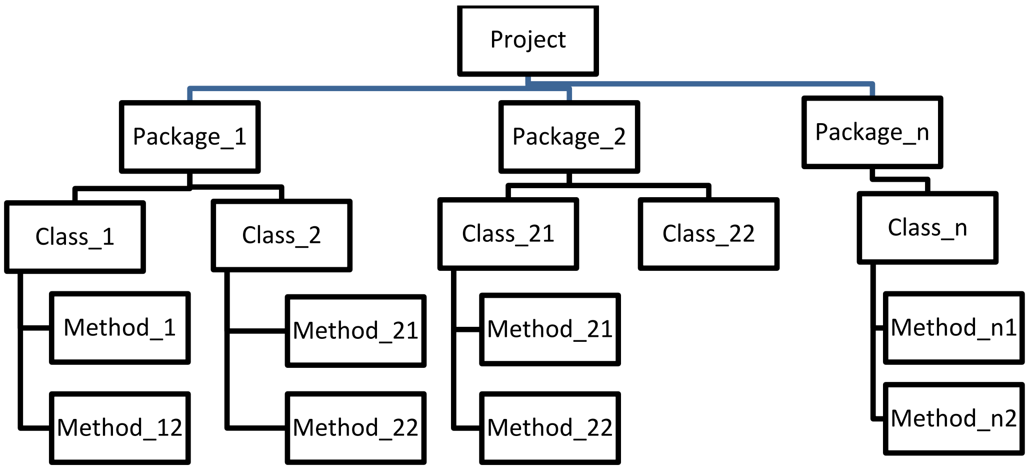
OOPs Concepts

1. Data Hiding
2. Encapsulation
3. Abstraction
4. Is-A Relationship
5. Method Signature
6. Polymorphism
7. Constructors
8. Type Casting

Let us discuss them in detail:

1. Data Hiding:

- Every Java project has following structure.



- Data can reside in either class or methods.
- We can hide data in class or method from other classes and methods of a same project or other project.
- Hiding of data can be on project level, package level or class level. If package level hiding is applied, then no other package in same project or different project can access the data.
- Data hiding is achieved using Access modifiers.
- There are four access modifiers: private, protected, public and default.
  - **Private:**
    i. If we declare any member of a class as private then no one can access them outside of the class in which they are declared and defined.

ii. Private members are accessible with class only.
iii. We can apply private modifiers to variables, methods and constructors only.
iv. Ex. (Refer above diagram) If I declare *method_1*as private then it will only be visible to objects of *class_1*.

- **Public:**
  i. If we declare any member of a class as public then they will become accessible to everyone in project and outside project as well.
  ii. We can apply public modifiers to variables, methods, constructors and classes.
- **Default:**
  i. Default members are accessible inside package only in which they are declared and defined.
  ii. Ex. If we make *method_1* as default, then object of *class_21* cannot access it as *class_21* is in different package. Whereas it will be accessible to object of *class_2*
  iii. We can apply default modifier to variables, methods, constructors and classes
- **Protected:**
  i. Protected members are accessible to child and grandchildren of the current class**.**
  ii. Ex. If we make *method_1* as protected and *class_2* as child of *class_1* then only *class_2* can access *method_1*.
  iii. We can apply protected modifier to variables, methods, constructors.

2. **Abstraction:**
   - Hiding background implementation of the services is abstraction.
   - Let say we want to build an application for e-commerce. Then we cannot disclose our business information to everyone. We would just disclose set of services we may offer but not the procedure of how we will offer the services.
   - Greatest example of abstraction would be a bank ATM machine, which displays different services an ATM can offer. But it doesn't allow you to view the background implementation of the services.
   - By using abstraction we can maintain security and confidentiality among our business.
   - We can achieve abstraction by means of Abstract classes and Interfaces.

Sr. No	Abstract Class	Interface
1	Abstract class is used to achieve 0 to 100% abstraction	Interface is used to achieve 100% abstraction
2	Every method of abstract class should be public and abstract. But concrete methods are also allowed	Compulsorily every method in interface should be public and abstract
3	We can use any modifiers for the abstract class methods	We can use only private, protected, static, final and synchronized modifiers for the interface methods.
4	Abstract class variables need not be public ,static and final	Every variable inside interface are by default public, static and final.
5	It is not compulsory to initialize abstract class variables	We should compulsorily initialize variables in interface
6	Inside abstract class we can define instance and static blocks	Inside interface we cannot define instance and static blocks
7	Inside abstract class we can take constructor	Inside interface we cannot define constructor

3. **Encapsulation:**
   - Binding data and methods into a single entity is called as Encapsulation.
   - A Class can be a example of encapsulation.
   - An encapsulated class may uses Data Hiding and Abstraction both.
   - Using encapsulation we can achieve security and modularity.
   - Encapsulation allows organizing data and methods.
4. **Is-A Relationship (Inheritance):**
   - This is also known as inheritance.
   - Using Is-A Relationship we can transfer data and methods of parent class to child class but reverse is not possible.
   - We can achieve inheritance using ‘extends’ keyword.
   - Ex.

```
public class Parent {
	char ch = 'A';

	public void display() {

	}
}
public class Child extends Parent{
   public static void main(String[] args) {
      Child c=new Child();
      c.display(); //method of Parent class.
   }
 }
```

Types of Inheritance in Java:

There are four types of inheritance.

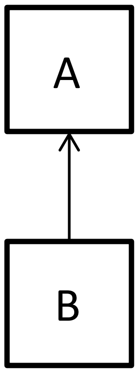
I. Single Inheritance
II. Multi-level inheritance
III. Multiple inheritance

IV. Hierarchical Inheritance
V. Hybrid Inheritance

Let us discuss them in detail.

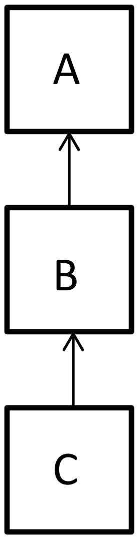
I. Single Inheritance:

- When there is only one parent and one child, then it is called as single inheritance.
- Ex. `public class B extends A.`
- Here A is parent and B is child
- All methods and variables (instance and static) of class A will be accessible to the object of Class B except those are private.



II. Multi-level Inheritance:

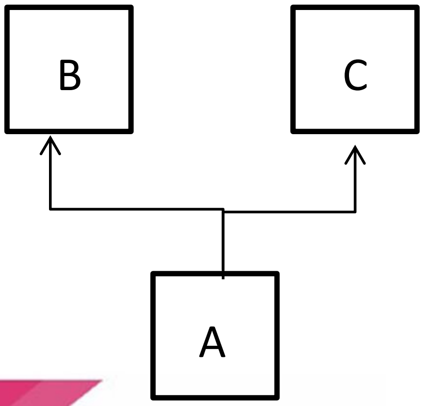
- When there is Grand Parent, Parent, Child and Grand Child hierarchy, it is called as multi-level inheritance.
- Ex. `public class B extends A`
  `public class C extends B`



- In above example A is grand parent of C and parent of B.
- Class C object will have access to all features of Class A, Class B and its own.
- Class B object will have access to all features of Class A and its own.
- Class A object will have access to its own features only as parent cannot extend properties of child.

III. Multiple Inheritance:

- Multiple inheritance means, single child multiple parents.
- Multiple Inheritance is not allowed in Java.
- Still we can achieve Multiple Inheritance using Interfaces.
- Ex. `public class A extends B, C //Not allowed`
  `Public interface I1 implements I2, I3, I4 //Allowed`

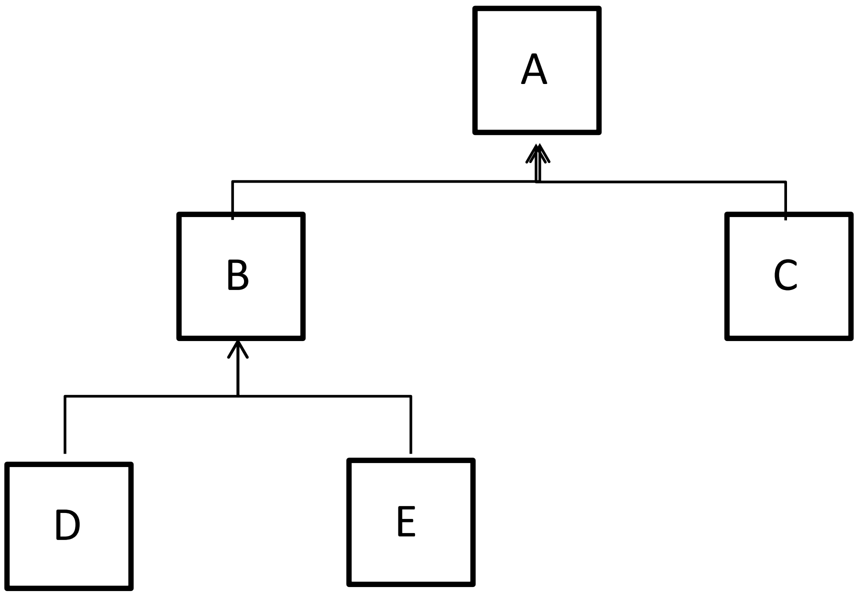


IV. **Hierarchical Inheritance:**

- Hierarchical inheritance creates tree structure of Parent child relationship.
- There is only one root parent and multiple leaf children possible.
- Ex. `public class A`

```
public class B extends A
public class C extends A
public class D extends B
public class E extends B
```

- Above statements are represented using diagram given below.



5. **Method Signature:**
   - Method Signature consists of method name and list of parameters.
   - Ex. m1(int a, int b);
   - Return type is not a part of signature.
   - Compiler uses method signature to resolve method call.
   - In same class two methods with same signature is not allowed.
6. **Polymorphism:**
   - Polymorphism- '*Poly*' means many, '*morphism*' means forms.
   - We can achieve polymorphism using method overloading and overriding.

- Method overloading is compile time or static.
- Method overriding is runtime or dynamic.

I. Method Overloading:

- Overloaded methods should be in same class
- Overloaded methods have same name but different list of arguments.
- Ex.

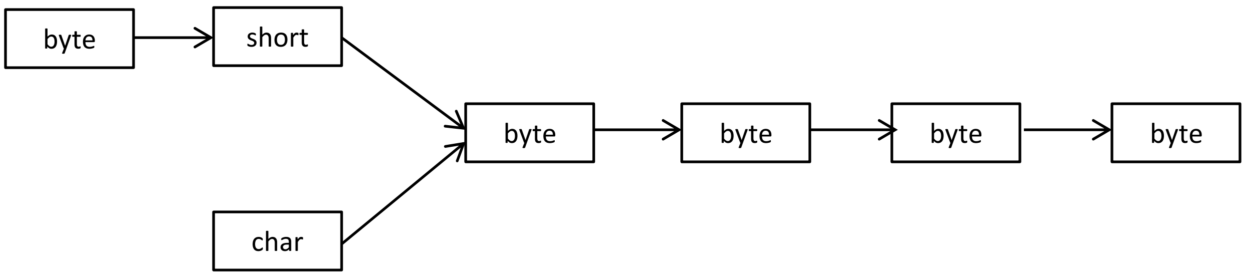
```
public class Demo{
      public void add(int a, int b){

      }
      Public void add(char a, char b){

      }
```

- In above example 'add()' method is overloaded method.

- **Type Promotion in Method Overloading:**
  - In method overloading, if methods with specified list of arguments is not found, then compiler will not through any error immediately. Instead it will promote the argument(Data Type of argument) to next level and checks for the match
  - If list of parameter matches then the method will be executed else it will again promote the data type.
  - This process will continue until all possible promotions are checked.
  - If no matching method found, Compile Time Exception will be thrown.
  - This is called automatic type promotion.
  - Allowed Type promotions.



  - Below program illustrates the same.

```
public class Test {
	public void add(int a, int b) {
		System.out.println("Add itegers");

	}
	public void add(float a, float b) {
		System.out.println("Add floats");

	}
	private void add(short a, short b) {
		System.out.println("Add shorts");

	}
	public static void main(String[] args) {
		Test test=new Test();
		byte a=5;
		byte b=10;
		test.add(a, b);
	}
}
```

II. Method Overriding:

- Overridden methods must be in different class.
- Classes in which overridden methods are declared should possess parent-child relationship compulsorily.
- Name and list of parameters of overridden methods should be same.
- In overriding, return type also should be same. This rule is valid till Java V1.4. Since Java 1.5 we can mention different return type for overridden methods provided return type of child method should be child of Parent method's return type.
- Ex.

```
public class Parent {
   public Object add() {
        return null;
   }
}
```

```
public class Child {
	public String add() {
		return null;
	}
```

- In above example return type of add() method of Child class is String which is child of Object. Hence we can write return type of add method of Parent class as Object. But reverse is not possible.
- This is called Co-variant return types.
- Only objects are allowed as Co-variant return types but not primitive data types.
- We cannot override final method
- We can mention static modifier to overridden methods but it is not overriding. It is called as ‘Method Hiding’.
- We can mention private access modifier to overridden methods, but it will not be considered as overriding. Those methods will be treated differently at run-time.

Property	**Overloading**	**Overriding**
Method name	Must be Same	Must be same
List of arguments	Must be different (at least order)	Must be same
Method signature	No restriction	Must be same until Java1.4. But since 1.5 Co-variant return types are allowed
Private, static and final methods	Can be overloaded	Can’t be overridden.
Access modifiers	No restriction	We can’t decrease the scope. • Parent-Public and Child-Default not allowed. • Parent-Default and Child-Public is allowed
Level of polymorphism	Compile time polymorphism	Runtime Polymorphism.
Checks	We should only check method name(must be same) and list of arguments (must be different)	All should be checked, like return type, access modifier, name, list of arguments.

7. Constructor:

- Constructor is a special type of method which is used to initialize an object.
- Java allows objects to initialize themselves when they are created.
- This automatic initialization is performed through the use of a constructor.
- Constructor is a special type of method which doesn't have return type. Explicitly its return type is 'Class'.
- If we by mistake provide return type to constructor then compiler will not give any error because it will treat it as a method.
- Constructor's name is exactly same as the class name in which it resides.
- Allowed Modifiers: public, private, protected, default.
- Final static not allowed.
- First line inside constructor is always super( ). It calls super class's default constructor.
- Two types of constructors: Default and Parameterized.

- **Default Constructor:**
    - If we are not providing any constructor, then compiler will automatically generate default constructor.
    - If we are writing at least one constructor then compiler will not provide default constructor.
    - Class can contain programmer defined constructor or a parameterized constructor but not both.

Programmers code	**Compiler's code**

class Test{
}
	class Test{
	    Test(){
	        super();
	    }
	}

public class Test{
}
	public class Test{
	    public Test(){
	        super();
	    }
	
	}

class Test{
    int Test(){
    }// Not a constructor
 	
}
	class Test{
	    Test(){
	        super()
	    }
	    int Test(){
	    }

- **Super and this keywords:**
    - Using 'this' keyword we can access anything of current class
    - Using 'super' keyword we can access anything of Parent class.
    - We can use these keywords anywhere except in static methods or blocks.
    - We can write more than one constructor with same name in one class. It is called as 'Constructor Overloading'.
    - We cannot override constructors.

8. Type-Casting:

- If we assign int value to a long variable, then Java will automatically convert int to long as int child and long is parent. But reverse will not be possible
- Parent class reference can be used to hold child class's object.
- Ex. Parent p=new Child();
- Similarly interface reference can be used to hold implementation class's object.
- But if there are certain situations where we need to convert parent into child. Ex. If we want to assign long value to an int variable, then we can do this by **Type Casting.**
- **Syntax:** (*target-type*)*value;*
- Type-Casting allows parent to convert into child.
- Ex.

```
    int i=10;
    long l=i;   //This is possible
     long l=10;
     int i=l;     //This is not possible. Possible only using
Type-Casting
     int i=(int)l;   //Using Type-Casting
```

- Similarly we can convert float into int using type casting but there is chance of loss of fractional value.
- Ex.

```
  float f=10.23f;
 int i=(int)f; //Here I will contain 10. .23 is truncated
```