Inheritance, polymorphism, interfaces and abstract

- Inheritance, polymorphism, interfaces and abstract
  - Inheritance
    - Types of inheritance
      - Diamond problem
  - What is Polymorphism?
    - Subtyping
    - Method Overloading
    - Method Overriding
    - Advantages of Polymorphism
    - Problems with Polymorphism
  - Interfaces
    - Why use an interface?
    - How to create an interface?
  - Abstract Classes
    - Why use an abstract class?
    - How to create an abstract class?
  - Reading List

Inheritance

Inheritance is the mechanism that allows one class to acquire all the properties from another class by inheriting the class. We call the inheriting class a child class and the inherited class as the superclass or parent class.

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

Imagine, as a car manufacturer, you offer multiple car models to your customers. Even though different car models might offer different features like a sunroof or bulletproof windows, they would all include common components and features, like engine and wheels.

It makes sense to create a basic design and extend it to create their specialized versions, rather than designing each car model separately, from scratch.

Similarly, with inheritance, we can create a class with basic features and behavior and create its specialized versions, by creating classes, that inherit this base class. In the same way, interfaces can extend existing interfaces.

Let us create a new class `User` which should be the parent class of `Student`:

```java
public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }
}
```

Now, let us create a new class `Student` which should be the child class of `User`. Let us add some methods specific to the student class:

```
public class Student {
    private String batchName;
    private Integer psp;

    ...
}
```

Now in order to inherit the methods and fields of the parent class, we need to use the keyword `extends`:

```
public class Student extends User {
    private String batchName;
    private Integer psp;

    ...
}
```

To pass the values to the parent class, we need to create a constructor and use the keyword `super`:

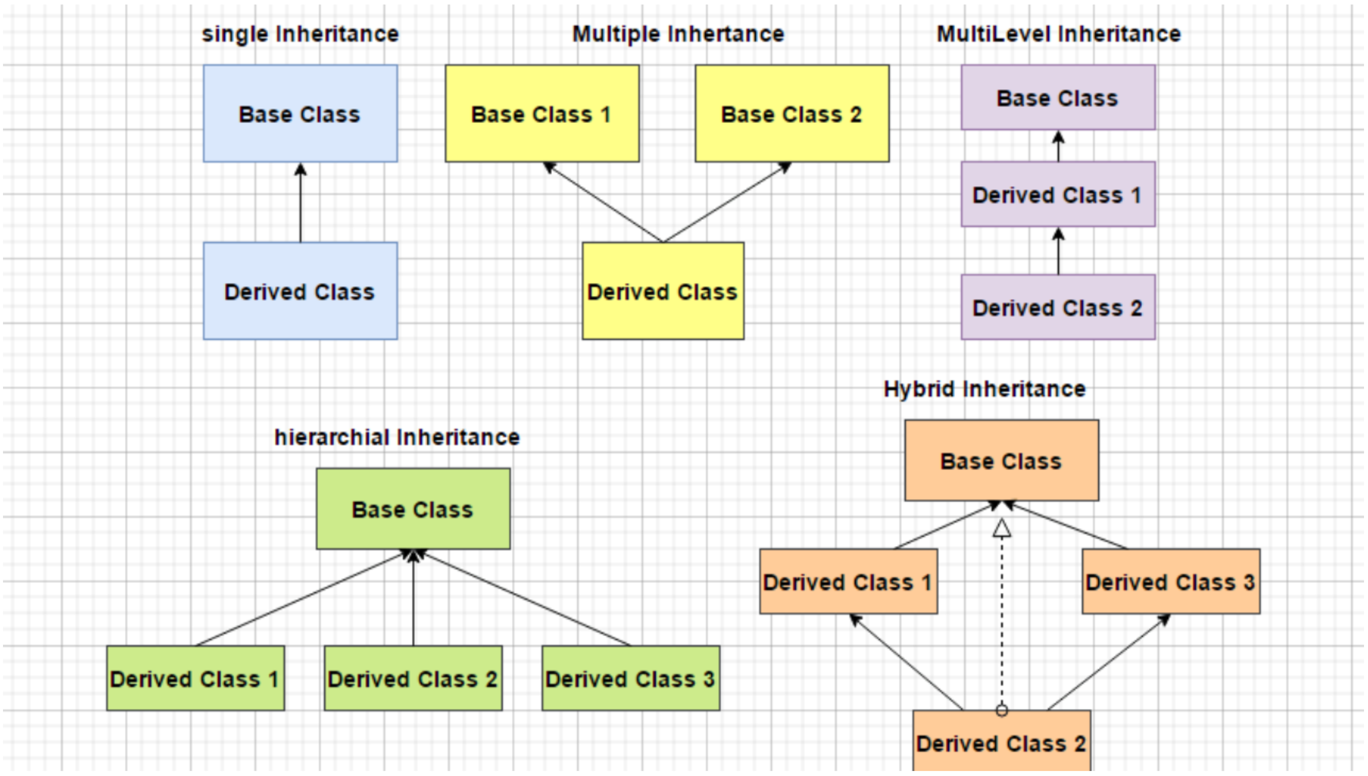
```
public class Student extends User {
    private String batchName;
    private Integer psp;

    public Student(String name, String email, String batchName, Integer
psp) {
        super(name, email);
        this.batchName = batchName;
        this.psp = psp;
    }
}
```

Types of inheritance

There are four types of inheritance:

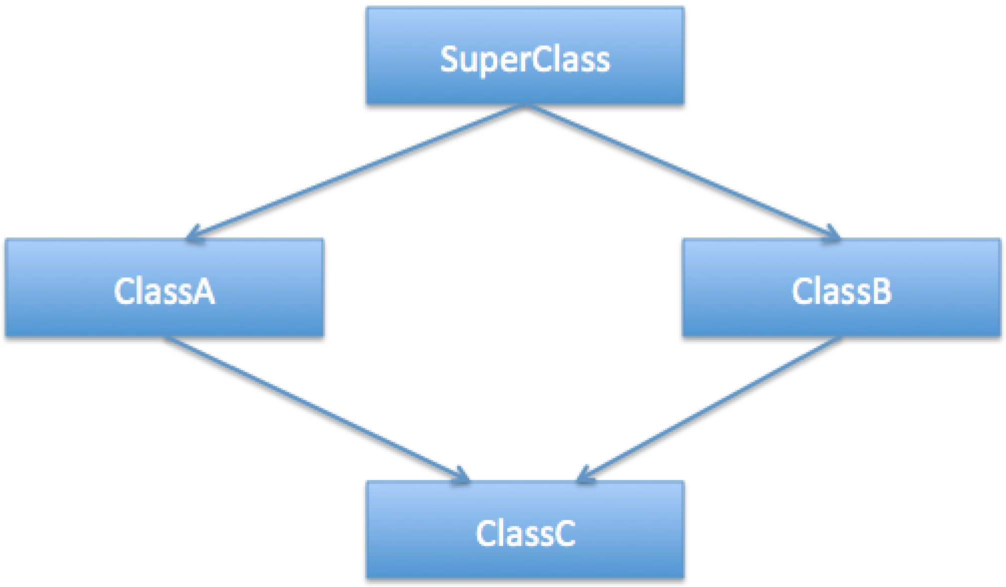
- `Single` - A single inheritance is when a class can have only one parent class.

- `Multilevel` - A multilevel inheritance is when a class can have multiple parent classes at different levels.
- `Hierarchical` - When two or more classes inherits a single class, it is known as hierarchical inheritance.
- `Multiple` - When a class can have multiple parent classes, it is known as multiple inheritance.



Diamond problem

In multiple inheritance one class inherits the properties of multiple classes. In other words, in multiple inheritance we can have one child class and n number of parent classes. Java does not support multiple inheritance (with classes).



The "diamond problem" (sometimes referred to as the "Deadly Diamond of Death") is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C.

What is Polymorphism?

Polymorphism is one of the main aspects of Object-Oriented Programming(OOP). The word polymorphism can be broken down into “Poly” and “morphs”, as “Poly” means many and “Morphs” means forms. In simple words, we can say that ability of a message to be represented in many forms.

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this polymorphism occurs, the object's declared type is no longer identical to its run-time type
- Base classes may define methods, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. In your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Polymorphism in Java can be achieved in two ways i.e., method overloading and method overriding.

Polymorphism in Java is mainly divided into two types.

- Compile-time polymorphism
- Runtime polymorphism

Compile-time polymorphism can be achieved by method overloading, and Runtime polymorphism can be achieved by method overriding.

Subtyping

Subtyping is a concept in object-oriented programming that allows a variable of a base class to reference a derived class object. This is called polymorphism, because the variable can take on many forms. The variable can be used to call methods that are defined in the base class, but the actual implementation of the method is defined in the derived class.

For example, the following is our User class:

```java
public class User {
    private String name;
    private String email;
}
```

The user class is extended by the Student class:

```
public class Student extends User {
    private String batchName;
    private Integer psp;
}
```

The Student class inherits the name and email properties from the User class. The Student class also has its own properties batchName and psp. The Student class can be used in place of the User class, because the Student class is a subtype of the User class. The following is an example of how this works:

```
User user = new Student();
```

Method Overloading

Method overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

Let's take an example of a class that has two methods having the same name but different in parameters.

```
public class User {
    private String name;
    private String email;

    public void printUser() {
        System.out.println("Name: " + name + ", Email: " + email);
    }

    public void printUser(String name, String email) {
        System.out.println("Name: " + name + ", Email: " + email);
    }
}
```

In the above example, the class has two methods with the same name printUser but different in parameters. The first method has no parameters, and the second method has two parameters. This is called method overloading.

The compiler distinguishes these two methods by the number of parameters in the list and their data types. The return type of the method does not matter.

Method Overriding

Runtime polymorphism is also called Dynamic method dispatch. Instead of resolving the overridden method at compile-time, it is resolved at runtime.

Here, an overridden child class method is called through its parent's reference. Then the method is evoked according to the type of object. In runtime, JVM figures out the object type and the method belonging to that object.

Runtime polymorphism in Java occurs when we have two or more classes, and all are interrelated through inheritance. To achieve runtime polymorphism, we must build an "IS-A" relationship between classes and override a method.

If a child class has a method as its parent class, it is called method overriding.

If the derived class has a specific implementation of the method that has been declared in its parent class is known as method overriding.

Rules for overriding a method in Java

- There must be inheritance between classes.
- The method between the classes must be the same(name of the class, number, and type of arguments must be the same).

Let's add a method to our User class:

```
public class User {
    private String name;
    private String email;

    public void printUser() {
        System.out.println("Name: " + name + ", Email: " + email);
    }
}
```

Now, let's add a method to our Student class:

```
public class Student extends User {
    private String batchName;
    private Integer psp;

    @Override
    public void printUser() {
        System.out.println("Name: " + name + ", Email: " + email + ",
Batch: " + batchName + ", PSP: " + psp);
    }
}
```

In the above example, we have added a method to the Student class that overrides the method in the User class. The Student class has a method with the same name and parameters as the User class. The Student class method has an additional print statement that prints the batchName and psp properties.

The @Override annotation is optional, but it is a good practice to use it. It is used to ensure that the method is actually being overridden. If the method is not being overridden, the compiler will throw an error.

Advantages of Polymorphism

- Code reusability is the main advantage of polymorphism; once a class is defined, it can be used multiple times to create an object.
- In compile-time polymorphism, the readability of code increases, as nearly similar functions can have the same name, so it becomes easy to understand the functions.
- The same method can be created in the child class as in the parent class in runtime polymorphism.
- Easy to debug the code. You might have intermediate results stored in arbitrary memory locations while executing code, which might get misused by other parts of the program. Polymorphism adds necessary structure and regularity to computation, so it is easier to debug.

Problems with Polymorphism

- Implementing code is complex because understanding the hierarchy of classes and its overridden method is quite difficult.
- Problems during downcasting because implicitly downcasting is not possible. Casting to a child type or casting a common type to an individual type is known as downcasting.
- Sometimes, when the parent class design is not built correctly, subclasses of a superclass use superclass in unexpected ways. This leads to broken code.
- Runtime polymorphism can lead to the real-time performance issue (during the process), it basically degrades the performances as decisions are taken at run time because, machine needs to decide which method or variable to invoke

Interfaces

An interface is a reference type in Java. It is similar to a class, but it cannot be instantiated. It can contain only constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

It can be thought of as a blueprint of behavior. It is used to achieve abstraction and multiple inheritance in Java.

Why use an interface?

- It is used to achieve abstraction.
- Due to multiple inheritance, it can achieve loose coupling.
- Define a common behavior for unrelated classes.

How to create an interface?

Let us create an interface for a Person

```java
public interface Person {
    String getName();
    String getEmail();
}
```

Now let's create a class that implements the Person interface:

```java
public class User implements Person {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getEmail() {
        return email;
    }
}
```

Abstract Classes

An abstract class is a class that is declared abstract. It may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Why use an abstract class?

- It is used to achieve abstraction.
- It can have abstract methods and non-abstract methods.
- When you don't want to provide the implementation of a method, you can make it abstract.
- When you don't want to allow the instantiation of a class, you can make it abstract.

How to create an abstract class?

Let us create an abstract class for a Person You can create an abstract class by using the abstract keyword. Similarly, you can create an abstract method by using the abstract keyword.

```java
public abstract Person {

    public abstract String getName();
    public abstract String getEmail();
}
```

Now let's create a class that extends the Person abstract class:

```
public class User extends Person {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getEmail() {
        return email;
    }
}
```

Reading List

- Deadly Diamond of Death
- Detailed explanation of the diamond problem
- Duck Typing
- OOP in Python