# Java OOPS

* Data hiding
* Abstraction Security
* Encapsulation
* Tightly Encapsulated
* Is A relationship
* Has A relationship
* Method signature
* Overloading
* Overriding
* Static control flow
* Instance control flow
* Constructor
* Coupling
* Cohesion
* Type casting

# Data hiding:

1. Internal data should not be accessed from outside.
2. This is achieved by declaring data member (variable) as **private**. To get data, a method has to be called which has validation logic to return the data. Advantage is data security.

Note : It is recommended to declare data member as **private**.

Ex. public class Account {

private double balance;

public double getBalance() {

//validation

Return balance ;}}

# Abstraction

1. Hide internal implementation and just highlight set of services being offered is called Abstraction.

Ex. Through ATM gui screen the bank displays only the services provided by them but the implementation is unknown to us.

By using Abstract class and Interface this can be implemented.

Advantages are

1. Security – Achieved by not highlighting internal implementation.
2. Without affecting end user, any enhancements can be implemented.
3. Improves maintainability of the application.
4. Improves easiness to use the system.

* Encapsulation – Process of binding data and corresponding methods into a single unit.

Just like a capsule contains multiple medicines as a single unit.

Ex. class Student {

Data members + behavior (methods)

}

Note : If any component follows data hiding and Abstraction, such type of component is said to be Encapsulated component.

**Encapsulation = Data hiding + Abstraction**

Ex.

public class Account {

public double getBalance() {

//validation

return balance;

}

public void updateBalance(double balance ) {

//validation

this.balance = balance ;

}

}

So in UI just 2 buttons are provided. One to get balance one to update balance. So end user will never know the implementation of the internal code.

Advantages are

1. Security – Achieved by not highlighting internal implementation.
2. Without affecting end user, any enhancements can be implemented.
3. Improves maintainability of the application.
4. Improves easiness to use the system.

Disadvantage – It increases length of code (Validations) and slows down the execution.

Tightly Encapsulated class: If all variables in the class are private then it’s a tightly Encapsulated class. No need to check other criteria.

Note : If a parent class is not tightly Encapsulated, then **none of its child** classes are tightly Encapsulated.

Ex. public class A{ int x = 2;}

public class B extends A { private int y = 3;}

public class C extends B { private int z = 4 ;}

# Is A relationship (Inheritance)

1. Uses extends keyword.
2. Main advantage is code re-usability (Parent code used by child)

Scenarios

Parent p = new Parent() ;

p.parentMethod() – valid

p.childMethod() – invalid

Child c = new Child()

c.parentMethod() – valid

c.childMethod() – valid

Parent p = new Child()

p.parentMethod() - valid

p.childMethod() – invalid. Cannot find symbol method childMethod

On parent referen

Child c = new Parent () – Invalid. Incompatible type required Child found Parent.

Total Java api is implemented based on inheritance concept. Most of St common methods which are applicable for any Java object are defined in Object class. Hence every class in Java is the child class of Object directly or indirectly so that Object class methods are available to every Java class. Hence Object is root.

Throwable class defines most common methods which are required for every Exception and Error classes. Hence this is root for Java exception hierarchy.

Note:

If our class does not extend any other class then our class is direct child class of Object.

If our class extends any other class then our class is indirect child class of Object. Ie Multi level inheritance and its not multiple inheritance.

Multiple inheritance is only available for interfaces. Since there are no implementation in interface. Technically there is no inheritance in interfaces because there is only method declarations and no implementation.

Cyclic inheritance (Class extending same class, A extends B and B extends A) is not allowed in Java and is not required also. Compile time error is thrown: Cyclic inheritance involving class name.

# Has A relationship :

Also known as Composition or Aggregation

No specific keywords to implement Has A relation, but most of the times We use new keyword

Code re usability.

Ex. Class Car {

Engine e = new Engine () ; // car HAS A engine reference.

}

**Difference between Composition and Aggregation**

Without existing container object, if there is no chance of existing contained objects, then container and contained objects are strongly associated and this **strong** association is **Composition**.

Ex. University consists of several departments. Without university, there is no chance of department to exist. Hence University and department are strongly associated which is called **Composition**.

Without existing container object if contained can still exist, then container and contained objects are **weakly** associated and this weak association is **Aggregation**.

Ex. Department consists of several professors. Without department, there are chances that professors can still exists. Hence deportment and professors objects are weakly associated called as **Aggregation**.

Note:

Composition – Strong, Aggregation – weak.

In composition, container object holds directly contained objects. In Aggregation Container objects hold only the reference of contained objects.

If total functionality is required, then go for IS A relationship. Student class must have Is A relationship, because methods of person should be in student.

If only a part of functionality is required, then use Has A relationship.

**Method signature:**

Consists of method name and argument types.

Ex:

public static int methodName(int I, float f)

The signature is - methodName(int, float)

Return type is not part of method signature in Java

Compiler uses method signature to resolve method calls.

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

# Over loading :

Two methods are said to be overloaded, off both methods have same name but different argument types. Return type are not considered. Method resolution is taken care by **compiler** based on reference type.

This is also called compile time polymorphism / static polymorphism / early binding.

*Case 1* : Automatic Promotion in Overloading

While resolving overloaded methods, if exact match method is not available then we won’t get compile time error immediately. First it will promote argument to next level and if matched method is available or not. If found, that method will be considered. If not found, then compiler promotes argument to next level until all possible promotions. Still if the matched method is not available, we get compile time error.

Promotion order:



Public void m1(int i) { sop( integer arg)}

Public void m1(float f) {sop (float arg)}

Calling –

m1(10) – integer arg

m1(10.5f) – float arg

m1(‘a’) – integer arg

m1(10l) – float arg

m1(10.5) – compilation error.

*Case 2* :

Ex :

Public void m1(String str) { sop(String arg)}

Public void m1(Object obj) {sop (Object arg)}

Calling –

m1(new Object() ) – Object arg

m1(“test name”) – String arg

m1(null) – String arg

While resolving overloaded methods, compiler gives presidence for child type argument when compared with parent type argument.

*Case 3 :*

Ex:

Public void m1(String str) { sop(String arg)}

Public void m1(StringBuilder builder) {sop (builder arg)}

Calling –

m1(“name” ) – String arg

m1(new StringBuilder (“name”) – builder arg

m1(null) – compilation error - ambiguity

*Case 4 :*

Public void m1(int I, float f) { sop(int and float)}

Public void m1(float f, int i) {sop (float and int)}

Calling –

m1(10, 10.5f) – int and float

m1(10.5f, 10) – float int

m1(10, 10) – compilation error – ambiguity ( because of promotion, both methods are possible.)

m1(10.5f, 10.5f) – compilation error : cannot find symbol

*Case 5 :*

Public void m1(int i) {sop (only int)}

Public void m1(int... x) {sop (var arg int)}

Calling -

m1() – var arg int

m1(10, 10) – var arg int

m1(10) – only int: (this gets priority because it’s legacy and to avoid breaking applications which might be using this, Java has give priority to int than “int… “.

Var arg method gets least priority. It only gets chance when there are no such arguments matching. Like default case in switch.

*Case 6:*

Class Animal{}

Class Monkey extends Animal{}

Class Test{

Public void m1(Animal a) {sop (animal version)}

Public void m1(Monkey a) {sop (monkey version)}

}

Calling –

Animal animal = new Animal () ;

o/p - m1(animal) - animal version

Monkey monkey = new Monkey () ;

o/p - m1 (monkey) - monkey version

Animal animal = new Monkey () ;

o/p - m1(animal) - animal version //method resolution depends on reference and not on object.

# Overriding:

Parent methods are default available to the child through inheritance. If child class is not satisfied with parent class implementation, child can redefine that method based on its requirement. This process is called Overriding.

Note:

In overriding method resolution is taken care by JVM based on **run-time** **object**. Hence overriding is also called as **Run-time polymorphism, dynamic polymorphism** or **late binding.**

class Test {

public static void main(String [] args){

Parent p = new Parent ();

p.marry(); //parent method. because runtime object is parent

Child c = new Child ();

c.marry(); // child method. because runtime object is child

Parent p1 = new Child ();

p1.marry(); // child method. because runtime object is child

// During compilation, compiler will check if there is marry () in Parent. At run-time JVM will check if Child has overridden that method. JVM will execute the method based on run-time object.

Hence **child method** is called.

}

|

**Rules for overriding**

1. Method name and argument types (signature) must be same.
2. Return types must be same. But this is applicable until Java 1.4 only. From v1.5, we can take co-variant return types, where Child class method return type, need not be same as parent method return type. Its child type is also allowed.

class Parent{

public **Object** m1(){ //... }}

class Child{

public **String** m1(){ //... }}

Note:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parent Class method return type | Object | Number | String | double |
| Child class method return type | Object / String / StringBuffer | Integer / Double / Number | Object | int |
|  | ***Possible*** | ***Possible*** | ***Error*** | ***Error \**** |

\*Co-variant return types are not applicable for **primitive data types.**

1. Parent class **private** methods are not available to the child. So, overriding concept not applicable for private methods.
2. Parent class **final** methods cannot be overridden. If tried so, compile time error is observed.

class Parent{

public final Object m1(){//...}}

class Child{

public String m1(){ //...}}

*m1() in Child cannot be override m1() in Parent; overridden method is final*

1. Parent class Abstract methods should be overridden to provide implementation.

abstract class Parent{

public abstract Object m1();

}

class Child{

public String m1(){

//...

}

}

1. We can override **non-abstract** methods as **abstract**

class Parent{

public void m1(){

//...

}}

abstract class Child{

public abstract void m1();

}

Main advantage is we can stop parent method implementation to next level child object.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Parent method | final | Non-final | abstract | synchronised | native | Strictfp |
| Child method | Non-final / final | final / non-final | non-abstract / abstract | non-synchronised | Non-native | Non-strictfp |
|  | ***Error*** | ***Possible (both ways)*** | ***Possible (both ways)*** | ***Possible (both ways)*** | ***Possible (both ways)*** | ***Possible (both ways)*** |

1. The scope (access modifier) of the overriding method **cannot be reduced** but can be **increased**.

class Parent{

public void m1(){

//...

}}

class Child{

void m1(){

//...

}

}

Since m1() has **default** scope, we get compilation error.

m1() in Child cannot override m1() in Parent; 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 applicable for private** |

*Why scope cannot be reduced?*

If parent method was public everyone could access it. But if child method overrides it and makes the scope to default, it will not be accessible outside. Hence reducing scope is invalid

1. Overriding and exception

Parent -> void m1() throws Exception

Child -> void m1() Valid

Parent -> void m1()

Child -> void m1() throws Exception **Invalid**

Parent -> void m1() throws Exception

Child -> void m1() throws IOException Valid

Parent -> void m1() throws IOException

Child -> void m1() throws Exception **Invalid**

Parent -> void m1() throws IOException

Child -> void m1() throws FileNotFoundException, EOFException Valid

Parent -> void m1() throws IOException

Child -> void m1() throws EOFException,InterruptedException **Invalid** (m1() in Child cannot override m1() in Parent ; overridden method does not throw InterruptedException

Parent -> void m1() throws IOException

Child -> void m1() throws ArithmaticException,NullPointerException Valid

Note:

* If child method throws any checked exception, then parent should throw the same checked exception or its parent. (Compile time error)
* No restrictions for unchecked exception.

1. Static methods and Overriding