① Data Hiding

OutSide Person can't access internal data directy (without authentication).

OR

Internal data shoudt not go outside directy without authentication.

OR More technicalty

Hiding internal data within the class to Prevent it direct access from outside class.

```
class BankAccount {

    Private double balance;
    long userId;

    Public getBalance (long userId) {
        if(this.userId == userId) { // Authentication
            return balance;
        }
        return -1;
    }
}
```

② Abstraction

Hiding the internal implementation and Just highliting the set of services is known as Abstraction.

PAGE No.
DATE / /

(3) Encapsulation

Binding data members with/under corresponding methods into a single unit (class) is known as Encapsulation.

Example:

Theoritically → Java class is an example of encapsulation

(1) class Myclass {
data members
+
methods(behaviours)
}

Encapsulation = Data Hiding + Abstraction

(2) Tightly Encapsulation

A class said tightly encapsulated if and only if all data members (variables) are declared as private.

class Myclass {
only variables should be Private { Private int age;
Private String name;

Public String getInfo() {
return name;
}

If parent class is not tightly encapsulated then its no every child class is tightly encapsulated.

IS-A Relationship known as Inheritance

```
class Parent {                                  (Super / Base class)
    public void m1() {
        S.o.p("m1");
    }
}

class Child Extends Parent {                    (Derived class)
    public void m2() {
        S.o.p("m2");
    }
}

class IsA {
    public Static void main(String[] args) {

        Parent parent = new Parent();   ✓
        parent.m1();                    ✓
        parent.m2();                    ✗

        Child child = new Child();      ✓
        child.m1();                     ✓
        child.m2();                     ✓

Compile time:
P
↑
C       Parent parent = new Child();    ✓
        parent.m1();                    ✓
        parent.m2();                    ✗   (UPCasting)

        Child child = new Parent();     ✗
Run time:
P
↓
C       child child = (child) Parent();     ✓   (DOWN Casting)
        child.m1(); ✓ child.m2(); ✓
}
```

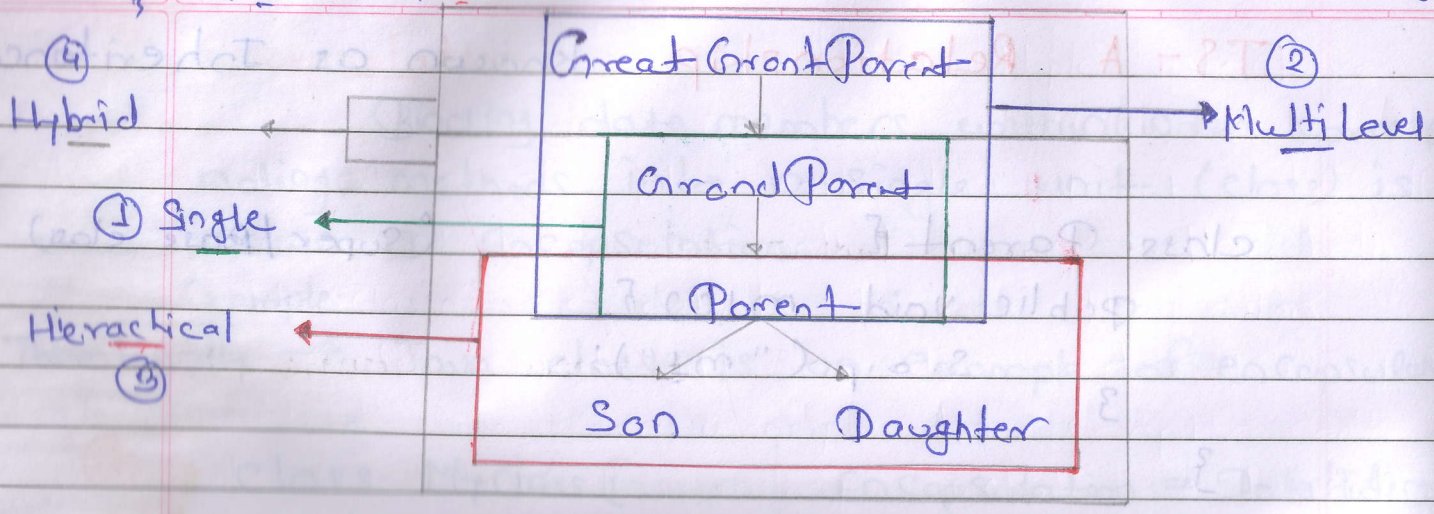
Imp

```
Parent Parent = new Child ();
Child Child = (Child) Parent;
Child.m1();
Child.m2();
}
```

Down casting

Dart 2.9 Imp

does not support down casting.



4 Inheritance

1 Single Inheritance

```
class GrandParent { }

class Parent extends GrandParent { }
```

2 Multilevel Inheritance

```
class GreatGrantParent { }

class GrandParent extends GreatGrantParent { }

class Parent extends GrandParent { }
```

3 Hierachical Inheritance

```
class Parent { }

class Son extends Parent { }

class Daughter extends Parent { }
```

Hybrid Inheritance = Single / Multilevel Inheritance
+
Hierarchical Inheritance

PAGE No.
DATE / /

④ Hybrid Inheritance

```
class GreatGrandParent { }
   class is option here
class GrandParent extends GreatGrandParent { }
                  extends is option here
class Parent extends GrandParent { }

class Son extends Parent { }

class Daughter extend Parent { }
```

⑤ Multiple Inheritance

```
class GrandParent { }
  Public void money() { = }
}class Pa

class Parent {
   public void money() { = }
}

class Son extends GrandParent, Parent {
                              ↑
}                    Compiler error
                     Ambiguity Problem

main() {
    Son son = new Son();
    Son.money();
}
```

Que. So why java doesn't provide multiple inheritance?

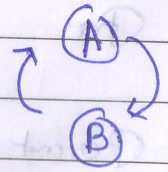
Ans:- In above example both class Parent and GrandParent classes contain (having) or contain some method money(). Here both classes are extends to (class) Son. Now when we create object of Son class and call money() method like Son.money() then compiler can't decide which class money() method have to call either from in Parent or in GrandParent. And so here ambiguity problem occurs. and thats why java does not support multiple inheritance.

Cyclic Inheritance — java does not support

① class A extends A { }

② class A extends B { }

class B extends A { }



CompileTime Error: Cyclic Inheritance involving A

```
class Parent {
    default void car() { }
    public void money() { }
    Private void property() { }
}
class child extends Parent { }
class IsA {
    main() {
        child child = new child();
```

```
child.car();
child.money();
child.property();
}
}
```

Private method should not call

~~Private members from Parent class OR methods are not available in child class~~

Java: Private members in Parent class are not available in child class. (Imp)

Dart: Private members in Parent class are available in child class if and only if both classes are in same .dart file. (Imp)

Associations → Aggration HAS-A
→ Composition Part-of

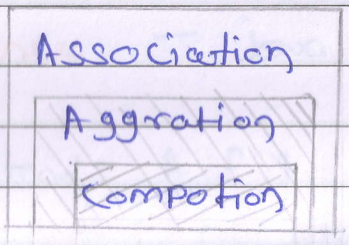
PAGE No.
DATE / /

① ~~Aggregation (HAS-A Relation)~~

- ~~Dependency~~ : ~~child~~

Associations is relation between to seperate classes with establish through their objects.

Association can be { one to one, one to many, many to one, many to many }



① Aggration ~~(HAS-A)~~ (Has-A)

- Dependency : Child object can exist independently of parent
- Type of Relation : Has-A
- Type of Association : Weekly association

Example - 1

```
class Engine { }

class Car {
    Engine engin = new Engine();
}
```

"Car has a engine."

Example - 2 :

```
class Square{
    double getSquare(double value) {
        return value * value;
    }
}

class Circle {
    square square;  //Aggrigation

    Circle(Square square) {
        this.square = square;
    }

    double area(double radious) {
        return 3.14 * square.getSquare(radius);
    }
}

double area(Square square, double radious){
    return 3.14 * Square.getSquare(radious);
}

class Rectangle {
    square square;  //Aggrigation

    Rectangle() {
        square = new Square();
    }

    double area(double length) {
        //square = new Square();
        return square.getSquare(length);
    }
}
```

Both Globes are valid

```
class CalculateArea {

    P. S. v. main(String[] args) {

        Square square = new Square();
        Circle circle = new Circle(square);
        S.o.p.ln(circle.area(2.0));

        Rectangle rectangle = new Rectangle();
        S.o.p.ln(rectangle.area(2.0));
    }
}
```

output: 12.56
4

② Composition (Part-of)

- Dependency: child can't exist independently of parent.
- Type of Relation: Part-of
- Association: Strongly associated.

Example-1:

```
class Square {}{
    double getSquare(double value){
        return value * value;
    }
}
```

Private access modifire is mondetory — Composition

```
class Circle {
    Private final Square square; // Composition
    Private Square square;       // Composition
    Circle (Square square) {
        this.square = square;
    }

    double area (double radious) {
        return 3.14 * square.getSquare(radious);
    }
    double area (Square square, double radious) {
        this.square = square;
        return square.getSquare(radious);
    }
}

class Rectangle {
    Private final Square square; // Composition

    Rectangle () {
        Square = new Square();
    }

    double area (double length) {
        return square.getSquare(length);
    }
}
```

OR

```
class Rectangle {
    Private final S.
    Private Square square; // Composition

    double area (double length) {
        Square = new Square();
        return Square.getSquare(length);
    }
}
```

```
class CalculateArea {

    P. S. V. main(String[] args) {

        Square square = new Square();
        Circle circle = new Circle(square);
        S.o.p.ln(circle.area(2));

        Rectangle rectangle = new Rectangle();
        S.o.p.ln(rectangle.area(2));
    }
}
```

Output: 12.56
4

Rules Overloading → Compile time

1) Automatic Promotion

byte → short
char → int → long → float → double

2) String ⇏ Object

```
Public void m(String str){ }
Public void m(Object obj) { }

m("Hiraj") → String
m(new Object()) → Object

m(null) → String
```

3) String & StringBuffer

```
void m(String s){ }
void m(StringBuffer sb) { }
```

obj
Str SB
At Some level

```
m("Hiraj"); → String
m(new StringBuffer("Hiraj") → StringBuffer
m(null) → Compile Time error
```

4)

```
void m(int i, float f)
void m(float f, int i)

m(10, 10.5f) → int, float
m(10.5f, 10) → float, int
m(10, 10) ✗        m(10.5f, 10.5f) ✗
```

5) void m(int i)
void m(int ...i)

m(10) → int i
m(10, 20) → int ...i
m() → int ...i

6)

class Parent { }

class Child extends Parent { }

main() {
    void m(Parent p)
    void m(Child c)
}

void main() {
    Parent p = new Parent();
    m(p) → Parent
    Child c = new Child();
    m(c) → Child
→   Parent p = new Child(); } imp
    m(p) → Parent
→   Child c = (Child)p; } imp
    m(c) → Child
}

In method overloading method resolution always takes decross by compiler based at compile time.

In overloading runtime object won't play any role

Overriding → Run time (Lately Binding)

Rules:

1) Return type:

JDK<1.4 → Some return type must
JDK >=1.5 → Covariant return type support.

Some Return type

```
class Parent {
    void m() {=}
}
class Child extends Parent{
    void m() {=}
}
```

```
class Parent {
    int m() {=}
}
class Child extends Parent{
    int m() {=}
}
```

Covariant Return type

This concept is only applicable for object (Number, Integer, float, Object, String etc) but not for primitive data type (int, float, char).

```
class Parent {
    Object m() {=}
}                    ← Parent class
class Child extends Parent {
    String m() {=}
}
```

Some Parent class
OR
It's child class

```
class Parent {
    Number m() {=}
}
class Child extends Parent{
    Integer m() {=}
}
```

If Parent class method return any object class then child class method should some object class or its child

object class.

2) Private methods are not applicable in overriding.

```
class Parent {
    private m() {=}
}
class child extends Parent {
    private m() {=}
}
```

This is valid

But not overriding method

↑ optional

Private method never override

3) Parent class final method can not override.

```
class Parent {
    final void m() {=}
}
class child extends Parent {
    void m() {=}
}
```

↳ compile time error

4) Abstract class & method

```
abstract class GrandParent {
    abstract void m1();
    abstract void m2();
    abstract void m3();
    void m4() {
        S.o.p.ln("GrandParent - M4");
    }
}
```

```
abstract class Parent extends GrandParent {

    @Override
    abstract void m1();

    @Override
    void m2() {
        S.O.P.ln("Parent-M2");
    }

    @Override
    abstract void m4();
}

class Child extends Parent {

    @Override
    void m1() {
        S.O.P.ln("Child-M1");
    }

    @Override m2() {
        Super.m2();
        S.O.P.ln("Child-M2");
    @Override
    void m2() {
        Super.m2();
        S.O.P.ln("Child-M2");
    }

    @Override
    void m3() {
        S.O.P.ln("Child-M3");
    }
```

```
    @Override
    void m4(){
        S.o.p.ln("Child-M4");
    }
}

Public class Main{

    P.S.V.main(String[] args){

        Child child = new Child();
        child.m1();
        child.m2();
        child.m3();
        child.m4();
    }
}
```

Output: Child-M1
Parent-M2
Child-M3
Child-M3
Child-M4

Parent class abstract method should be override in child class or its child next to child class.

Abstract class implementation is best example of overriding.

Abstract method can override as non-abstract method & Non-abstract method override as abstrat method.

5)

Parent Method		Child Method
final	⟶ X	non final
non final	⟶	final
abstract	⟹ ⟸	non abstract
synchronized	⟹ ⟸	non synchronied
native	⟹ ⟸	non native
strictfp	⟹ ⟸	non Stricfp

6) While overriding scope of access modifier can not reduce where as can increase.

Parent Method: private < default < protected < public

Child Method			
can't override	default	Protected	Public
X	Protected	Public	
	Public		

7) Exception handling or throwing

If parent method checked exception then child method should through same or its child exception.

7) Exception throw (Checked Exception)

If Parent class method throwing any checked Exception then child class method can skip throw exception or it should have to throw some checked exception or its child checked exception.

OR

If child class method throw any checked exception then compulsory Parent class method should throw some checked exception or its Parent checked exception

(1) Parent: void m() throw Exception ✓
Child: void m()

(2) Parent: void m() throw Exception ✓
Child: void m() throw IOException

(3) Parent: void m() throw IOException ✗
Child: void m() throw Exception

(4) Parent: void m() ✗
Child: void m() throw Exception

8) Overriding w.r.t to static methods (Method Hiding)

class Parent {
    static void m() {=}
}
class Child extends Parent {
    void m() {=}
}
Compile time error

class Parent {
    void m() {=}
}
class Child extends Parent {
    static void m() {=}
}
Compile time error

Static ⇏ Non Static

It is method hiding not overriding

```
class Parent {
    static void m() { = }
}
class Child extends Parent {
    static void m() { = }
}
```

```
class Parent {
    static void m() {
        S.o.p.ln("Parent");
    }
}

class Child extends Parent {
    static void m() {
        S.o.p.ln("Child");
    }
}

class Main {
    p. s. v. main(String[] args) {

        Parent parent = new Parent();
        Parent.m();          // Parent

        Child child = new Child();
        child.m();           // Child

        Parent = new Child();
        Parent.m();          // Parent

        Child = (child) Parent;
        child.m();           // Child
```

Method Hiding

Parent. m(); // Parent

Child. m(); // child

}

}

— It is compile time Polymorphism or Static Polymorphism. Also it knows as Early binding

9) Overriding w.r.t to Var-args methods

```
class Parent {
    void m(int ...x) {=}
}
class Child extends Parent {
    void m(int ...x) {=}
}
```

Method Overriding

But,

imp

```
class Parent {
    void m(int ...x) {=}
}
class Child extends Parent {
    void m(int x) {=}
}
```

Method Overloading

One Var-args method can override with another Var-args method.

But

A Var-args method can not override with normal method, it is overloading not overriding.

Overriding Parent class method in Child class with different signatuare is not method overriding where as it is method overloading.

Example

```
class Parent {
    void m(int n) { ≡ }
}
class Child extends Parent {
    void m(String str) { ≡ }
}
```

This is valid
It is method
Overloading

108) Overriding w.r.t to variables

Variable resolution always takes cross by compiler based on reference type irrespect-iN of wether the variable is static or non-static. or final or non-final.

imp { Overriding concept is only applicable for methodes not for variables.

Example

```
class Parent {
    int x = 111;
}
class Child extends Parent {
    int x = 999;
}
class Main {
    P.S. v. main (String[] args) {
```

whatever they both Static or non-static or one static & another non-static & vice versa or they final

PTO

```
Parent parent = new Parent();
parent.x;  // 111

Child child = new Child();
child.x;   // 777

Parent parent = new Child();
parent.x;  // 111

Child child = (Child) parent;
child.x;   // 777
}
}
```

Parent : static
Child : overriding static
Not applicable

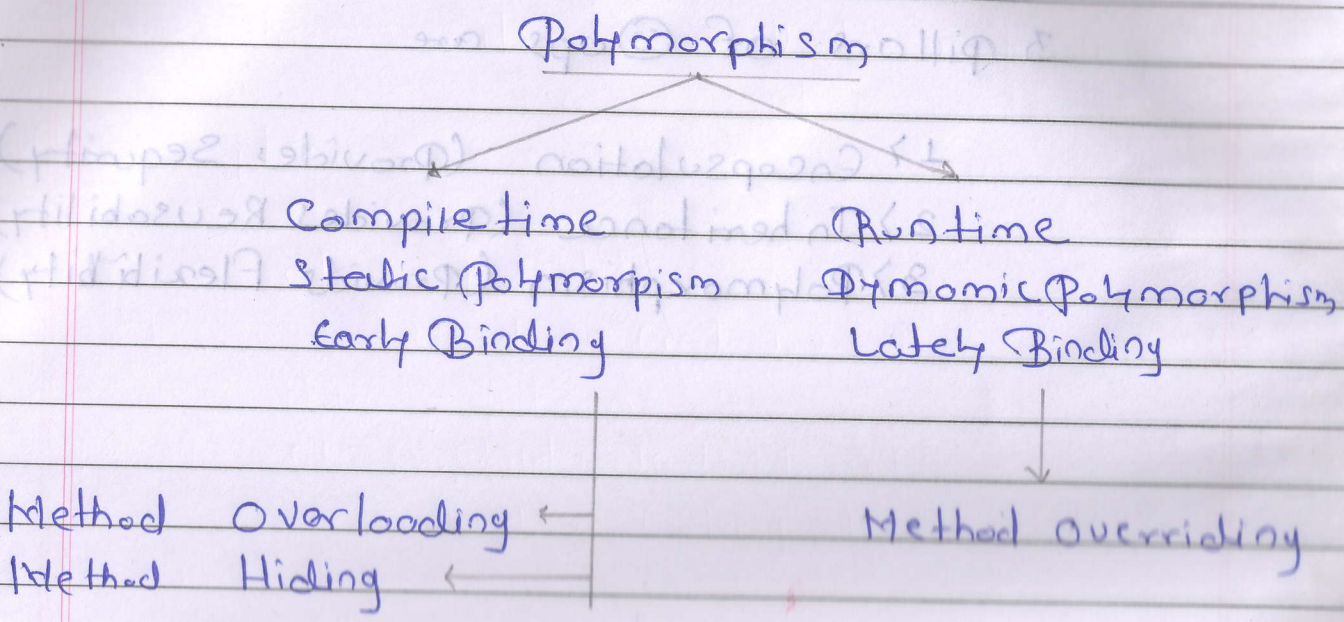
Overriding concept is not applicable for variables.
(int, char, String ... etc)
(Static, non-Static, final, non-final
(Private, non-Private)

111 Static Abstract is invalid combination.

```
Public static abstract void m(int x);
```

Compiletime error : illegal combination of modi

Polymorphism



Beautiful Defination of Polymorphism

The Boy starts Love with the word Friendship. But, the Girl ends Love with the Some word Friendship. Word is some but attitude is different. This beautiful concept of Oops is nothing but Polymorphism.

PAGE No.
DATE / /

Oops

3 pillars of Oops are

1> Encapsulation (Provides Sequrity)
2> Inheritance (Provides Reusobility)
3> Polymorphism (Provids Flexibility)