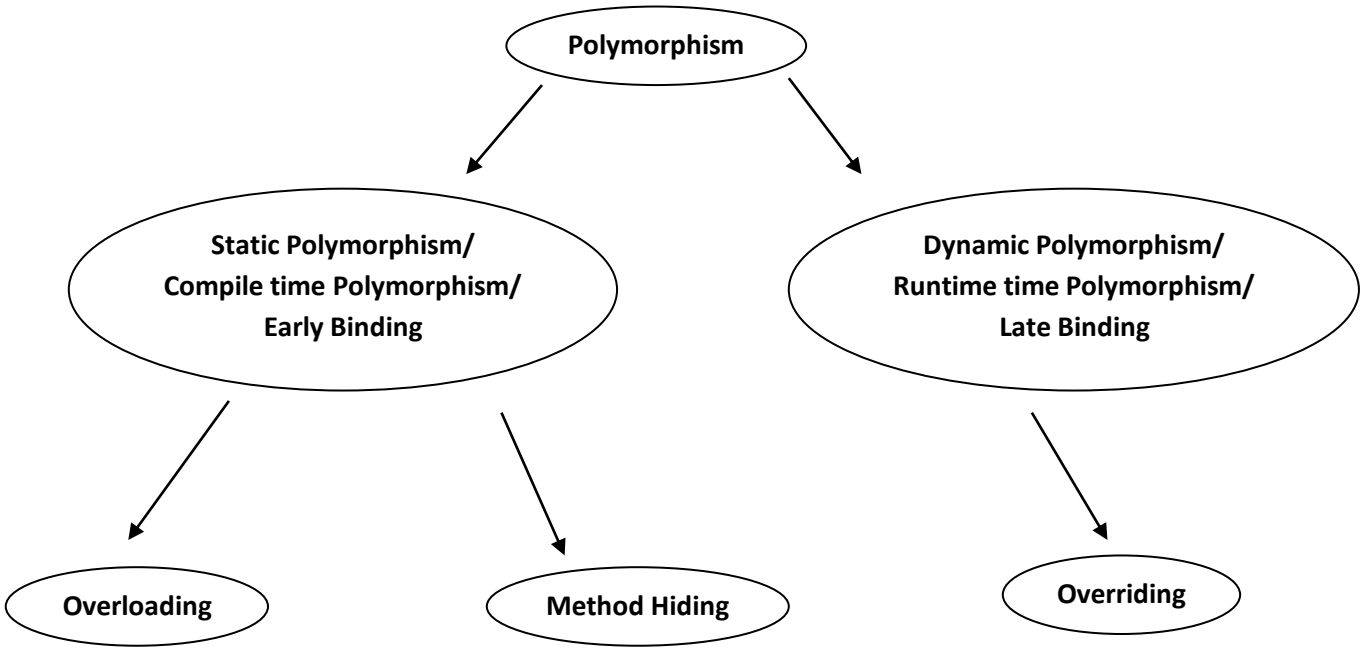
Child reference Vs Parent reference

No.	Child c = new Child ()	Parent p = new Child ()
1.	e.g. ArrayList l = new ArrayList ();	e.g. List l = new ArrayList ();
2.	We can use this approach if we know exact runtime type of Object.	We can use this approach if we don't know exact runtime type of Object.
3.	By using child reference, we can call both parent class & child class methods (this is an advantage of this approach).	By using parent reference, we can call only parent methods & we can't call child specific methods (this is a disadvantage of this approach).
4.	We can use child reference to hold only particular child class object (this is a disadvantage of this approach)	We can use parent reference to hold any child class object (this is an advantage of this approach).

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

class Child extends Parent {
    public void m1 () {
        System.out.println("Child 1");
    }

    public void m2 () {
       System.out.println("Child 2");
    }
}

public class OverridingDemo {

       public static void main(String[] args) {

         Parent p1 = new Parent();
         p1.m1();            // Parent

         Child c1 = new Child();
         c1.m1();            // Child 1
         c1.m2();            // Child 2

         Parent p2 = new Child();
         p2.m1();            // Child 1
         //p2.m2();          // Parent class reference can't call child specific methods

       }
}
```

3 Pillars of OOPs





Overloading (Compile Time Polymorphism / Early Binding)

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

  E.g. `void m1 (int i);` `void m1 (String str);`

- Having overloading concept in Java, reduces complexity of 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 (String s) {
        System.out.println("String arg");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.m1();          // no - arg
        t.m1(12);        // int arg
        t.m1("sam");     // String arg
    }
}
```

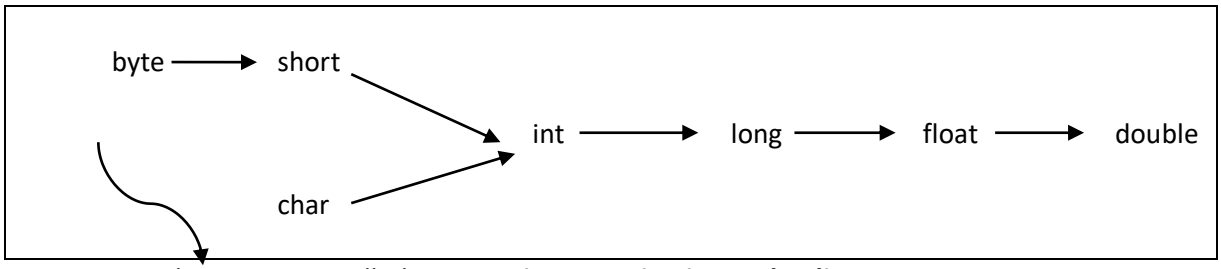
- In overloading, method resolution is always taken care by compiler based on reference type, hence Overloading is also considered as **Compile – time polymorphism** or **Early Binding.**

Cases in Overloading

Case 1: Automatic promotion in Overloading

- While resolving overloaded methods, if exact matched method is not available, then we won't get any compile time error instead compiler will promote argument to the next level & check whether matched method is available or not.
- If matched method is available then it will be considered. If matched method is not available then compiler promotes argument once again to the next level. This process will be continued until all possible promotions. Still if the matched method is not available then we will get Compile time error.

- The following are all possible promotions in overloading



This process is called **Automatic promotion in overloading**

e.g.

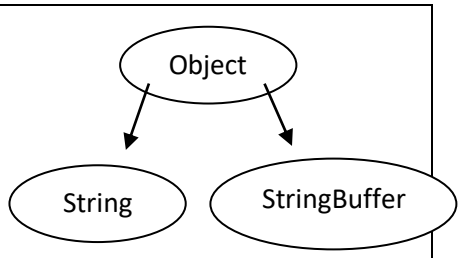
```
public class Test {

    public void m1 (int i) {
        System.out.println("int arg");
    }

    public void m1 (double d) {
        System.out.println("double arg");
    }

    public static void main (String [] args) {
        Test t = new Test ();
        t.m1(12);       // int arg
        t.m1(12.5f);    // double arg
        t.m1('a');      // int arg
        t.m1(10l);      // double arg
        t.m1(10.5);     // double arg
    }
}
```

Case 2: While resolving overloaded methods, compiler will always give the precedence for child type argument when compared with parent type argument.

```
public class Test {

    public void m1 (Object obj) {
        System.out.println("Object version.");
    }

    public void m1 (String str) {
        System.out.println("String version.");
    }

    public static void main(String[] args) {

        Test t = new Test();
        t.m1(new Object());     // Object version
        t.m1(new String());     // String version
        t.m1(null);             // String version
    }
}
```

Case 3:

```
public class Test {

    public void m1 (String str){
        System.out.println("String version.");
    }

    public void m1 (StringBuffer strBuffer) {
        System.out.println("StringBuffer version.");
    }

    public static void main(String[] args) {

        Test t = new Test();
        t.m1("srvcode");                     // String version
        t.m1(new StringBuffer("srvcode")); // StringBuffer version
        //t.m1(null);                       //error: reference to m1 is ambiguous
    }
}
```



Case 4:

```
public class Test {

    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);           // error: reference to m1 is ambiguous
        //t.m1(10.5f, 10.5f);     // error: no suitable method found for m1(float,float)
    }
}
```

Case 5: 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.

```
public class Test {

    public void m1 (int i){
        System.out.println("General int version.");
    }

    public void m1 (int... x) {
        System.out.println("var-arg int version.");
    }

    public static void main(String[] args) {

        Test t = new Test();
        t.m1();          // var-arg int version
        t.m1(10);        // General int version
        t.m1(10, 20);    // var-arg int version
    }
}
```

Case 6: In overloading method resolution always taken care by compiler based on reference type. Runtime object won't play any role.

```
class P {

}

class C extends P {

}

public class Test {

    public void m1 (P p) {
        System.out.println("Parent");
    }

    public void m1 (C c) {
        System.out.println("Child");
    }

    public static void main(String[] args) {

        Test t = new Test();

        P p1 = new P();
        t.m1(p1);              // Parent

        C c1 = new C();
        t.m1(c1);              // Child

        P p2 = new C();
        t.m1(p2);              // Parent
    }
}
```

6. Overriding (Runtime Polymorphism / Late Binding)

- Whatever methods parent class have, by default available to the child through inheritance. If child class is not satisfied with parent class implementation, then child is allowed to redefine that method based on its requirement. This process is called **Overriding**.
- The parent class method which is overridden is called **overridden method** while the child class method which is overriding is called **overriding method**.
- In overriding, method resolution always taken care by JVM based on runtime object & hence overriding is also considered as **Runtime polymorphism** or **Dynamic polymorphism** or **Late Binding**.

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

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

public class Test {

    public static void main(String[] args) {

        Test t = new Test();

        P p1 = new P();
        p1.m1();        // Parent

        C c1 = new C();
        c1.m1();        // Child

        P p2 = new C();
        p2.m1();        // Child not Parent
    }
}
```

Rules for overriding

1. In overriding, method names & argument types must be matched i.e. **method signatures must be same.**
2. In overriding, return types must be same but this rule is applicable until 1.4 version only. From 1.5 version onwards, we can take co-variant return types i.e. child class method return type need not to be same as parent method return type, its child type also allowed.

But Co-variant return type concept is applicable only for Object types but not for primitive types.

```
class P {
    public Object m1 () {
        System.out.println("Parent");
        return null;
    }
}

class C extends P {
    public String m1 () {
        System.out.println("Child");
        return null;
    }
}

public class Test {

    public static void main(String[] args) {

        Test t = new Test();

        P p1 = new P();
        p1.m1();        // Parent

        C c1 = new C();
        c1.m1();        // Child

        P p2 = new C();
        p2.m1();        // Child not Parent
    }
}
```

3. Parent class private methods are not available to the child & hence overriding concept not applicable for private methods. Based on our requirement, we can define exactly same private method in child class, **it is valid but it is not overriding.**

```
class P {                   // Class P specific method
    private void m1 () {

    }
}

class C extends P {         // Class C specific method
    private void m1 () {

    }
}
```

4. We can't override parent class final method in child classes if we're trying to override, we will get compile time error.

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

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

public class Test {

    public static void main(String[] args) {
        Test t = new Test();
        C c = new C();
        c.m1();
        //error: 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 ();
}

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

public class Test {

    public static void main(String[] args) {
        Test t = new Test();
        C c = new C();
        c.m1();            // Child
    }
}
```

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

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

abstract class C extends P {
    public 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. While overriding, we can't reduce scope of access modifier but we can increase the scope.

private < default < protected < public

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

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

Output:

```
error: m1() in C cannot override m1() in P
    void m1 () {
         ^
attempting to assign weaker access privileges; was public
```

8. If child class method throws any checked exception compulsory parent class method should throw the same checked exception otherwise, we will get compile time error. But there are no restrictions for unchecked exception.

```
import java.io.*;

class P {
    public void m1 () throws IOException {
    }
}

class C extends P {
    public void m1() throws EOFException, InterruptedException {
    }
}
```

Output:

```
error: m1() in C cannot override m1() in P
    public void m1() throws EOFException, InterruptedException {
                ^
overridden method does not throw InterruptedException
```

Overriding with respect to static methods (Method Hiding)

Case 1: We can't override a static method as non – static otherwise we will get compile time error.

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

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

Output:

```
 error: m1() in C cannot override m1() in P
    public void m1 () {
                ^
 overridden method is static
```

Case 2: Similarly, we can't override a non – static method as static.

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

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

Output:

```
 error: m1() in C cannot override m1() in P
    public static void m1 () {
                       ^
   overriding method is static
```

Case 3: If both parent & child class methods are static, then we won't get any compile time error. It seems overriding but it is not overriding & it is **Method Hiding**.

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

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

7. Method Hiding

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

No.	Method Hiding	Overriding
1.	Both parent & child class methods should be static.	Both parent & child class methods should be non – static.
2.	Compiler is responsible for method resolution based on reference type.	JVM is always responsible for method resolution based on runtime object.
3.	It is also known as Compile time Polymorphism or Static Polymorphism or Early Binding.	It is also known as Runtime Polymorphism or Dynamic Polymorphism or Late binding.

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

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

public class MethodHidingDemo {

	public static void main(String[] args) {

		P p1 = new P();
		p1.m1();		// Parent

		C c1 = new C();
		c1.m1();		// Child

		P p2 = new C();
		p2.m1();		// Parent
	}
}
```

Overriding with respect to var – arg method

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

```
class ParentVarArg {
	public void m1(int... x) {
		System.out.println("Parent");
	}
}

class ChildVarArg extends ParentVarArg {
	public void m1(int... x) {
		System.out.println("Child");
	}
}

public class OverridingVarArgDemo {

	public static void main(String[] args) {

		ParentVarArg p1 = new ParentVarArg();
		p1.m1(10);                                  // Parent

		ChildVarArg c1 = new ChildVarArg();
		c1.m1(10);                                  // Child

		ParentVarArg p2 = new ChildVarArg();
		p2.m1(10);                                  // Child
	}
}
```

Overriding with respect to variables

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

```
class ParentVariable {
		int x = 888;
}

class ChildVariable extends ParentVariable {
		int x = 999;
}

public class OverridingVariables {

	public static void main(String[] args) {

		ParentVariable p1 = new ParentVariable();
		System.out.println(p1.x);  // 888

		ChildVariable c1 = new ChildVariable();
		System.out.println(c1.x);  // 999

		ParentVariable p2 = new ChildVariable();
		System.out.println(p2.x);  // 888
	}
}
```

Difference between Overloading & Overriding

No.	Property	Overloading	Overriding
1.	Method names	Must be same	Must be same
2.	Argument types	Must be different (at least order)	Must be same (including order)
3.	Method Signature	Must be different	Must be same
4.	Return type	No Restrictions	Must be same until 1.4 v but from 1.5 v onwards Co – variant return types allowed.
5.	private, static, final methods	Can be overloaded	Can't be overridden
6.	Access Modifier	No restrictions	We can't reduce scope of access modifier but we can increase the scope
7.	Throws statements	No restrictions	If child class method throws any checked exception compulsory parent class method should throw same checked exception or its parent but no restrictions for unchecked exceptions.
8.	Method resolution	Always taken care by compiler based on reference type	Always taken care by JVM based on runtime object
9.	It is also known as	Compile time Polymorphism, Static Polymorphism, Early Binding	Runtime Polymorphism, Dynamic Polymorphism, Late Binding

In **overloading,** we have to check only method names (must be same) & argument types (must be different). We're not required to check remaining like return types, Access modifiers etc.

But

In **Overriding,** everything we have to check like method names, argument, return types etc.

8. Static control flow

- Whenever we're executing a java class the following sequence of steps will be executed as the part of static control flow: -

  Step 1: Identification of static members from top to bottom.

  Step 2: Execution of static variable assignments & static blocks from top to bottom.

  Step 3: Execution of main method.

```
public class StaticControlFlowDemo {

	static int i = 10;

	static {
		System.out.println("First static block.");
		System.out.println("i : " + i);	//Direct Read
		// System.out.println(j);	// Direct Read (CE: illegal forward reference)
		m1 ();
	}

	public static void main(String[] args) {
		System.out.println("Main method.");
		m1 ();
	}

	public static void m1 () {
		System.out.println("j : " + j);	// Indirect Read
	}

	static {
		System.out.println("Second static block.");
	}

	static int j = 20;

}

Output:

First static block.
i : 10
j : 0
Second static block.
Main method.
j : 20
```

Note:

- Inside static block if we're trying to read a variable that read operation is called as **Direct Read.**
- If we're calling a method & within that method if we're trying to read a variable, that read operation is called **Indirect Read.**
- If a variable is just identified by the JVM & original value not yet assigned then the variable is said to be **Read indirectly & write only state (RIWO).**
- If a variable is in Read indirectly & write only (RIWO) state then we can't perform direct read but we can perform indirect read. If we're trying to read directly then we will get Compile time error (CE): illegal forward reference.

Static block

- Static block will be executed at the time of class loading; hence at the time of class loading if we want to perform any activity, we have to define that activity inside static block.
- Within a class, we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

E.g.1: At the time of java class loading, the corresponding native libraries should be loaded hence we have to define this activity inside static block.

```
public class StaticBlockDemo {
	static {
		System.loadLibrary("Native library path");
	}
}
```

E.g.2: After loading every database driver class, we have to register driver classes with driver manager but inside database driver class there is a static block to perform this activity & we're not responsible to register explicitly.

Static control flow in Parent to Child relationship

- Whenever we're executing child class, the following sequence of events will be executed automatically as the part of static control flow: -
  - **Step 1:** Identification of static members from parent to child.
  - **Step 2:** Execution of static variable assignments & static blocks from parent to child.
  - **Step 3:** Execution of only child class main method.

- Whenever we're loading child class automatically parent class will be loaded but whenever we're loading parent class, child class won't be loaded as parent class members by default available to the child class whereas child class members by default won't available to the parent.

```
class Base {
	static int i = 10;
	static {
		System.out.println("Base static block.");
		m1();
	}

	public static void main (String [] args) {
		System.out.println("Base main.");
		m1();
	}

	public static void m1 () {
		System.out.println("j : " + j);
	}
	static int j = 20;
}

public class Derived extends Base {

	static int x = 100;
	static {
		System.out.println("Derived first static block.");
		m2();
	}
```

```
	public static void main (String [] args) {
		System.out.println("Derived main.");
		m2();
	}

	public static void m2() {
		System.out.println("y : " + y);
	}

	static {
		System.out.println("Derived second static block.");
	}
	static int y = 200;

}
```

`javac Base.java` → will generate 2 classes i.e. Base.class & Derived.class

Case 1: `java Derived.class`

Output:

```
Base static block.
j : 0
Derived first static block.
y : 0
Derived second static block.
Derived main.
y : 200
```

Case 2: `java Base.class`

Output:

```
Base static block.
j : 0
Base main.
j : 20
```

9. Instance Control flow

- Whenever we're executing a java class, first static control flow will be executed. In the static control flow if we're creating an object then for every object creation, the following sequence of events will be executed as a part of instance control flow:

  Step 1: Identification of instance members from top to bottom.

  Step 2: Execution of instance variable assignments & instance blocks from top to bottom.

  Step 3: Execution of constructor.

```
class Parent {

	int i = 10;
	{
		System.out.println("Parent instance block.");
		m1();
	}

	public Parent() {
		System.out.println("Parent constructor.");
	}

	public static void main(String[] args) {
		Parent P = new Parent();
		System.out.println("Parent main.");
	}

	public void m1 () {
		System.out.println("j : " + j);
	}
	int j = 20;

}

class Child extends Parent {

	int x = 100;
	{
		System.out.println("Child first instance block.");
		m2();
	}

	public Child() {
		System.out.println("Child constructor.");
	}



	public static void main(String[] args) {
		Child c = new Child();
		System.out.println("Child main");
	}

	public void m2() {
		System.out.println("y : " + y);
	}
	{
		System.out.println("Child second instance block.");
	}
	int y = 200;

}
```

`javac Parent.java` → will generate 2 classes i.e. Parent.class & Child.class

Case 1: `java Child.class`

Output:

```
Parent instance block.
j : 0
Parent constructor.
Child first instance block.
y : 0
Child second instance block.
Child constructor.
Child main
```

Case 2: `java Parent.class`

Output:

```
Parent instance block.
j : 0
Parent constructor.
Parent main.
```

- 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?

Ans: -

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 Deserialization:**

FileInputStream fis = new FileInputStream ("abc");

ObjectInputStream ois = new ObjectInputStream (fis);

Dog d2 = (Dog) ois. readObject();

10. Constructor

- Once we create an Object, compulsory we should perform initialization.
- Whenever we're 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 an object but not to create object.
- Both constructor & instance block will be executed for every object creation but instance block first followed by constructor next.
- Rules of writing constructors
    a) Name of the class & name of the constructor must be matched.
    b) Return type concept is not applicable for constructor even void also. By mistake if we're trying to declare return type for the constructor then we won't get any compile time error because compiler treats it as a method i.e., it is legal to have a method whose name is exactly same as class name.
    c) The only applicable modifiers for constructors are public, private, protected & default.

Default Constructor

- Compiler is responsible to generate default constructor.
- If we're not writing any constructor then only compiler will generate default constructor i.e. if we're writing at least one constructor then compiler won't generate default constructor. Hence every class in java can contain constructor it may be default constructor generated by compiler or customizer constructor explicitly provided by programmer but not both simultaneously.
- Prototype of default constructor
    a) It is always no – arg constructor (but no – arg constructor is not default)
    b) The access modifier of default constructor is exactly same as access modifier of class (this rule is applicable only for public & default).
    c) It contains only one line i.e. super (); [super () is a no – arg call to super class constructor]

No.	Programmer's Code	Compiler Generated Code

1.	`class Test {`
	
	`}`
		`class Test {`
		`  Test () {`
		`  super ();`
		`  }`
		`}`

2.	`public class Test {`
	
	`}`
		`public class Test {`
		`  public Test () {`
		`  super ();`
		`  }`
		`}`

3.	`public class Test {`
	`  void Test () {`
	
	`  }`
	`}`
		`public class Test {`
		`  public Test () {`
		`  super ();`
		`  }`
		
		`  void Test () {`
		
		`  }`
		`}`

4.	`class Test {`
	`  Test () {`
	
	`  }`
	`}`
		`class Test {`
		`  Test () {`
		`  super ();`
		`  }`
		`}`

5.	`class Test {`
	`  Test (int i) {`
	`  super ();`
	`  }`
	`}`
		`class Test {`
		`  Test (int i) {`
		`  super ();`
		`  }`
		`}`

6.	`class Test {`
	`  Test () {`
	`  this (10);`
	`  }`
	
	`  Test (int i) {`
	
	`  }`
	`}`
		`class Test {`
		`  Test () {`
		`  this (10);`
		`  }`
		
		`  Test (int i) {`
		`  super ();`
		`  }`
		`}`

- The first line inside every constructor should be either super () or this () & if we're not writing anything then compiler will always place super ();

Case 1: We can take super () or this () only in first line of constructor. If we're trying to take anywhere else, we will get compile time error.
Case 2: Within the constructor, we can take either super () or this () but not both simultaneously.
Case 3: We can use super () or this () only inside a constructor, if we're trying to use outside of constructor, we will get compile time error i.e. we can call a constructor directly from another constructor only.

Difference between super (), this () & super, this

No.	super (), this ()	Super, this
1.	These are constructor calls to call super class & current class constructors.	These are keywords to refer super class & current class instance members.
2.	We can use only in constructor as first line.	We can use anywhere except static area.
3.	We can use only once in constructor.	We can use any number of times.

Overloaded Constructors

- Within a class, we can declare multiple constructors & all these constructors having same name but different type of argument; hence all these constructors are considered as overloaded constructors. Hence Overloading concept is applicable for constructors.

Note:

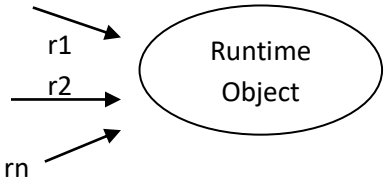
a) For constructors, inheritance & overriding concepts are not applicable but overloading concept is applicable.
b) Every class in java including abstract class can contain constructor but interface can't contain constructor.
c) If parent class contains any argument constructor, then while writing child classes, we have to take special care with respect to constructors.
d) Whenever we're writing any argument constructor, it is highly recommended to write no – arg constructor also
e) If parent class constructor throws any checked exception, compulsory child class constructor should throw the same checked exception or its parent exception otherwise the code won't compile.

11. Singleton class (Application of Private Constructor)

- For any java class, if we're allowed to create only one object such type of class is called Singleton class.
  e.g. Runtime etc.
- Objects are created by using Factory method only.
- We can create our own singleton classes. We have to use private constructor, private static variable & public Factory method.
- Advantage of Singleton class

1. If several people have same requirement, then it is not recommended to create a separate object for every requirement. We have to create only one object & we can reuse same object for every similar requirement so that performance & memory utilization will be improved.
2. This is the central idea of singleton class.

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



There are 2 approaches to create our own singleton class:

Approach 1:

```
class Singleton {
	private static Singleton s = new Singleton();

	private Singleton() {

	}

	public static Singleton getSingletonObject() {
		return s;
	}
}

public class SingletonClassApproach1 {

	public static void main(String[] args) {
		Singleton singletonObj1 = Singleton.getSingletonObject();
		System.out.println("Obj1 Hashcode : " + singletonObj1.hashCode());

		Singleton singletonObj2 = Singleton.getSingletonObject();
		System.out.println("Obj2 Hashcode : " + singletonObj2.hashCode());
	}

}
```

Output:

```
Obj1 Hashcode : 366712642
Obj2 Hashcode : 366712642
```

Approach 2:

```
import java.util.Objects;

class SingletonClass {
	private static SingletonClass s = null;

	private SingletonClass() {

	}

	public static SingletonClass getSingletonObject() {
		if(Objects.isNull(s)) {
			s = new SingletonClass();
		}
		return s;
	}
}

public class SingletonClassApproach2 {

	public static void main(String[] args) {
		SingletonClass singletonObj1 = SingletonClass.getSingletonObject();
		System.out.println("Obj1 Hashcode : " + singletonObj1.hashCode());

		SingletonClass singletonObj2 = SingletonClass.getSingletonObject();
		System.out.println("Obj2 Hashcode : " + singletonObj2.hashCode());
	}

}
```

Output:

```
Obj1 Hashcode : 366712642
Obj2 Hashcode : 366712642
```

Q. Class is not final but we're not allowed to create child classes. How is it possible?

Ans: - By declaring every constructor as private we can restrict child class creation as child class constructor super () will unable to call parent class constructor.

12. Coupling, Cohesion

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 coupled.
-

```
class A {
        static int i = B.j;
}
```

```
class B {
        static int j = C.k;
}
```

```
class C {
        static int k = D.m1();
}
```

```
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.
- Tight coupling is not a good programming practice because it has several serious disadvantages:
    a) Without affecting remaining components, we can't modify any component & hence Enhancement will become difficult.
    b) It suppresses reusability.
    c) It reduces maintainability of the application.

- Hence, we have to maintain dependency between the components as less as possible i.e. loosely coupling is a good programming practice.

Cohesion

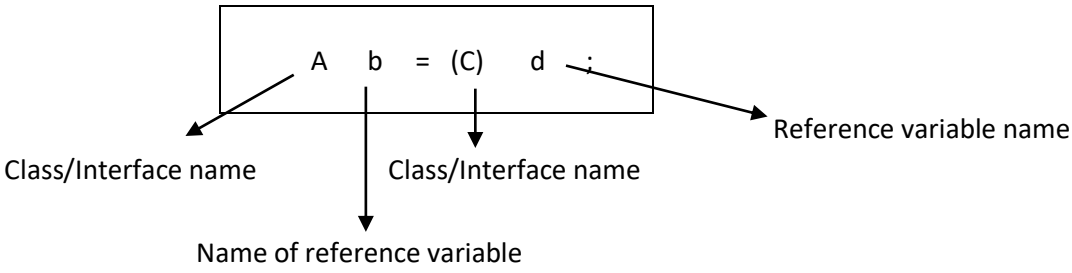
- For every component, a clear functionality is defined then that component is said to be follow High Cohesion.
- High cohesion is always a good programming practice because it has several advantages:
    a) Without affecting remaining components, we can modify any component, hence enhancement will become easy.
    b) It promotes reusability of the code.
    c) It improves maintainability of the application.

Note: Loosely Coupling & High Cohesion are good programming practice.

13. Type – casting

Object type casting

- We can use parent reference to hold child Object.
  - e.g. Object obj = new String ("srvcode");
- We can use interface reference to hold implemented class object.
  - e.g. Runnable r = new Thread ();
- Syntax



Rule 1: Compile time checking

- The type of 'd' & 'C' must have some relation either child to parent or parent to child or same type otherwise we will get compile time error saying "Inconvertible types".

e.g.
```
Object obj   =   new String ("srvcode");
StringBuffer sb   = (StringBuffer) obj; // correct

String str   =   new String ("srvcode");
StringBuffer sb   = (StringBuffer) str; // wrong
```

Rule 2: Compile time checking

- 'C' must be either same or derived type of 'A' otherwise we will get compile time error saying "incompatible types".

e.g.
```
Object obj   =   new String ("srvcode");
StringBuffer sb   = (StringBuffer) obj; // correct

Object obj   =   new String ("srvcode");
StringBuffer sb   = (String) obj;      // wrong
```

Rule 3: Runtime checking

- Runtime object type of 'd' must be either same or derived type of 'C' otherwise we will get runtime exception saying ClassCastException.

e.g.
```
Object obj   =   new String ("srvcode");
StringBuffer sb   = (StringBuffer) obj; // correct

Object obj   =   new String ("srvcode");
StringBuffer sb   = (String) obj;      // RE: ClassCastException
```