**Object – Oriented Programming**

Topics

1. Data hiding

2. Abstraction

3. Encapsulation

4. Tightly encapsulated class

5. Is\_A Relationship

6. Has\_A Relationship

7. Method signature

8. Overloading

9. Overriding

10. Static control flow

11. Instance control flow

12. Constructors

13. Coupling

14. Cohesion

15. Type - casting

**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. 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.

|  |
| --- |
| public class Account {    private double balance;  //....    private double getBalance () {  // validations  return balance;  }  } |

> The main advantage of data hiding is Security.

*After validation only we will get balance*

**2. 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.

**3. Encapsulation**

> The process of binding data & corresponding methods into a single unit is nothing but encapsulation.

e.g.

|  |
| --- |
| class Student {    // *data members*    +    // *behaviour (methods)*  } |

> If any component follow data hiding & abstraction, such type of component is said to be encapsulated component.

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

|  |
| --- |
| public class Account {  private double balance; // Data hiding    public double getBalance () { // Abstraction  return balance;  }  } |

e.g.

> 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.

**4. 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.

**5. 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.

|  |
| --- |
| 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*      }  } |

**> 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.

> Notes

a) 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.

b) 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.***

c) **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.***

**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.

P1 🡪 m1 () P2 🡪 m1 ()

C

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

**> Cyclic Inheritance** : - Not allowed in java.

e.g. Class A extends A { }

**6. 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 container 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.

|  |
| --- |
| class Car {  Engine e = new Engine();  //....  }  class Engine {  //Engine specfic functionality  } |

Car Has\_A Engine reference

**> Is\_A 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. |

**7. Method Signature**

> In java, method signature consists of method names followed by arugment 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 is not allowed.

**8. Overloading**

> 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.

e.g.

|  |
| --- |
| 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

|  |
| --- |
| byte short    int long float double  char |

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  }  } |

i.e.

**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**  }  } |

**9. Overriding**

> 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**.

e.g.

|  |
| --- |
| 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-varient 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-varient return type concept is applicable only for Object types but not for primitive types.

e.g.

|  |
| --- |
| 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.**

e.g.

|  |
| --- |
| 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.

e.g.

|  |
| --- |
| 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.

e.g.

|  |
| --- |
| 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.

e.g.

|  |
| --- |
| 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 |

e.g.

|  |
| --- |
| 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.

e.g.

|  |
| --- |
| 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**

**> Case 1:** We can’t 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 () {  }  }  **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.

e.g.

|  |
| --- |
| 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**.

e.g.

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

**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. |

e.g.

|  |
| --- |
| **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 |