1. **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. After validation or authentication out side person can access our internal data.

E.g 1) after proving proper user name and password we can able to access our gmail inbox information

2) even though we are valid customer of the bank we can able to access our account information and we cant access others account information

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

e.g public class account{

private double balance;

public double getBalance(){

//validation

return balance;

}

}

Advantages:

* Main advantage of data hiding is **security**.

Note:

It is highly recommended to declare data member as private

1. **Abstraction :** Hiding internal implementation and just highlight the set of services what we are offering, is concept of abstraction.

**e.**g through bank ATM GUI screen bank people are highlighting the set of service what they are offering without highlighting internal implementation.

Advantages:

* We can achieve **security** because we are not highlighting our internal implementation
* Without effecting outside person, we can able to perform any type of changes in our internal system and hence **enhancement** will become easy.
* It improves **maintainability** of the application.
* It improves **easiness** to use our system.

By using interfaces and abstract classes we can implement abstraction.

1. **Encapsulation** : the process of binding data and the corresponding methods into a single unit is a nothing but encapsulation.

**e**.g

|  |  |
| --- | --- |
| class student{  Encapsulation  data members + methods  Data  } | example of capsule |

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

**Encapsulation = Data hiding + abstraction**

Public class account{ **Encapsulation**

Private double balance; **data hiding**

Public double getBalance(){ **abstraction**

//validation

Return balance;

}

}

advantages :

* We can achieve security
* Enhancement becomes easy
* It improvises maintainability of the application

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

* **Tightly encapsulation class** : A class is said to be tightly encapsulated if and only if each and every variable declared as private whether class contains corresponding getter and setter methods or not and whether these methods are declared as public or not these things we not require to check.

Public class Account{

Private double balance;

Public double getBalance{

return balance;

}}

e.g

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class a{  Private int x =10;  } | Tightly Encapsulated |  | Class a{  int x =10;  } | Not Tightly Encapsulated |
| Class b extends a{  int y =10;  } | Not Tightly Encapsulated |  | Class b extends a{  Private int y =10;  } | Not Tightly Encapsulated |
| Class c extends b {  Private int z =10;  } | Tightly Encapsulated |  | Class c extends b {  Private int z =10;  } | Not Tightly Encapsulated |

* If parent class is not tightly encapsulated then no child class is tightly encapsulated.

1. **Is-A relationship (Inheritance) :**

* it is also known as **inheritance**
* the main advantage of is a relationship is code **reusability**.
* By using **extends** keyword we can implement Is-A relationship

|  |  |
| --- | --- |
| Class Parent{  Public void m1(){  SOP(“Parent”);  }} | Class Child extends Parent{  Public void m2(){  SOP(“Child”);  }} |

|  |  |
| --- | --- |
| Parent p = new Parent();  p.m1();  p.m2(); | Will work fine  Compile Exception CannotFindSymbol M2() |
| Child c = new Child();  c.m1();  c.m2(); | Will work fine  Will work fine |
| Parent p = new Child();  p.m1();  p.m2(); | Will work fine  Compile Exception CannotFindSymbol M2() |
| Child c = new Parent(); | Compile Time exception incompatibletypeException |

* **Conclusions** :
* Whatever methods parent has by default available to the child and hence child reference we can call both parent and child class methods.
* Whatever methods child has by default not available to parent and hence on the parent reference we can not call child specific methods.
* Parent reference can be used to hold child object but using that refence we can’t call child specific methods but we can call the methods present in parent class.
* Parent reference can be used to hold child object but child reference can not be used to hold parent object.

|  |  |
| --- | --- |
| Without inheritance | With inheritance |
| Class VehicleLoan{  300 methods  }  Class HomeLoan{  300 methods  }  Class CarLoan{  300 methods  }  //In total 900 methods out of which 250 ar common method | Class loan{  250 common methods  }  Class VehicleLoan extends loan{  50 common methods  }  Class HomeLoan extends loan{  50 common methods  }  Class CarLoan extends loan{  50 common methods  }  In total 400 methods due to reusability |

**Notes:**

* The most common methods which are applicable for any type of child, we have to define it in parent class
* The specific methods which are applicable for a particular child we have to define it in child class.
* Total java API is implemented based on inheritance concept.
* The most common methods which are applicable for any java object are defined in object class and hence every class in java is child class of object 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 most common method which are required for every exception and error class hence this class acts as root for java exception hierarchy.

Object

String StringBuffer …………………… Throwable

Exception Error

RuntimeException IOException OutOfMemoryError ……

* Multiple inhertance :
* A java class cant extends more than one class at a time hence java wont provide support multiple inheritance in classes

e.g class a extends c,b{ // compile time inheritance

}

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

e.g class A{ object

} A

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

e.g class A extends B{ object

} B

A

* Either directly or indirectly java won’t provide support for inheritance with respect to class.
* There may be a chance of ambiguity problem hence java wont provide support for multiple inheritance.

c

|  |  |
| --- | --- |
| Class A{  Void M1(){  }}  Class B{  Void M1(){  }}  Class C extends A,B{  Void M1(){  }} | A B  C  Won’t support due to ambiguity issue. |

* But interface can extends any number of interface simultaneously hence java provide support for multiple inheritance with respect to interface.

e.g Interface A extends B, C{

}

* why ambiguity problem in case of interface for multiple inheritance.

Interface A { interface B{

Void m1(); void m1();

} }

Interface C extends A, B{

Void M1();

}

* even though multiple method declaration are available but implementation is unique and hence there is no chance of ambiguity problem in interfaces.
* Strictly speaking through interfaces we wont get any inheritance.
* **Cyclic inheritance :**
* Cyclic inheritance is not allowed in java of course it is not required
* Cyclic dependency involving Exception.

e.g

|  |  |
| --- | --- |
| Class A extends A{  } | Class A extend B{  }  Class B extends A{  } |

1. **Has-A relationship** :

* Has-A relationship is also known as composition or aggregation.
* There is no specipic key word to implement has-A relationship but most of the time we are depending on NEW keyword.
* The main advantage of Has-A relationship is reusability of the code.

e.g

|  |  |
| --- | --- |
| Class car{  Engine e = new Engine();  } | Class Engine{  // engine specific functionality.  } |

* Car Has-A engine reference.
* **Difference composition and aggregation.**
* Without existing container object if there is no chance of existing contained object the container and contend objects are strongly associated(coupled) and this strong association is nothing but composition.
* University consist 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.

Contained object

1University

2MCA dept

3MBA dept

4Eng dept

Container object

* Aggregation :
* Without existing container object if there is chance of existing contained object then container and contained objects weakly associated and this weak association nothing but aggregation.

e.g

- department consist of several professor, without existing department there may be a chance of existing professor object, hence department and professor objects are weakly associated and this weak association is nothing but aggregation.

5MCA dept

6P3

7P2

8P1

Container object Contained object

* In composition objects are strongly associated whereas in aggregation object are weakly associated.
* in composition container object holds directly contained object whereas in aggregation container object holds just references of contained object.
* **Is-A verses Has-A**
* If you want total functionality of a class automatically then we should go for Is-A relationship

e.g

10Student

9Persone

if you want total functionality

* If you want part of the functionality then we should go for Has-A relationship.

10Student want only part of the funcnality

if you want part functionality

9Persone with 100 method

1. **Method signature :**

* In java method signature consist of method names followed by argument types

e.g

public static int m1(int I float f) // so in this signature only m1(int I, float f) is the part of method signature.

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

e.g

|  |  |  |
| --- | --- | --- |
| Class test{  Public void m1(int i){  }  Public void m2(String s){  }} | Test t = new test();  t.m1(1);  t.m2(“string”);  t.m3(10.5); | Will work fine  Will work fine  CE : cannot find symbol |

* Within a class two methods with the same signature are not allowed.

e.g

|  |  |
| --- | --- |
| Class test{  Public void m1(int i){  }  Public void m1(in i){  }} | Test t =new Test();  t.m1(10);  CE : m1(int) is already in defined in test. |

1. **Method overloading :**

* Two methods are said to be overloaded if and only if both methods having same name and different arguments types.
* In C language method overloading concept is not available hence we cant declare multiple methods with same name but different argument types. If there is change in argument type compulsory we should go for new method name which increases complexity of programing
* But in java we can declare multiple methods with same name but different argument types such type of methods are called overloaded methods.

e.g abs(int i)

abs(log l)

abd(double d)

* Having overloading concept in java reduces complexity of programing.

|  |  |
| --- | --- |
| Class test{  Public void m1(){  Sop(“no arg”);  }  Public void m1(int i){  Sop(“int arg”);  }  Public void m1(double d){  Sop(“double arg”);  }} | Test t = new Test();  t.m1();  t.m1(10);  t.m1(10.5);  o\p  no arg  int arg  double arg |

* In overloading method resolution always takes care by compiler based on reference type hence overloading is also considered as compile time polymorphism or static polymorphism or early binding.
* **Case 1: Automatic premotion in overloading:**
* While resolving overloaded methods if exact matched method is not available then we wont get any compile time error immediately. first it will promot argument to the next level and then check whether matched method is available or not. If matched method is available then it will be considered and if the 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 over loading.

|  |
| --- |
| Byte short  Int long float double  Char |

* This process is called automatic promotion in overloading.

e.g

|  |  |
| --- | --- |
| Class test {  Public void m1(int i){  SOP(“int arg”);  }  Public void m1(float f){  SOP(“float arg”);  }} | Test t= new test();  t.m1(10); // int arg  t.m1(10.5F); // float arg  t.m1(‘a’); // int arg  t.m1(10L); // float arg  t.m1(10.5); // CE : cannot find symbol m1(double)s |

* **Case 2 : null argument in child parent**

e.g

|  |  |
| --- | --- |
| Class test {  Public void m1(Sting ){  SOP(“String arg”);  }  Public void m1(Object o){  SOP(“Object arg”);  }} | Test t= new test();  t.m1(new Object()); // Object arg  t.m1(“String”); // String arg  t.m1(null); // String arg |

* While resolving overloaded methods compiler always gives precedence for child type argument when compared with parent type argument.
* Case 3 same level data type:

|  |  |
| --- | --- |
| Class test {  Public void m1(Sting s){  SOP(“String arg”);  }  Public void m1(StringBuffer sb){  SOP(“StringBuffer arg”);  }} | Test t= new test();  t.m1(new StringBuffer());s // StringBuffer arg  t.m1(“String”); // String arg  t.m1(null); //CE : reference M1() is **ambiguous**  Object  String StringBuffer |

* **Case 4: with multiple argument :**

|  |  |
| --- | --- |
| Class test {  Public void m1(int I,float f){  SOP(“int-float arg”);  }  Public void m1(float f,int i){  SOP(“float-int arg”);  }} | Test t= new test();  t.m1(10,10.5F); // int-float arg  t.m1(10.5F,10); // float-int arg  t.m1(10,10); // CE : reference M1() is **ambiguous**  t.m1(10.5F,10.5F); // CE:cannot find symbol m1(float,float) |

* **Case 5: with var-arg datatype :**

|  |  |
| --- | --- |
| Class test {  Public void m1(int I){  SOP(“int arg”);  }  Public void m1(int… i){  SOP(“var-arg arg”);  }} | Test t= new test();  t.m1(); // var-arg arg  t.m1(10,20); // var-arg arg  t.m1(10); // int arg |

* In general, var-arg will get least priority that is if no other method matched then only var-arg method will get chance. It is exactly same as default case inside switch case.
* **Case 6: Parent-child class :**

|  |  |
| --- | --- |
| Class Animal{  }  Class Monkey extends Animal{  }  Class test {  Public void m1(Animal a){  SOP(“Animal”);  }  Public void m1(Monkey M){  SOP(“Monkey”);  }} | Test t= new test();  Animal a = new Animal();  t.m1(a); // Animal  Monkey m = new Monkey();  t.m1(m); // Monkey  Animal a1 = new Monkey();  t.m1(a1); // Animal |

* In Overloading method resolution always takes care by compiler based on reference type.
* In overloading runtime object won’t play any role.

1. **Overriding :**

* Whatever methods parent has by default available to the child through inheritance. if child class 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 overridden is called overridden method and child class method which overriding is called overriding method.

e.g

|  |  |
| --- | --- |
| Class P{  Public void property(){  SOP(“cash + gold + land”);  }  Public void marry(){  SOP(“subha lakshmi”);  }}  Class C extends Ps{  Public void marry(){  SOP(“Tina”);  }} | Class Test{  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 consider as runtime polymorphism or dynamic polymorphism or late binding.
* **Rule 1: Return type in Overriding**
* In overriding method names and argument types must be matched that is method signature must be same.
* 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 according to this, child class method return type need not be same as parent method return type, its child type is also allowed

e.g

|  |
| --- |
| Class P{  Public object m1(){  }}  Class C extends Ps{  Public String m1(){  }} |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parent class return type | Object | Number | String | double |
| Chilled class return type | Object/String/  StringBuffer | Number/Integer | Object | int |
| Acceptable | Yes | Yes | no | no |

* Co-variant return type concept applicable only for object types but not for primitive types.
* **Rule 2: Modifier in Overriding:**
* Parent class private methods not available to the child and hence overriding concept not applicable for private methods.
* Based on our requirement we can define exactly same private method it is valid but not overriding

e.g

|  |
| --- |
| Class P{  Private object m1(){  }}  Class C extends Ps{  Private String m1(){  }} |

* We cant override parent class final method in child classes if we are trying to override we will get compile time error

e.g

|  |  |
| --- | --- |
| Class P{  public final object m1(){  }}  Class C extends Ps{  Public final String m1(){  }} | CE: m1() in C cannot override m1() in P;  Overridden method is final |

* Parent class abstract methods we should override in child class to provide implementation.

e.g

|  |
| --- |
| Class P{  public abstract object m1();  }  Class C extends P{  Public final String m1(){  }} |

* We can overrider non-abstract method as abstract.

e.g

|  |
| --- |
| Class P{  public object m1(){  }}  Class C extends P{  Public abstract String m1();  } |

* The main advantage of this approach is we can stop availability of parent method implementation to the next level child classes.
* In overriding the following modifiers wont keep any restriction

1. Synchronized
2. Native
3. strictfp

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Parent method | final | Non-final | Abstract | Synchronized | native | Strictfp |
| Child method | Final /non-final | final | Non-abstract | Non- Synchronized | Non-native | Non-Strictfp |
| acceptable | No | yes | yes both way | Yes both way | Yes both way | Yes both way |

* **Rule 3 : access modifier :**
* While overriding we cant reduce scope of access modifier but we can increase the scope.

e.g

|  |  |
| --- | --- |
| Class P{  public object m1(){  }}  Class C extends P{  String m1();  } | M1() in C cannot override in m1() in P  Attempting to assign weaker access privileges was public. |

* Private < Default < Protected < Public

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parent method | public | Protected | default | Private |
| Child method | public | Protected /Public | default/Protected /Public | Not allowed |

* **Rule 4: Exception in Overriding :**
* If child class throws any checked exception compulsory parent class method should throw the same checked exception or its parent otherwise, we will get compile time error but there is no restriction for unchecked exceptions.

e.g

|  |  |
| --- | --- |
| Class P{  Public void m1() throws IOException{  }}  Class C extends P{  Public void m1() throws EOFEception, InterruptedException{  }} | CE : m1() in C cannot override m1() in P,  Overriden method does not throw java.lan.InterruptedExceptions |

|  |  |  |
| --- | --- | --- |
| **Parent method** | **Child method** | **Valid/invalid** |
| Public void m1() throws Exception | Public void m1() | Valid |
| Public void m1() | Public void m1() throws Exception | Invalid |
| Public void m1() throws Exception | Public void m1() throws IOException | Valid |
| Public void m1() throws IOException | Public void m1() throws Exception | Invalid |
| Public void m1() throws Exception | Public void m1() throws IOException, FileNotFoundException | Valid |
| Public void m1() throws IOException | Public void m1() throws EOFEception, InterruptedException | Invalid |
| Public void m1() throws IOException | Public void m1() throws AE, CSE,NPE | Valid |

* **Rule 5: Overriding with respect to static**
* We cannot override a static method as non-static otherwise we will get compile time error.

**e.g**

|  |  |
| --- | --- |
| Class P{  Public static void m1(){  }}  Class C extends P{  Public void m1(){  }} | CE: m1() in C cannot override m1() in P; Overrider method is static |

* Similarly we cant override a non-static method as static

e.g

|  |  |
| --- | --- |
| Class P{  Public void m1(){  }}  Class C extends P{  Public static void m1(){  }} | CE: m1() in C cannot override m1() in P; Overriding method is static |

* Method hiding vs Method overriding
* If both parent and child class are static then we wont get any compile time error it seems overriding concepts applicable for static method but it is not overriding and it is method hiding.

|  |  |
| --- | --- |
| Class P{  Public static void m1(){  }}  Class C extends P{  Public static void m1(){  }} | Note it is not method overriding it is method hiding. |

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

|  |  |
| --- | --- |
| Method hiding | Method overriding |
| Both parent method should be static | Both parent method should not be static |
| Method resolution always takes care by compiler based on reference type. | Method resolution always takes care by JVM based on run time objecta. |
| Also known as compile time polymorphism, static polymorphism and early binding | Also known as Run time polymorphism, dynamic polymorphism and late binding |

E.g

|  |  |
| --- | --- |
| Class P{  Public static void m1(){  SOP(“Parent”);  }}  Class C extends P{  Public static void m1(){  SOP(“Child”);  }} | Class Test{  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 class method are non-static then it will become overriding in this case output will be “parent child child”
* **Overriding with respect to var-arg :**
* We can override var-arg method with another var-arg method only if we are trying to override with normal method then it became overloading but not overriding

e.g

|  |  |
| --- | --- |
| Class P{  Public void m1(int… i){  SOP(“Parent”);  }}  Class C extends P{  Public void m1(int i){  SOP(“Child”);  }} | Class Test{  P p= new P();  p.m1(10); //Parent  C c = new C();  c.m1(10); //Child  P p1 = new C();  P1.m1(10); //Parent  } |

* In the above program if we replace child method with var-arg method then it will become overriding in this case the output is “parent child child”.
* **Overriding with respect to variable :**
* Variable resolution always takes care by compiler based on reference type irrespective of whether the variable is static or non-static.(overriding concept is only applicable for method but not for variable)

e.g

|  |  |
| --- | --- |
| Class P{  Int x= 888;  }  Class C extends P{  Int x=999;  } | Class Test{  P p= new P();  SOP(p.x) //888  C c = new C();  SOP(c.x) //999  P p1 = new C();  SOP(p1.x) //888  } |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parent:  Child : | Non-static  Non-static | Static  Non-static | Non-static  Static | Static  Static |
|  | 888  999  888 | 888  999  888 | 888  999  888 | 888  999  888 |

* Difference between overloading and overriding

|  |  |  |
| --- | --- | --- |
| **Property** | **Overloading** | **Overriding** |
| Method names | Must be same | Must be same |
| Argument types | Must be different (at lest order) | Must be same |
| Method Signature | Must be different | Must be same |
| Return types | No restriction | Must be same until 1.4 verssion but from1.5 version co-varient return type is allowed |
| Private, final , static method | Can be overloaded | Cannot be overridden |
| Access specifier | 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 method should throw the same checked exception or its parent but no restriction for unchecked exception |
| Method resolution | Always takes care by compiler based on reference type | Always takes care by JVM based on runtime object |
| It is also known as | Compile time polymorphism  Static polymorphism  Early binding | Runtime polymorphism  Dynamic polymorphism  late binding |

Note :

* In overloading we have to check only method names (must be same) and argument types (must be different). We are not require to check remaining like return type access modifier etc.
* But in overriding everything we have to check like method name, argument types, return types, access modifiers and throws class ect.
* Consider the following method in parent class

Public void m1(int x)throws IOException

In the child class which of the following method we can take

|  |  |
| --- | --- |
| Public void m1(int x) | Valid in overriding |
| Public static int m1(long x) | Valid in overloading |
| Public static void m1(int x) | Invaliding overriding due to static to non-static |
| Public void m1(int x)throws Exception | Invaliding overriding due to Exception handling |
| Public static abstract void m1(double x) | Combination of static and abstract is illegal |

* Polymorphism :
* One name but multiple forms is the concept of polymorphism

e.g 1

– method name is same but we can apply for different types of arguments(overloading)

e.g 2

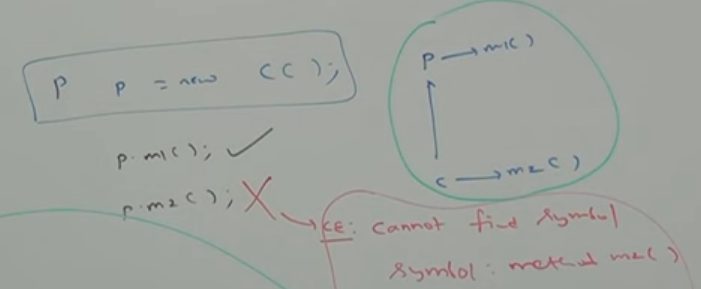
- method signature is same but in parent class one type of implementation and the in child class another type of implementation (overriding)

e.g 3

- usage of parent reference to hold child object is the concept of polymorphism.

* Parent class reference can be used to hold the child object but by using that reference we can call only the methods available in parent class and we cant call child specific methods.

e.g



* But by using child object reference we can call both parent and child reference methods

Child c= new child();

c.m1(); c.m2();

* When we should go for to parent reference to hold child object.
* If we don’t know exact runtime type of object then we should go for parent reference

e.g

the first element present in arrayslist can be any type it may be student object or customer object or string object or stringbuffer object hence the return type of get method object, which can hold any object.

Object o = l.get(0);

|  |  |
| --- | --- |
| **Child c = new Child(); or**  **ArrayList al = new ArrayList();** | **Parent p =new child(); or**  **List al = new ArrayList();** |
| 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. |
| By using child reference, we can call both parent and child method. This is the advantage of this approach. | By using parent reference, we can call only methods available in parent and we can call child specific methods (this is the disadvantage of this approach). |
| We can use child reference to hold only particular child class object (this is the disadvantage of this approach) | We can use parent reference to hold any child class object (this is the advantage of this approach) |

* **Three pillars of Opps:**

Encapsulation

Security

Opps

Reusability Flexibility

Inheritance Polymorphism

Polymorphism

Static Polymorphism dynamic Polymorphism

Or or

Compile time Polymorphism Run time Polymorphism

Or or

early binding late binding

method overloading method hiding

1. **Coupling :**

* The degree of dependency between the component is called coupling.
* If a dependency is more the it is consider as tightly coupling and if dependency is less then it is consider as loosely coupling.

e.g

|  |  |  |  |
| --- | --- | --- | --- |
| 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 m1(){  Return 10;  }} |

* The above component said to be tightly coupled with each other because dependency between the component is more.
* Tightly coupling is not a good programing practice because it has several serious disadvantages.

1. Without effecting remaining components we cant modify any component and hence enhancement will become difficult.
2. It suppresses reusability.
3. It reduces maintainability of the application.

* Hence we have to maintain dependency between the components as less as possible that is loosely coupling is good programing practice.

1. **Cohesion :**

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

e.g

total servlet

|  |  |
| --- | --- |
| Login page detail  Validation  Inbox page  Reply page  Compose page  Error page  .  .  **Contain 70 lakh lines of code** | Validation.jsp compose.jsp  Login.jsp inbox.jsp Error.jsp    Reply.jsp xyz.jsps |

**Low cohesion high cohesions**

* High cohesion is always good programing practice because it has several advantages.

1. Without effecting remaining components, we can modify any component hence enhancement will become easy
2. It promotes reusability of the code wherever validation is required we can use the same validate servlate without rewriting.
3. It improves maintainability of the application.

**Note**: Loosely coupling and high cohesion of good programing practices.s

1. **Object typecasting :**

* We can use parent reference to hold child object.

e.g

Object o = new String(“Durga”);

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

e.g

Runnable r= new Thread();

A b = (C) d;

Class/interface name reference variable name

* **Rule 1: (compile time checking one) :**
* The type of ‘d’ cand ‘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 found ‘d’ type required ‘C’.

e.g

Object o = new String(“Durga”);

StringBuffer sb = (StringBuffer)o;

e.g 2:

String s= new String(“Duraga”);

StringBuffer sb = (StringBuffer)s;

Error:

Inconvertable type

Found Java.lang.String

required Java.lang.StringBuffer

* Rue 2: (compile time checking for assignment):
* ‘C’ must be either same or derived type of ‘A’ other wise we will get compile time error saying

Incompatible type

Found C

Required A

e.g 1

Object o = new String(“Durga”);

StringBuffer sb = (StringBuffer)o;

e.g 2

Object o = new String(“Durga”);

StringBuffer sb = (String)o;

Error

Incompatible type

Found java.lang.String

Required java.lang.StringBuffer

* **Rule 3 : Run time checking:**
* Runtime object type of ‘d’ must be either same or derived type of ‘C’ otherwise we will get runtime exception saying CalssCastException

e.g 1

Object o =new String(“Durga”);

StringBuffer sb = (StringBuffer)o;

Error :

ClassCastException: Java.lang.String can not cast to Java.lang.StringBuffer

e.g 2

Object o =new String(“Durga”);

Object o1 =new (String)o;

Will work fine

object

base1 base2

der1 der2 der3 der4

Base2 b =new der4();

Object o = (base2)b; // will work fine

Object o s= (base1)b; // compile time Exception “inconvertableType Exception”

Object o = (der3)b; // Runtime Exception “ClassCastException”

Base2 b1 = (base1)b; // compile time exception “inconvertableType Exception”

Base1 b1 = (der4)b; // compile time exception “incompatibleType Exception”

Base1 b1 =(der1)b // compile time Exception “inconvertableType Exception”

* Strictly speaking through type casting we are not creating any new object. For the existing object we are providing another type of reference variable that is we are performing type casting but not object casting.

e.g 1 : String s

String s = new String(“Durga”);

Object o =(Object)s; Durga

Object o

e.g 2 :

Integer I = new Integer(10); Integer i Durga

ssNumber n = (Number)I;

Object o = (object)n; Number n

Object o

**Note :**

A 🡪 B 🡪 C (A is parent of B, B is parent of C)

C c = new C();

B b = new C(); 🡪 (B)c

A a= new C(); 🡪 (A)((B)c)

e.g 1:

P 🡪 m1()

|

C 🡪 m2();

|  |  |
| --- | --- |
| C c = new c();  c.m1() ///valid  c.m2() ///valid  ((P)c).m1() //valid  ((P)c).m2() //invalid | - parent reference can be used tos hold child object but by using that reference we cant called child specific method and we can call only the methods available in parent class |

e.g 3

|  |  |  |
| --- | --- | --- |
| Class A{  Void m1(){  SOP(“A”)  }}  Class B extends A{  Void m1(){  SOP(“B”)  }}  Class C extends B{  Void m1(){  SOP(“C”)  }} | C c = new C();  c.m1() // output will be C  ((B)c).m1() // output will be C  ((A)((B)c)).m1() // output will be C | It is overriding and the method resolution is always take place at runtime based on runtime object. |

e.g 3:

|  |  |  |
| --- | --- | --- |
| Class A{  Static Void m1(){  SOP(“A”)  }}  Class B extends A{  Static Void m1(){  SOP(“B”)  }}  Class C extends B{  Static Void m1(){  SOP(“C”)  }} | C c = new C();  c.m1() // output will be C  ((B)c).m1() // output will be B  ((A)((B)c)).m1() // output will be A | It is method hiding and the method resolution is always take place at compile time based on reference object. |

e.g 4:

|  |  |  |
| --- | --- | --- |
| Class A{  Int x=777;  }  Class B extends A{  Int x=888;  }  Class C extends B{  Int x=999;} | C c = new C();  SOP(c.x); // output will be 999  SOP(((B)c).x); // output will be 888  SOP(((A)((B)c)).x); // output will be 777 | Variable resolution is always based on reference type but not based on runtime object |

1. **Static Control flow :**

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

1. **Identification of static member from top to bottom. [1 to 6]**
2. **Execution of static variable assignment and static block from top to bottom. [7 to 12]**
3. **Execution of main method. [13 to 15]**

**e.g**

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6 | **class base{**  **static int i=10;**  **static{**  **m1();**  **SOP(“First static block”);**  **}**  **Public static void main(String[] args){**  **m1();**  **SOP(“main method”);**  **}**  **Public Static void m1(){**  **SOP(j);**  **}**  **Static{**  **SOP(“second static block”);**  **}**  **Static int j=20;**  **}** | **7**  **8**  **10**  **13**  **15**  **9 and 14**  **11**  **12** |

**Output**

**0**

**First static block**

**Second static block**

**20**

**Main methods**

* **Read Indirectly Write Only :**
* Inside static block if we are trying to read a variable that read operation is direct read.
* If we are calling a method and within that method if we are trying to read a variable that read operation is called indirect read.

e.g

class test{

static int i=10;

static{

m1(); ---------------------🡪 direct read

SOP(i);

}

Public static void m1(){

SOP(i); -----------------------🡪 indirect readss

}}

ss

* If a variable is just identified by the JVM and original value not yet assigned then the variable said to be in Read Indirectly and Write only state (RIWO)
* If a variable is in RIWO state then we cant perform direct read but we can perform indirect read.
* If we are trying to read directly the we will get compile time error saying “illegal forward reference”

e.g

|  |  |  |
| --- | --- | --- |
| Class test{  Static int x=10; 🡪 1 and 3  Static{ 🡪2  SPO(x);  }}  Output  10  NoSuchMethodError Main() | Class test{  Static{ 🡪1  SPO(x);  }  Static int x=10; 🡪2  }  Output  Illegal Forward reference | Class test{  Static{ 🡪1  M1();  }  Public static m1(){ 🡪 2  SOP(x);  }  Static int x=10; 🡪 3 and 4  }  Output  0  ssNoSuchMethodError Main() |

* 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 have to define that inside static block.

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.

Class test{

Static{

System.loadlibrary(“native library path”);

}}

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

e.g

class DbDriver{

static{

register this river with driver manager

}}

**Note :**

* Within a class we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

Q1 :Without writing main method is it possible to print some statement to the console

A1: yes by using static block

e.g class test{

static{

SOP(“hello”)

System.exit(0);

}}

output

hello

Q2: Without writing main method and static block is it possible to print some statement to the console

A2: Yes, there multiple ways.

|  |  |  |
| --- | --- | --- |
| Class test{  Static int x = m1();  Public static m1(){  SOP(“hello”)  System.exit(0);  Return 10;  }}  //Using static method | Class test{  {  SOP(“hello”)  System.exit(0);  }  }  //instance block | Class test{  Public test(){  SOP(“hello”)  System.exit(0);  }  }  //inside constructor |

**Note :**

* From 1.7 version onward main method is mandatory to start a program execution hence 1.7 version onwards without writing main method it is imposible to print some statement to the console.
* **Static control flow in parent to child relationship:**
* **Whenever we are executing child class the following sequence of events will be executed automatecaly as the part of static control flow.**

1. **Identification of static member from top to bottom and from parent to child. [1 to 11]**
2. **Execution of static variable assignment and static block from top to bottom and from parent to child. [12 to 22]**
3. **Execution of only main method of child class. [23 to 25]**

**e.g**

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5    6  7  8  9  10  11 | class base{  static int i=10;  static{  m1();  SOP(“First static block”);  }  Public static void main(String[] args){  m1();  SOP(“main method”);  }  Public Static void m1(){  SOP(j);  }  Static int j=20;  }  Class Child{  Static int x =10;  Static{  m2();  SOP(“Derived first static block”);  }  Public static void main(String[] args){  m2();  SOP(“derived main”)  }  Public static void m2(){  SOP(y)  }  Static{  SOP(Derived second static block”);  }  Static int y=200;  SS  S  }  Output  0  Base static block  0  Derived first static block  Derived second static block  200  Derived main | **12**  **13**  **15**      **14**  **16**      **17**  **18**  **20**    **19**  **21**  **22** |

**Note:**

* Whenever we are loading chilled class automatically parent class will be loaded but whenever we loading parent class child class wont be loaded (because parent class members by default available to child class whereas child class member by default won available to parent).

1. **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 a object the following sequence events will be executed as part of instance control flow

1. Identification for instance member from top to bottom.
2. Execution of instance variable assignments and instance blocks from top to bottom.
3. Execution of constructor.

**e.g :**

|  |  |  |
| --- | --- | --- |
| **3**  **4**  **5**  **1**  **2**  **6**  **7**    **8** | Class Test{  Int i=0;  {  m1();  SOP(“first instance flow”);  }  Public Test(){  SOP(“Constructor”);  }  Public static void main(String[] args){  Test t = new Test();  SOP(“main”);  }  Public void m1(){  SOP(j);  }  {  SOP(“second Instance flow”);  }  Int j=20;  }  **Output**  0  First instece flow  second instece flow  constructor  mains  if comment line one then the out will be “main” only | **9**  **10**  **12**    **15**  **11**  **13**  **14** |

**Note :**

* Static control flow is one time activity which will be performed at the time of class loading
* But instance control flow is not one time activity and it will be performed for every object creation.
* Object creation is most costly 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.

1. Identification of instance member from parent to child.[4-14]
2. Execution of instance variable assignments and instance block only in parent

class. [15-19]

1. Execution of parent constructor. [20]
2. Execution of instance variable assignments and instance block only in child

class. [21-26]

1. Execution of child constructor. [27]

|  |  |  |  |
| --- | --- | --- | --- |
| 4 & 15  5  16  18  6  20  1    7  17  8 & 19 | Class parent{  Int i=10;  {  m1();  SOP(“parent instance block”);  }  Public Parent(){  SPO(“parent constructor”);  }  Public static void main(String[] a){  Parent p =new Parent();  SOP(“parent main”)  }  Public void m1(){  SOP(j);  }  Int j=20;  } | Class child extend Parent{  Int x =100;  {  m2();  SOP(“1st Child Instance Block”)  }  Public Parent(){  SOP(“child constructor”);  }  Publi static void main(String[] a)  Child c =new Child();  SOP(“Child Main”);  }  Public void m2(){  SOP(y);  }  {  SOP(“2nd Child Instance Block”)  }  Int y = 200;  }  **output**  0  Parent instance flow  Parent contractor  0  1st Child Instance Block  2nd Child Instance Block  Child contractor  Child main | 9 & 21  10  22  11  26  2  3  27  12  23  13 & 24  14 & 25 |

e.g 2

|  |  |
| --- | --- |
| Class test{  {  SOP(“FIB”);  }  Static{  SOP(“FSB”);  }  Public test(){  SOP(“Constroctor”);  }  Public static void main(String[] args){  Test t= new test();  SOP(“main”);  Test t= new test();  }  {  SOP(“SIB”);  }  Static{  SOP(“SSB”);  }} | Output  FSB  SSB  FIB  SIB  Constructor  Main  FIB  SIB  Constructor |

e.g 3 and 4

|  |  |
| --- | --- |
| Class test{  Private static String m1(String msg){  SOP(msg);  return msg  }  Public test(){  m1(“1”ss);  }  {  m = m1(“2”);  }  m = m1(“3”);  public static void main(String[] arg){  test t = new test();  }}  **Output**  2  3  1 | Class test{  Private static String m1(String msg){  SOP(msg);  return msg  }  Static string m = m1(“1”);  {  m = m1(“2”);  }  Static{  m = m1(“3”);  }  public static void main(String[] arg){  test t = new test();  }}  Output  1  3  2 |

* From static area we can access instance members directly because while executing static area JVM may not identify instance members.

e.g

class test{

int x= 10;

public static void main(String[] args){

SOP(x); ----------------🡪 non-static variable cannot be references from static context

}}

* **In how many ways we can create an object in java / in how mays we can get object in java**

1. By NEW operator

Test t= new test();

1. By using newIntance() method

Test t = (test)Class.forName(“test”).newInstance();

1. By using factory method

Runtime r = Runtime.getRuntime();

DateFormat d = DateFormat.getInstence();

1. By using clone method

Test t1 = new test();

Test t2 =(Test)t1.clone();

1. By using deserialization

FileInputStream fis = new FileInputStream(“abc.txt”);

ObjectInputStream ois = new ObjectInputStream(fis);

Dog d =new ois.readObject();

1. **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 a object some piece of the code will be executed to perform initialization object this piece of the is nothing but constructor hence the main purpose constructor is to perform initialization of an object.

e.g

class student{

int rollno;

String name;

Public static Student(String rollno,Sting name){

This.rollno = rollno;

This.name – name;

}

Public static void main(Sting[] args){

Student s1 = new Student(1,“Shailesh”);

Student s2 = new Student(2,“Neha”);

}}

**Note :**

* The main purpose of constructor is to perform initialization of an object but not to create object.
* **Difference between constructor and instance block**
* The main purpose of constructor is to perform initialization of an object. Nut 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 DB for every object creation or incrementing count value for object creation etc)
* Both constructor and instance block have their own different purposes and replacing one concept with another may not work always.
* Both constructor and instance block will be executed for every object creation but instance block first followed by constructor.

e.g demo program to print number of object created for a class.

|  |  |
| --- | --- |
| Class test{  Static int i=0;  {  Count++;  }  Public test(){  }  Public test(int i){  }  Public test(double d){  }} | Public static void main(String[] args){  Test t = new test();  Test t = new test(10);  Test t = new test(10.5);  SOP(“number of object created for test class”+count)  } |

* **Rule for constructor** :
* name of the class and name of constructor must be matched.
* Return type concept is not applicable for constructor even void
* **By mistake if we are declare return type for the constructor then we wont get any compile time error because compiler treated it as a method.**

e.g

class test{

void test(){ // it is method but not constructor

}}

* Hence it is legal but stupid to have a method whose name is exactly same class name.
* The only applicable for contractors are public, private, protected and default if we are trying to use any other modifier we will get compile time error.

e.g

Class test{

Static test{ // compile time Exception: static modifier is not allowed here

}}

* **Default constructor:**
* Compiler is responsible to generate default constrictor (but not JVM).
* If we are not writing any contractor then only compiler will generate default constructor that is if we are 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 customized constructor explicitly provided by programmer but not both simultaneously
* **Prototype for contractor:**
* It is always no arg constructor
* The access modifier of default constructor is exactly same as access modifier of class (this rule is applicable only for **public and default**).
* It contains only one line **supper();** it is no-argument call to super class constructor.

|  |  |
| --- | --- |
| Programs code | Compiler generated code |
| Class test{  } | Class test{  Test(){  Supper();  }} |
| Public Class test{  } | Public Class test{  Public Test(){  Supper();  }} |
| Public Class test{  Void test(){  }} | Public Class test{  Public test(){  Supper();  }  Void test(){  }} |
| Public Class test{  test(){  }} | Public Class test{  test(){  Supper();  }} |
| Public Class test{  test(int i){  Supper();  }} | Public Class test{  test(int i){  Supper();  }} |
| Public Class test{  Void test(){  this(10);  }  test(int i){  }} | Public Class test{  Void test(){  this(10);  }  test(int i){  suppor()  }} |

* The first line inside every constructor should either supper() or this() and if we are not writing anything compiler will always place supper()
* **Case 1:**
* We can take supper() or this() only in first line of constructor, if we are trying to take anywhere else we will get compile time error “call to supper() must be first statement”.

e.g

class test {

test(){

SOP(“contractor”);

Supper(); // compile time Exception “call to supper() must be first statement”

}}

* **Case 2:**
* Within a constructor either supper() or this() but not both simultaneously.

**e.g**

class test {

test(){

Supper();

This() // compile time Exception “call to This() must be first statement”

}}

* **Case 3:**
* We can use supper() or this() only inside a constructor, if we are trying to use out side of constructor we will get compile time error.

**e.g**

class test {

public void m1(){

Supper(); // compile time Exception “call to supper() must be first statement”

SOP(“contractor”);

}}

* We can call a constructor directly from another constructor only.

Supper() we can use only in constructor.

This() only in first line.

Only one but not both simultaneously.

|  |  |
| --- | --- |
| **Supper() and This()** | **Supper and this** |
| These are constructor call to call supper class and current class constructor. | These are keywords to refer supper or current class instance member. |
| 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 number of times. |

e.g

class test{

public static void main(String[] args){

SOP(supper.getStudent()); // CE: non-static variable supper cannot be refer for }} a static context

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

e.g

|  |  |
| --- | --- |
| class test{  test(){  SOP(“no args”);  }  test(int i){  SOP(“int args”);  }  test(double d){  SOP(“double args”);  }} | Test t = new test(); // no args int args double args  Test t = new test(10); // int args double args  Test t = new test(10.5); // double args  Test t = new test(10l); // double args |

* For constructor 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.

e.g :

|  |  |  |
| --- | --- | --- |
| Class test{  Test(){  }} | Abstract Class test{  Test(){  }} | interface test{  Test(){  }} |
| valid | valid | Invalid |

* **Case 1 Recursive constructor error:**
* Recursive method call is a runtime exception saying StackOverFlow error.
* But in our program if there is a chance of recursive constructor invocation then the code wont compile and we will get compile time error.

**e.g**

|  |  |
| --- | --- |
| **Class test{**  **Public static void m1(){**  **m2();**  **}**  **Public static void m2(){**  **m1();**  **}**  **Public static void main(String[] args){**  **m1()**  **SOP(“hello”);**  **}}**  **//Runtime Exception: StackOverFlow Error** | **Class test{**  **test(){**  **This(10);**  **}**  **test(int i){**  **This();**  **}**  **Public static void main(String[] args){**  **SOP(“hello”);**  **}}**  **//compile time Exception :**  **Recursive constructor invocation** |

* **Case 2: constructor in Parent Child class:**

|  |  |  |
| --- | --- | --- |
| Class P{  }  Class C extends P{  } | Class P{  P(){  }}  Class C extends P{  } | Class P{  P(int i){  }}  Class C extends P{  } |
| Valid | Valid | Invalid  Compile time Exception :  Cannot find symbol constructor P() |

**Note :**

* If parent class contains any argument constructor then while writing child classes we have to take special care with respect to constructors.
* Whenever we are writing any argument constructor it is highly recommended to write no argument constructor also.
* **Case 3:**

|  |  |
| --- | --- |
| Class P{  P() throws IOException{  }}  Class C {  C(){  }}  //Compile time Exception  Java.io.IOException in default constructor | Class P{  P() throws IOException{  }}  Class C {  C() throws IOSException, Exception, Throwable{  Supper();  }}  Will work fine |

Note:

* If parent class constructor throws any checked exception compulsory child class constructor should throw the same checked exception or its parent otherwise the code wont compile.
* **Which of the following statement is valid.**

1. The main purpose of constructor is to create an object //invalid
2. The main purpose of constructor to perform initialization of an object // valid
3. The name of the constructor need not be same as class name //invalid
4. Return type concept applicable for constructor but only void //invalid
5. We can apply any modifier for constructor // invalid
6. Default constructor generated by JVM // invalid
7. Compiler is responsible to generate default constructor // valid
8. Compiler will always generate default constructor //invalid
9. If we are writing no argument constructor the compiler will generate default constructor // invalid
10. Every no argument constructor is always default constructor // invalid
11. Default constructor is always no argument constructor // valid
12. The first line inside every constructor either supper() or this() if we are not writing anything then compiler will generate This() //invalid
13. For constructors both overloading and overriding constructor applicable //invalid
14. For constructor inheritance concept applicable but not overriding //false
15. Only concrete class can contain constructor but abstract classes cannot //invalid
16. Interface can contain constructors //invalid
17. Recursive constrictor invocation is a runtime exception // invalid
18. If parent class constructor throws some checked exception the compulsory child class constructor should throw same checked exception of its child //invalid

* **Private constructor : singleton class:**
* **For any class if we are allowed to create only on object such type of class is called singleton class**

**e.g**

**1) Runtime**

**2) businessDelegate**

**3) serviceLocator**

* **Advantage of singleton class**
* If several people have same requirement, then it is not recommended to created separate object for every requirement. We have to 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 the central idea of singleton classes.

e.g

Runtime r1 = Runtime.getRuntime();

Runtime r2 = Runtime.getRuntime();

Runtime r3 = Runtime.getRuntime();

r1

r2

r3

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

Approch 1)

Class test{

Private static test t = new test();

Private test(){

}

Public static test getTest(){

Retrun t;

}}

Note :

* Runtime class is internally implemented by using this approach.

Approch 2)

Class test{

Private static test t =null;

Private test(){

}

Public static test getTest(){

If(t == null){

t = new test();

}

Retrun t;

}}

* At any point of time for test class we can create only one object hence test class is singleton class
* Class is not final but we are not allowed to create child classes.
* By declaring every constructor as private we can restrict child class creation.

Class P{

Private P(){

}}

* For the above class it is impossible to create child class.