

به نام خدا



برنامه‌سازی پیشرفته

دانشگاه شهید بهشتی · دانشکده مهندسی و علوم کامپیوتر

دکتر مجتبی وحیدی اصل

کلاس‌های انتزاعی و رابط‌ها

سید محمد حسینی

فهرست مطالب

- 1. انتزاع (Abstraction)
- 2. کلاس‌های انتزاعی
- 3. واسط (Interface)
- 4. حذف سطوح دسترسی در Interface
- 5. رابطه بین کلاس‌ها و Interface‌ها
- 6. شباهت و تفاوت‌های Class, Interface
- 7. ارث بری چندگانه در جاوا به کمک Interface
- 8. مقایسه Interface, Abstract Class
- 9. واسط Iterator
- 10. واسط Clonable

انتزاع (Abstraction)

یکی از مزایای **شیءگرایی انتزاع** است. حالا ممکنه برآتون سوال پیش بیاد که **انتزاع** چیه اصلا؟
انتزاع یعنی برنامه ما **لایه‌های** متفاوتی دارد و ما می‌خواهیم یکسری **لایه‌های** مشخص برای **کاربر** قابل مشاهده باشد و سایر **لایه‌ها** صرفا برای **برنامه‌نویس** قابل مشاهده باشد.
این اقدام برای **کارکردها** برای **کاربر** قابل مشاهده باشد و **جزئیات** این موارد را صرفا **برنامه‌نویس** با **کدهای** خود مدیریت کند.



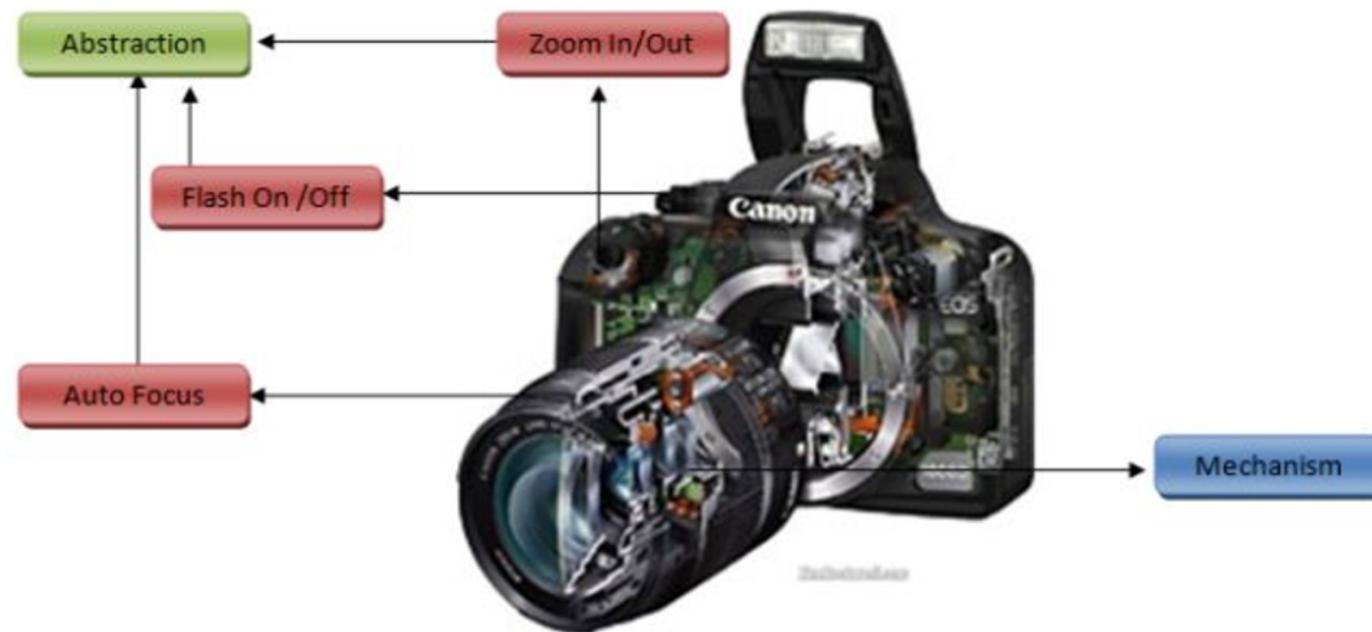
به عنوان مثال برای گوشی موبایل، شما صرفا به دوربین و عکس‌گرفتن دسترسی دارید اما نحوه ذخیره این عکس در گالری گوشی شما و نحوه عکس گرفتن از مواردی است که صرفا برنامه‌نویس به آن دسترسی دارد و آنرا کنترل می‌کند.

حالا این انتزاع را به وسیله چه مواردی می‌توانیم پیاده‌سازی کنیم؟

- کلاس‌های انتزاعی (Abstract classes)
 - رابطه‌ها (Interfaces)
-

انتزاع (Abstraction)

یک دوربین دیجیتالی را در نظر بگیرید



حال اگر بخواهید از دوربین استفاده کنید، صرفا یکسری دکمه‌هایی برای zoom in, zoom out و گرفتن عکس وجود دارد که می‌توانید از این موارد برای عکس‌گرفتن استفاده کنید. حال اگر شما دوربین را باز کنید، متوجه خواهید شد برای هر کدام از دکمه‌ها مکانیزم‌های پیچیده‌ای وجود دارد که فهمیدن آنها کار راحتی نخواهد بود. به علاوه، شما به عنوان کاربر اصلا نیازی به دانستن این موارد ندارید. شما صرفا می‌خواهید از منظرهای عکس بگیرید و این عکس ذخیره شود، حال به هر شکلی که ممکن است. طریقه ذخیره‌سازی عکس‌ها و نحوه پیاده‌سازی ویژگی‌های دیگر، از مواردی است که سازنده آن دوربین باید بداند و برای سایرین ضروری نخواهد بود.

با استفاده از این خاصیت (**Abstraction**) تنها موارد مهم و امکانات یک کلاس به کاربر نشان داده می‌شود و جزئیات داخلی مثل نحوه پیاده‌سازی از کاربر پنهان می‌شود.

انتزاع سبب می‌شود کاربر بر روی قابلیت‌های یک شی به جای نحوه پیاده‌سازی این قابلیت‌ها متمرکز شود.

کلاس‌های انتزاعی

کلاسی که با کلمه کلیدی **abstract** بیان شده باشد، کلاس **انتزاعی** نامیده می‌شود.

```
abstract class A {}
```

متدهی که با کلمه کلیدی **abstract** اعلان شده و بدنه آن خالی است (پیاده‌سازی ندارد) **متدهای abstract** نامیده می‌شود.

```
abstract void printStatus(); // no body and abstract
```

مثالی از کلاس‌های انتزاعی

در این مثال کلاس **Bike** یک کلاس **abstract** است که حاوی یک **متدهای abstract** نیز می‌باشد. از کجا متوجه شدیم این متدهای **abstract** است؟ پیاده‌سازی **متدهای run** داخل کلاس **Ford** انجام شده است. جاییکه ما کلاس **Honda4** را پیاده‌سازی کرده‌ایم.

```
abstract class Bike{
    abstract void run();
}

class Honda4 extends Bike{
    void run(){System.out.println("running safely...");}
}

public static void main(String args[]){
    Bike obj = new Honda4();
    obj.run();
}
```

running safely...

مثالی از کلاس‌های انتزاعی

این یک مثال از **مفهوم انتزاع** در جاوا است.

یک کلاس **abstract** به نام Shape تعریف شده که شامل یک متدهای **abstract** به نام draw می‌باشد.

دو کلاس Circle و Rectangle از این کلاس ارث بری کرده و متدهای draw را پیاده‌سازی می‌کنند.

در متدهای main، یک شیء از نوع Circle ایجاد شده و متدهای draw آن فراخوانی می‌شود که خروجی "drawing circle" را تولید می‌کند.

```
abstract class Bank{
    abstract int getRateOfInterest();
}

class SBI extends Bank{
    int getRateOfInterest(){return 7;}
}

class PNB extends Bank{
    int getRateOfInterest(){return 6;}
}

class TestBank{
    public static void main(String args[]){
        Bank b=new SBI(); //If object is PNB, method of PNB will be invoked
        int interest=b.getRateOfInterest();
        System.out.println("Rate of Interest is: "+interest+" %");
    }
}
```

Rate of Interest is: 7 %

مثالی از کلاس‌های انتزاعی

این کد یک مثال عملی از مفهوم شی‌عگرایی و انتزاع (Abstraction) در زبان جاوا را نمایش می‌دهد. در این مثال، سیستم ساده‌ای از بانک‌ها با نرخ‌های سود مختلف شبیه‌سازی شده است.

کلاس اصلی به نام **Bank** تعریف شده که یک کلاس انتزاعی (abstract) است. این کلاس نقش یک قالب یا الگو را برای تمام بانک‌ها بازی می‌کند. در داخل این کلاس، یک متدهای انتزاعی به نام **getRateOfInterest** وجود دارد که هیچ پیاده‌سازی خاصی ندارد و فقط مشخص می‌کند که هر بانک باید بتواند نرخ سود خود را برگرداند.

دو کلاس **PNB** و **SBI** از کلاس **Bank** ارث‌بری می‌کنند. هر کدام از این کلاس‌ها متدهای **getRateOfInterest** را به صورت جداگانه پیاده‌سازی می‌کنند:

- کلاس **SBI** نرخ سود 7 درصد را برمی‌گرداند
- کلاس **PNB** نرخ سود 6 درصد را برمی‌گرداند

در کلاس **TestBank** که شامل متدهای **main** است، برنامه اصلی اجرا می‌شود. در این قسمت:

یک شیء از نوع **SBI** ایجاد می‌شود اما در متغیری از نوع **Bank** ذخیره می‌شود. این کار نشان‌دهنده یکی از مهم‌ترین مفاهیم شی‌عگرایی به نام "چندریختی" (Polymorphism) است. سپس متدهای **getRateOfInterest** فراخوانی شده و با توجه به اینکه شیء واقعی از نوع **SBI** است، متدهای مربوط به کلاس **SBI** اجرا می‌شود.

"Rate of Interest is: 7%"

مهم‌ترین نکته این کد این است که اگر ما بخواهیم شیء را از **PNB** به جای **SBI** ایجاد کنیم، بدون هیچ تغییری در کد اصلی، نرخ سود مربوط به **PNB** نمایش داده می‌شود. این نشان‌دهنده انعطاف‌پذیری و قدرت استفاده از مفاهیم انتزاع و چندریختی در برنامه‌نویسی شی‌عگرایی است.

متد abstract در کلاس abstract

از یک کلاس **abstract** با استفاده از کلیدواژه **new** نمی‌توان شی جدیدی ایجاد کرد اما همچنان می‌توانیم از **سازنده** این کلاس‌های انتزاعی در کلاس‌های فرزندشان استفاده کنیم.

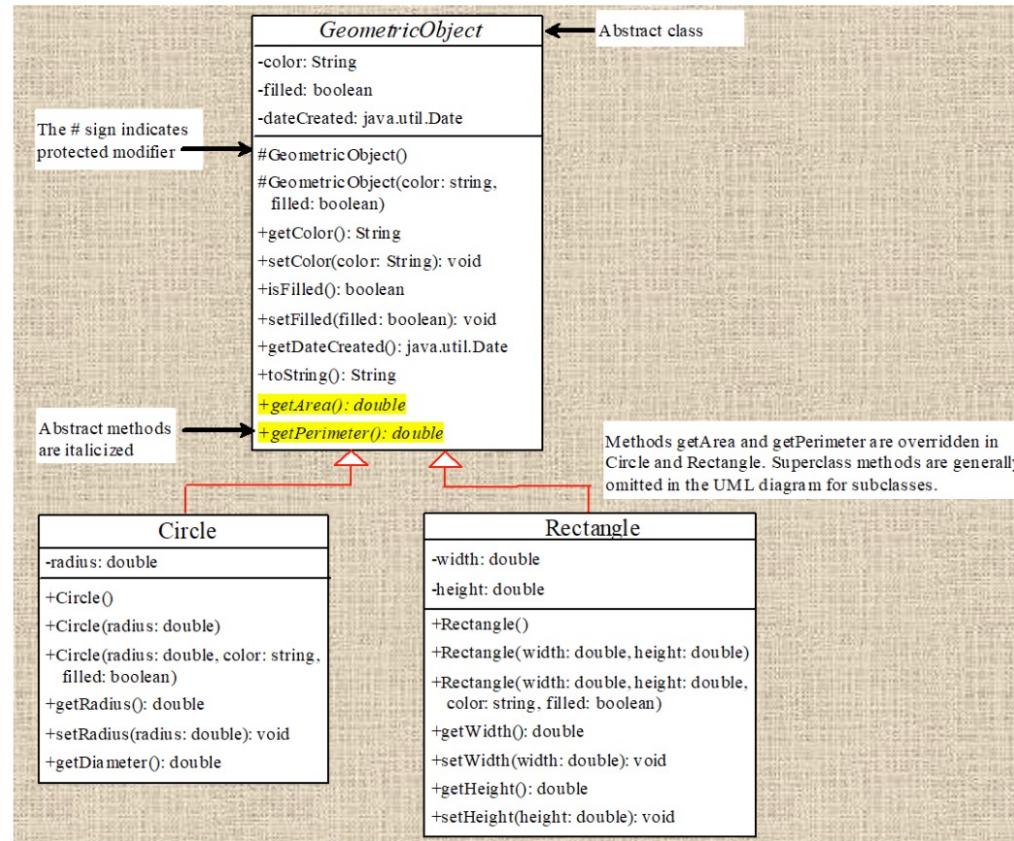
این کد یک ساختار شی‌عگرا را نشان می‌دهد که از مفاهیم کلاس انتزاعی و وراثت استفاده می‌کند.

- یک کلاس انتزاعی به نام **Animal** تعریف شده است.
- این کلاس دارای یک متغیر نمونه به نام **name** و یک سازنده است.
- کلاس شامل یک متد انتزاعی **makeSound** بدون پیاده‌سازی می‌باشد.
- یک کلاس معمولی به نام **Dog** از کلاس **Animal** ارث‌بری می‌کند.
- کلاس **Dog** سازنده‌ای دارد که سازنده کلاس والد را فراخوانی می‌کند.
- این کلاس متد **makeSound** را پیاده‌سازی کرده و بدنه آن را ارائه می‌دهد.
- متد **makeSound** نام حیوان و صدای مربوطه را در خروجی چاپ می‌کند.
- اگرچه نمی‌توان از کلاس **Animal** مستقیماً شیء ایجاد کرد.
- اما سازنده آن در کلاس‌های فرزند قابل استفاده است.
- این ویژگی امکان قرار دادن کدهای مشترک در سازنده کلاس والد را فراهم می‌کند.

```
abstract class Animal {  
    String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    abstract void makeSound();  
}  
  
class Dog extends Animal {  
    public Dog(String name) {  
        super(name);  
    }  
  
    void makeSound() {  
        System.out.println(name + " says: Woof!");  
    }  
}
```

متدهای abstract در کلاس abstract

به تصویر زیر دقت کنید.



این یک دیاگرام UML برای یک سلسله مراتب کلاس در برنامه‌نویسی شیءگرا را نمایش می‌دهد.

- کلاس **GeometricObject** به عنوان یک کلاس انتزاعی تعریف شده است.
 - این کلاس دارای سه ویژگی اصلی می‌باشد: color از نوع String, filled boolean و dateCreated از نوع Date.
 - کلاس دارای دو سازنده است: یک سازنده پیش‌فرض و یک سازنده با پارامتر.
 - متدهای getPerimeter و getArea به صورت انتزاعی تعریف شده‌اند و پیاده‌سازی ندارند.
 - سایر متدها برای دسترسی و تغییر مقادیر ویژگی‌ها پیاده‌سازی شده‌اند.
 - دو کلاس **Rectangle** و **Circle** از کلاس GeometricObject ارث‌بری می‌کنند.
 - کلاس Circle دارای ویژگی radius و متدهای مربوط به مدیریت شعاع و محاسبه قطر می‌باشد.
 - کلاس Rectangle دارای دو ویژگی width و height و متدهای مربوط به مدیریت ابعاد می‌باشد.
 - هر دو کلاس دارای چندین سازنده با پارامترهای مختلف هستند.
 - متدهای getArea و getPerimeter در هر دو کلاس Circle و Rectangle بازنویسی شده‌اند.
 - این بازنویسی برای ارائه پیاده‌سازی خاص هر شکل هندسی انجام شده است.
 - این ساختار امکان استفاده از چندrijختی را فراهم می‌کند.
 - در دیاگرام UML، متدهای انتزاعی به صورت ایتالیک نمایش داده می‌شوند.
 - نماد # برای مشخص کردن اعضای protected و + برای اعضای public استفاده شده است.
 - معمولاً متدهای سوپرکلاس در دیاگرام زیرکلاس‌ها حذف می‌شوند.
-

نکته مهم درباره کلاس‌های انتزاعی

متد **abstract** نمی‌تواند درون یک **کلاس غیر-abstract** قرار گیرد. این به این معنی است که اگر کلاسی قرار است نمونه‌سازی شود و به صورت کامل رفتار مشخصی داشته باشد، نباید شامل متدهای باشد که هنوز **بدون پیاده‌سازی** است. قرار دادن متد **abstract** در کلاس عادی باعث می‌شود آن کلاس ناقص باشد و نتوان از آن به طور مستقیم شیء ساخت.

اگر یک **زیرکلاس** از یک **ابرکلاس abstract**، **تمامی متدهای abstract** تعریف شده در ابرکلاس را **پیاده‌سازی** نکند، خود این زیرکلاس هم باید به عنوان **abstract** تعریف شود. این قاعده تضمین می‌کند که هر کلاسی که قرار است **قابل نمونه‌سازی** باشد، همه رفتارهای لازم را داشته باشد. در غیر این صورت، کلاس همچنان ناقص است و نباید بتوان از آن شیء ساخت.

به بیان دیگر، وقتی یک **زیرکلاس غیر-abstract** از یک **کلاس abstract** ارث بری می‌کند، موظف است **تمامی متدهای abstract** را **پیاده‌سازی** کند، حتی اگر در آن زیرکلاس مستقیماً از همه آن متدها استفاده نشود. این الزام باعث می‌شود که هر کلاس غیر-abstract **قابلیت نمونه‌سازی کامل** داشته باشد و هیچ متد تعریف‌نشده‌ای باقی نماند.

کلاسی با متدهای abstract باید به صورت **abstract** تعریف شود.

اما می‌توانیم یک **کلاس abstract** را بدون **متدهای abstract** تعریف کنیم.

قادر نیستید با **new** از یک **کلاس abstract** نمونه (شیء) ایجاد کنید.

از این **کلاس abstract** می‌توان به عنوان **کلاس والد** برای تعریف **زیرکلاس‌های جدید** استفاده کرد.

```
abstract class Shape {  
    public abstract double area();  
  
    public void display() {  
        System.out.println("This is a shape.");  
    }  
  
}  
  
class Circle extends Shape {  
    private double radius;  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    @Override  
    public double area() {  
        return Math.PI * radius * radius;  
    }  
}
```

```
abstract class Polygon extends Shape {  
}  
  
class Rectangle extends Polygon {  
    private double width;  
    private double height;  
  
    public Rectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    @Override  
    public double area() {  
        return width * height;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Circle c = new Circle(5);  
        Rectangle r = new Rectangle(4, 6);  
  
        c.display();  
        System.out.println("Circle area: " + c.area());  
  
        r.display();  
        System.out.println("Rectangle area: " + r.area());  
    }  
}  
  
This is a shape.  
Circle area: 78.53981633974483  
This is a shape.  
Rectangle area: 24.0
```

توضیح کد

کلاس **Shape** یک کلاس شامل یک متدهای **abstract** به نام **area** میباشد که هیچ پیادهسازی ندارد و همچنین یک متدهای **display** معمولی به نام دارد. این نشان میدهد که در یک کلاس **abstract** میتوان هم متدهای **abstract** و هم متدهای پیادهسازی شده داشت.

کلاس **area** از **Shape** ارث بری میکند. چون **Circle** یک کلاس غیر-**abstract** است، مجبور است متدهای **area** را پیادهسازی کند. در اینجا متدهای **area** را با استفاده از **شعاع** محاسبه میکند.

کلاس **Polygon** نیز از **Shape** ارث بری میکند، اما چون متدهای **area** را پیادهسازی نکرده، خودش هم **area** باقی میماند. این یعنی از کلاس **Polygon** به طور مستقیم نمیتوان نمونه سازی کرد.

کلاس **Rectangle** از **Polygon** ارث بری میکند و متدهای **area** را پیادهسازی کرده است. به همین دلیل، دیگر یک کلاس کامل و غیر-**abstract** است و میتوان از آن شیء ساخت. متدهای **area** در این کلاس مساحت مستطیل را با ضرب عرض در ارتفاع محاسبه میکند.

در کلاس **Main**، یک شیء از **Circle** و یک شیء از **Rectangle** ساخته میشود. سپس با فراخوانی متدهای **area** و **display**، مشخص میشود که هر دو کلاس میتوانند هم متدهای پیادهسازی شده والد و هم متدهای پیادهسازی شده خودشان را استفاده کنند.

در این مثال به طور کامل بررسی کردیم که:

- کلاس های **abstract** میتوانند شامل متدهای **abstract** و همچنین متدهای معمولی باشند.
- زیرکلاس های غیر-**abstract** موظف به پیادهسازی تمامی متدهای **abstract** هستند.
- اگر یک زیرکلاس متدهای **abstract** را پیادهسازی نکند، خودش باید **abstract** تعریف شود.
- تنها کلاس های غیر-**abstract** قابلیت نمونه سازی دارند.

آیا ابرکلاس یک کلاس انتزاعی می‌تواند خود یک کلاس غیرانتزاعی باشد؟

یک زیرکلاس **abstract** می‌تواند **والدی غیر-abstract** داشته باشد.

برای مثال، کلاس **Object** یک کلاس عادی (غیر-**abstract**) است، اما هر کلاس **abstract** می‌تواند فرزند آن باشد.

به مثال زیر دقت کنید:

```
class Vehicle {  
    public void start() {  
        System.out.println("Vehicle is starting...");  
    }  
  
    abstract class Car extends Vehicle {  
        public abstract void drive();  
    }  
  
    class ElectricCar extends Car {  
        @Override  
        public void drive() {  
            System.out.println("ElectricCar is driving silently...");  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        ElectricCar tesla = new ElectricCar();  
        tesla.start();  
        tesla.drive();  
    }  
}
```

```
Vehicle is starting...  
ElectricCar is driving silently...
```

توضیح کد

کلاس **Vehicle** یک کلاس غیر-**abstract** است که متدی به نام **start** دارد و پیاده‌سازی آن آماده استفاده است. این کلاس به عنوان **والد** برای دیگر کلاس‌ها عمل می‌کند.

کلاس **Car** یک زیرکلاس **abstract** است که از **Vehicle** ارث بری می‌کند. این کلاس یک متد **drive** به نام **abstract** دارد که هیچ پیاده‌سازی مشخصی ندارد و باید در زیرکلاس‌های فرزند پیاده‌سازی شود. از آنجا که **Car** یک کلاس **abstract** است، نمی‌توان از آن مستقیماً **نمونه‌سازی** کرد.

کلاس **ElectricCar** یک زیرکلاس غیر-**abstract** است که از **Car** ارث بری می‌کند و متد **drive** را پیاده‌سازی کرده است. از آنجا که این کلاس تمام متدهای **abstract** والد خود را پیاده‌سازی کرده، می‌توان از آن **نمونه‌سازی** کرد.

در کلاس **Main** یک شیء از نوع **ElectricCar** ساخته می‌شود. سپس متد **start** که از کلاس **Vehicle** به ارث برده شده و متد **drive** که در کلاس **ElectricCar** پیاده‌سازی شده، فراخوانی می‌شوند. در نتیجه خروجی برنامه ابتدا پیام شروع وسیله و سپس پیام رانندگی خودروی برقی را چاپ می‌کند.

در این مثال یاد گرفتیم که:

- یک زیرکلاس **abstract** می‌تواند از یک کلاس غیر-**abstract** ارث بری کند.
- تا زمانی که متدهای **abstract** والد در زیرکلاس پیاده‌سازی نشوند، نمی‌توان از آن زیرکلاس **نمونه‌سازی** کرد.
- زمانی که یک زیرکلاس همه متدهای **abstract** والد را پیاده‌سازی می‌کند، خودش تبدیل به یک کلاس غیر-**abstract** شده و قابلیت نمونه‌سازی پیدا می‌کند.
- متدهای ارث بری شده از کلاس والد عادی می‌توانند مستقیماً توسط نمونه کلاس فرزند استفاده شوند.

چند نکته مهم

با اینکه نمی‌توانیم با **new** از یک **کلاس abstract** شیء بسازیم، اما یک **کلاس abstract** می‌تواند به عنوان یک **نوع داده** مورد استفاده قرار گیرد.

برای مثال، می‌توانیم یک **آرایه** از نوع یک کلاس **abstract** تعریف کنیم. در این حالت، متغیرها یا خانه‌های آرایه می‌توانند به اشیاء ساخته شده از **زیرکلاس‌های آن کلاس abstract** اشاره کنند.

نمونه کد زیر نشان‌دهنده همین نکته است:

```
GeometricObject[] geo = new GeometricObject[10];
```

در این مثال:

- عناصر آرایه **geo** می‌توانند به اشیاء ساخته شده از **زیرکلاس‌های **GeometricObject**** اشاره کنند.
- با وجود اینکه نمی‌توانیم از **GeometricObject** به طور مستقیم شیء بسازیم، اما می‌توانیم از آن به عنوان **نوع متغیر** یا **نوع عناصر آرایه** استفاده کنیم.
- عناصر آرایه **geo** می‌توانند به اشیاء ساخته شده از **زیرکلاس‌های **GeometricObject**** مثل **Rectangle** یا **Circle** اشاره کنند.

```
//example of abstract class that have method body
abstract class Bike{
    Bike(){System.out.println("bike is created");}
    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}

class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}

class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

```
bike is created
running safely..
gear changed
```

توضیح کد

کلاس **Bike** یک کلاس **abstract** است. این کلاس شامل یک سازنده است که هنگام ایجاد شیء پیام "**bike is created**" را چاپ می‌کند. همچنین دارای یک متدهای **run** (بدون پیاده‌سازی) و یک متدهای **changeGear** (با بدنه و پیاده‌سازی) است که پیام "**gear**" را چاپ می‌کند.

کلاس **Bike** از کلاس **Honda** ارث بری می‌کند و چون **Bike** متدهای **run** و **changeGear** را از **Honda** موصوف است آن را پیاده‌سازی کند. در این کلاس متدهای **run** و **changeGear** پیاده‌سازی شده و پیام "**running safely..**" را چاپ می‌کند.

کلاس **TestAbstraction2** شامل متدهای **main** است که نقطه شروع برنامه محسوب می‌شود. در این متدها:

- یک شیء از نوع **Bike** اما با سازنده **Honda** ساخته می‌شود. به دلیل خاصیت **polymorphism**، شیء ساخته شده از نوع **Honda** است اما به متغیری از نوع **Bike** اختصاص داده می‌شود.
- به محض ایجاد شیء، سازنده **Bike** فراخوانی می‌شود و پیام "**bike is created**" چاپ می‌شود.
- متدهای **run** روی شیء فراخوانی می‌شود و به دلیل اینکه پیاده‌سازی آن در **Honda** انجام شده، پیام "**running safely..**" چاپ می‌شود.
- سپس متدهای **changeGear** فراخوانی شده و پیام "**gear changed**" نمایش داده می‌شود.

این کد نشان می‌دهد که:

- یک کلاس **abstract** می‌تواند شامل **متدهای abstract** و همچنین **متدهای معمولی** با بدنه پیاده‌سازی شده باشد.
- زیرکلاس یک کلاس **abstract** موظف است تمامی **متدهای abstract** آن را پیاده‌سازی کند تا بتوان از آن نمونه‌سازی کرد.
- خاصیت **polymorphism** در جاوا اجازه می‌دهد متغیر والد (در اینجا **Bike**) به شیء ساخته شده از کلاس فرزند (در اینجا **Honda**) ارجاع دهد.
- هنگام ایجاد شیء از کلاس فرزند، ابتدا سازنده والد فراخوانی می‌شود و سپس سازنده فرزند.

یکی از کاربردهای کلاس انتزاعی

یکی از مهم‌ترین کاربردهای کلاس **abstract** این است که می‌تواند برای پیاده‌سازی بخشی از متدهای یک **interface** مورد استفاده قرار گیرد. این کار باعث می‌شود که دیگر لازم نباشد هر کلاسی که از آن **interface** استفاده می‌کند، همهٔ متدهای آن را از ابتدا پیاده‌سازی کند.

در واقع، کلاس **abstract** به عنوان یک **لایهٔ میانی** بین **interface** و کلاس‌های معمولی عمل می‌کند. به این ترتیب، بخش‌هایی از پیاده‌سازی که برای همهٔ کلاس‌ها یکسان است، فقط یک بار در کلاس **abstract** نوشته می‌شود و کلاس‌های فرزند می‌توانند از آن استفاده کنند یا متدهای خاص خود را بازنویسی کنند.

مزایای این رویکرد:

- کاهش **تکرار کد** زیرا پیاده‌سازی متدهای مشترک در یک کلاس **abstract** انجام می‌شود.
- افزایش **خوانایی و نگهداری** کد چون تغییرات در منطق مشترک فقط در یک نقطه (کلاس **abstract**) انجام می‌شود.
- امکان اعمال **قرارداد (Contract)** از طریق **interface** و در عین حال دادن پیاده‌سازی پایه از طریق کلاس **abstract**.

نمونهٔ مفهومی:

توضیح مثال:

- قراردادی تعریف می‌کند که هر کلاسی که آن را پیاده‌سازی کند باید متدهای **resize** و **draw** را داشته باشد.
- کلاس **Shape** یک کلاس **abstract** است که متد **resize** را پیاده‌سازی کرده و متد **draw** را به کلاس‌های فرزند واگذار می‌کند.
- کلاس **Circle** از کلاس **Shape** ارث بری می‌کند و متد **draw** را پیاده‌سازی می‌کند.
- در متد **main** می‌توان از قابلیت‌های هم کلاس **abstract** و هم کلاس فرزند استفاده کرد.

این روش به توسعه‌دهندگان اجازه می‌دهد بخشی از پیاده‌سازی را یکجا در کلاس **abstract** انجام دهند و بقیهٔ جزئیات را به کلاس‌های فرزند بسپارند. این امر انعطاف‌پذیری، کاهش تکرار کد و معماری تمیزتر در پروژه‌های بزرگ را به همراه دارد.

```
interface Drawable {
    void draw();
    void resize();
}

abstract class Shape implements Drawable {
    public void resize() {
        System.out.println("Resizing shape...");
    }
}

class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing circle");
    }
}

class Main {
    public static void main(String[] args) {
        Shape c = new Circle();
        c.draw();
        c.resize();
    }
}
```

Drawing circle
Resizing shape...

توضیح کد

توضیح مثال:

- قراردادی تعریف می‌کند که هر کلاسی که آن را پیاده‌سازی کند باید متدهای `draw` و `resize` را داشته باشد.
- کلاس `Shape` یک کلاس `abstract` است که متدهای `draw` را پیاده‌سازی کرده و متدهای `resize` را به کلاس‌های فرزند واگذار می‌کند.
- کلاس `Circle` از کلاس `Shape` ارث بری می‌کند و متدهای `draw` را پیاده‌سازی می‌کند.
- در متدهای `main` می‌توان از قابلیت‌های هم کلاس `abstract` و هم کلاس فرزند استفاده کرد.

این روش به توسعه‌دهندگان اجازه می‌دهد بخشی از پیاده‌سازی را یکجا در کلاس `abstract` انجام دهند و بقیه جزئیات را به کلاس‌های فرزند بسپارند. این امر انعطاف‌پذیری، کاهش تکرار کد و معماری تمیزتر در پروژه‌های بزرگ را به همراه دارد.

Interface

در جاوا یک طرح کلی (**Contract**) برای کلاس‌ها است که فقط شامل تعاریف متدها و ثابت‌های استاتیک می‌باشد و هیچ پیاده‌سازی (بدنه متدها) ندارد. تمام متدهای یک interface به صورت **abstract** در نظر گرفته می‌شوند.

در جاوا سازوکاری برای ایجاد انتزاع (**Abstraction**) کامل است. به این معنی که با استفاده از interface، می‌توان یک لایه کاملاً انتزاعی تعريف کرد تا کلاس‌ها از آن پیروی کنند (abstract 100%).

ویژگی‌های کلیدی interface:

- در interface تمام متدها به صورت **abstract** هستند و پیاده‌سازی ندارند.
- به ما اجازه می‌دهد تا انتزاع کامل داشته باشیم و پیاده‌سازی متدها را به کلاس‌هایی که از interface استفاده می‌کنند واگذار کنیم.
- امکان ارث‌بری چندگانه را فراهم می‌کند، زیرا یک کلاس می‌تواند چندین interface را پیاده‌سازی کند.
- مانند کلاس‌های **abstract**، نمی‌توانیم از interface مستقیماً نمونه‌سازی کنیم.

با استفاده از interface می‌توان قراردادهای مشخصی را برای کلاس‌ها تعريف کرد و هر کلاس موظف است متدهای تعريف شده در interface را پیاده‌سازی کند. این رویکرد باعث انعطاف‌پذیری و قابلیت توسعه در پروژه‌های بزرگ می‌شود.

حذف سطوح دسترسی در Interface

تعریف فیلدها و متدها در **interface**: در یک **interface** تمام فیلدها به صورت **final static** و **final** هستند و همه متدها به صورت **abstract** در نظر گرفته می‌شوند. این یعنی فیلدها ثابت و اشتراکی هستند و پیاده‌سازی برای متدها در خود **interface** نوشته نمی‌شود.

حذف کلمات کلیدی: چون این ویژگی‌ها ضمنی هستند، نوشتن **public static final** برای فیلدها و **public abstract** برای متدها ضروری نیست و می‌توان آن‌ها را حذف کرد. نتیجه یکی است و کد خواناتر می‌شود.

مثال‌های معادل: دو تعریف زیر برای یک **interface** کاملاً همارز هستند و خروجی یکسانی دارند.

```
public interface T1 {  
    public static final int K = 1;  
    public abstract void p();  
}
```

```
public interface T1 {  
    int K = 1;  
    void p();  
}
```

دسترسی به مقادیر ثابت: برای ارجاع به ثابت‌های داخل یک interface از الگو شفاف و استاندارد است و نشان می‌دهد ثابت متعلق به کدام interface است.

InterfaceName.CONSTANT_NAME

نمونه دسترسی: اگر در interface به نام **T1** ثابتی با نام **K** داشته باشیم، به این شکل به آن دسترسی می‌گیریم.

T1.K

نکته مهم درباره ثابت‌ها: فیلدهای داخل interface همیشه public static final هستند، بنابراین مقدارشان قابل تغییر نیست و در سطح کلاس مشترک‌اند. این ویژگی باعث می‌شود ثابت‌ها برای همه پیاده‌سازی‌ها و مصرف‌کنندگان یکسان باشند.

```
public interface T1 {  
    public static final int K = 1;  
  
    public abstract void p();  
}
```

Equivalent

```
public interface T1 {  
    int K = 1;  
  
    void p();  
}
```

حذف سطوح دسترسی در Interface

مثال اولیه: در فایل منبع (`Printable.java`) می‌توان به صورت کوتاه زیر نوشت:

```
interface Printable {  
    int MIN = 5;  
    void print();  
}
```

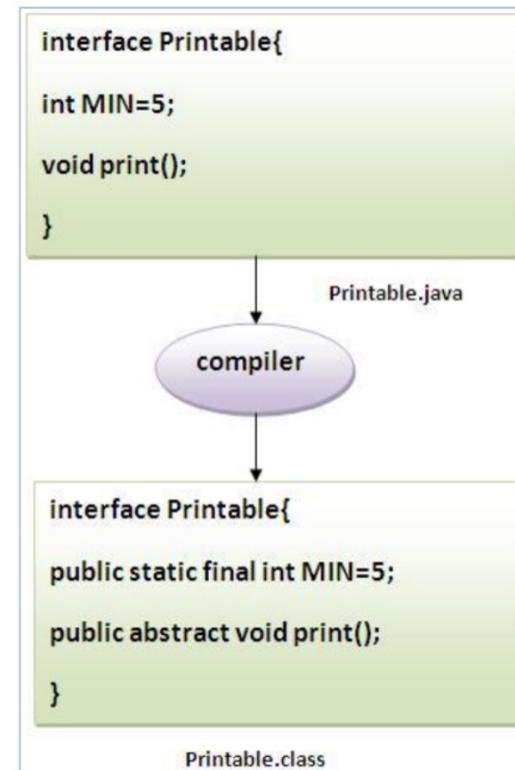
تبديل توسط کامپایلر: وقتی این کد کامپایل می‌شود، کامپایلر آن را به معادل زیر تبدیل می‌کند و به صورت ضمنی کلیدواژه‌های `public static final` برای فیلدها و `public abstract` برای متدها اضافه می‌کند:

```
interface Printable {  
    public static final int MIN = 5;  
    public abstract void print();  
}
```

دلیل این رفتار: در `interface` نیازی به نوشتن صریح این کلمات نیست زیرا پیش‌فرض زبان جاوا همین است. نوشتن آن‌ها فقط برای تأکید یا خوانایی است و تفاوتی در عملکرد ایجاد نمی‌کند.

نمونه استفاده: برای دسترسی به مقدار ثابت MIN از interface باла به صورت زیر عمل می‌کنیم:

Printable.MIN



فهم رابطه بین کلاس‌ها و Interface ها

رابطه کلاس با کلاس: در جاوا یک کلاس می‌تواند از یک کلاس دیگر ارث بری extends تعریف می‌شود. در این حالت، کلاس فرزند ویژگی‌ها و متدهای قابل ارث را از کلاس والد دریافت می‌کند.

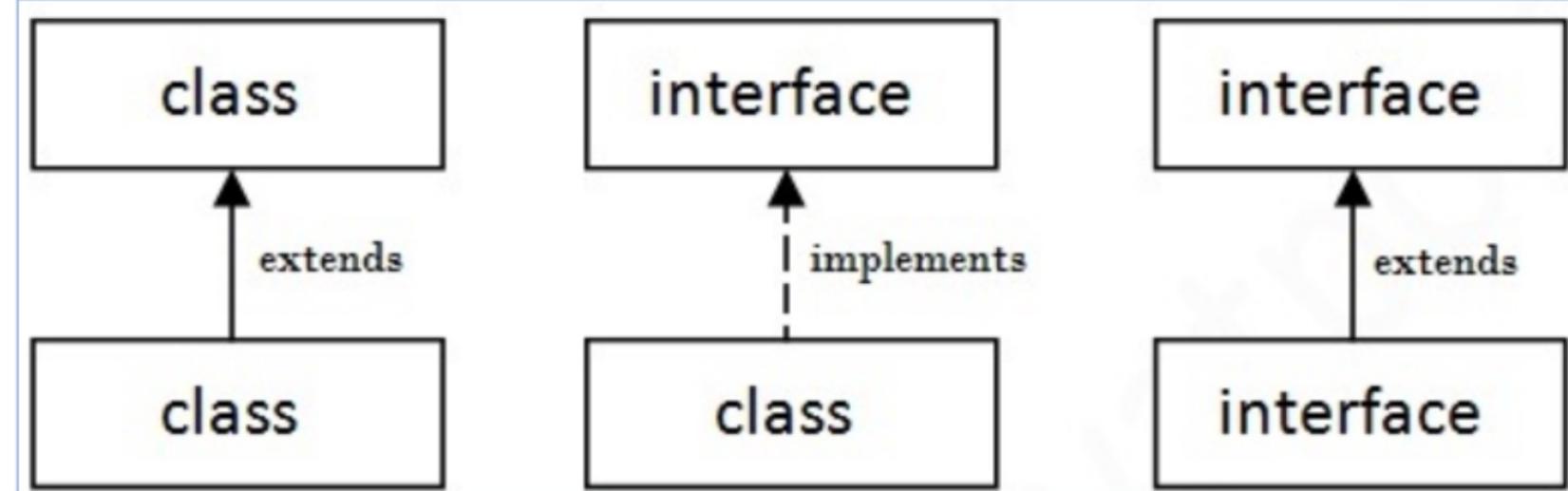
رابطه کلاس با اینترفیس: یک کلاس برای پیاده‌سازی قرارداد یک اینترفیس از کلیدواژه implements استفاده می‌شود که تمام متدهای abstract تعریف شده در آن اینترفیس را پیاده‌سازی کند.

رابطه اینترفیس با اینترفیس: یک اینترفیس می‌تواند از یک یا چند اینترفیس extends دیگر ارث بری کند. در این حالت از کلیدواژه استفاده می‌شود. این کار برای ترکیب چند قرارداد در یک اینترفیس جدید مفید است.

نکات :

- کلاس‌ها فقط می‌توانند از یک کلاس دیگر ارث بری کنند.
- کلاس‌ها می‌توانند چندین اینترفیس را پیاده‌سازی کنند.
- یک اینترفیس می‌تواند از چندین اینترفیس دیگر ارث بری کند.
- هیچ اینترفیسی از یک کلاس ارث بری نمی‌کند.
- هیچ کلاسی نمی‌تواند با extends از یک اینترفیس ارث بری کند.

ارث بری در کلاس‌ها برای به اشتراک‌گذاری پیاده‌سازی و ویژگی‌ها استفاده می‌شود، در حالی که استفاده از اینترفیس برای تعریف قراردادهای رفتاری و توانایی‌های مشترک بین کلاس‌های نامرتبط کاربرد دارد.



شباهت‌ها و تفاوت‌های Interface, Class

شباهت با کلاس abstract: در بسیاری از موارد، **interface** نقشی مشابه کلاس **abstract** ایفا می‌کند. هر دو نمی‌توانند به صورت مستقیم نمونه‌سازی (instantiate) شوند و از آن‌ها برای تعریف قراردادها و رفتارهای مشترک استفاده می‌شود.

تفاوت‌های کلیدی: با وجود شباهت‌ها، تفاوت‌های مهمی میان **interface** و **کلاس abstract** وجود دارد. در **کلاس abstract** می‌توان فیلدهای نمونه و متدهای دارای پیاده‌سازی داشت، اما در **interface** فقط امضای متدها و ثابت‌ها تعریف می‌شود (تا قبل از اضافه شدن **default** و **static method** در نسخه‌های جدید جاوا). به همین دلیل، **interface** برای تعریف **قرارداد رفتاری** مناسب‌تر است.

هدف از interface: هدف اصلی **interface** توصیف و بیان **Riftar** است، نه ذخیره‌سازی وضعیت یا داده. یعنی به ما می‌گوید که یک موجودیت باید چه کارهایی انجام دهد، بدون این‌که بگویید چگونه آن‌ها را انجام دهد.

تعريف یک Edible

مثال: قراردادی است که می‌گوید هر چیزی که آن را پیاده‌سازی کند باید **روش خوردن** را برگرداند. متدها در interface به صورت پیش‌فرض هستند؛ نوشتن آن‌ها اختیاری است.

```
public interface Edible {  
    public abstract String howToEat();  
}
```

```
public interface Edible {  
    String howToEat();  
}
```

پیاده‌سازی نمونه: هر کلاسی که این قرارداد را بپذیرد باید متدهای **String** **howToEat** را پیاده‌سازی کند. خروجی یک **String** توصیفی است و نوع داده یا منطق آماده‌سازی به انتخاب کلاس است.

```
public class Apple implements Edible {  
    @Override  
    public String howToEat() { return "slice and eat fresh"; }  
}  
  
public class Chicken implements Edible {  
    @Override  
    public String howToEat() { return "cook before eating"; }  
}
```

کاربرد: با معرفی یک interface مانند **Edible** می‌توان کلاس‌های نامرتبط را در یک **قرارداد مشترک** جمع کرد و به کمک **چندنما** آنها را یکجا پردازش کرد، بدون وابستگی به جزئیات پیاده‌سازی هر کلاس.

یک کلاس خاص است Interface

ماهیت **interface**: در جاوا، **interface** مانند یک **کلاس ویژه** است که از آن برای تعریف قراردادهای رفتاری استفاده می‌شود. با این حال، برخلاف کلاس معمولی، نمی‌توان از آن نمونه‌سازی (instantiate) کرد.

نیاز به فایل جداگانه: هر **interface** باید در یک **فاایل جداگانه** تعریف شود. این فایل با همان نام **interface** ذخیره می‌شود و ساختار آن مشابه فایل کلاس است، اما به جای کلیدواژه **class** از **interface** استفاده می‌شود.

عدم امکان نمونه‌سازی مستقیم: مشابه **کلاس abstract**، نمی‌توان با استفاده از کلیدواژه **new** از یک **interface** شیء ایجاد کرد. زیرا **interface** فقط تعریف‌کننده **Riftar** است و هیچ پیاده‌سازی در خود ندارد.

کاربرد در الگوهای مشابه: در بسیاری از موارد، از **interface** برای الگوهای مشابهی که نیاز به تعیین **Riftar مشترک** بین کلاس‌های متفاوت دارند استفاده می‌شود. این Riftar می‌تواند مستقل از ساختار و سلسله‌مراتب کلاس‌ها باشد.

استفاده به عنوان نوع داده: می‌توان از یک **interface** به عنوان نوع داده برای یک متغیر استفاده کرد. به این ترتیب، آن متغیر می‌تواند هر شیئی را که آن **interface** را پیاده‌سازی کرده است، در خود نگه دارد.

اگر **interface** به نام **Edible** تعریف شده باشد، می‌توان یک متغیر از نوع **Edible** ایجاد کرد و به آن شیئی از کلاسی را نسبت داد که **Edible** را پیاده‌سازی کرده است. این روش باعث انعطاف‌پذیری بیشتر برنامه می‌شود.

مثال

با یک **Edible** (قابل خوردن) مشخص می‌کنیم که آیا شیئی خوردنی می‌باشد یا خیر؟

به این منظور، کلاس‌هایی که می‌خواهیم بزنیم، اشیایی ایجاد شده از آن‌ها قابل خوردن هستند، با کلیدواژه **implements** این **interface** را پیاده‌سازی می‌کنند.

برای مثال، کلاس‌های **Fruit** و **Chicken** خوردنی (edible) است، پیاده‌سازی کرده‌اند.

```
public interface Edible {  
    /** Describe how to eat */  
    public abstract String howToEat();  
}  
  
public class TestEdible {  
    public static void main(String[] args) {  
        Object[] objects = {new Tiger(), new Chicken(), new Apple()};  
        for (int i = 0; i < objects.length; i++)  
            if (objects[i] instanceof Edible)  
                System.out.println((Edible)objects[i]).howToEat();  
    }  
}
```

```
class Animal {  
    // Data fields, constructors, and methods omitted here  
}  
  
class Chicken extends Animal implements Edible {  
    public String howToEat() {  
        return "Chicken: Fry it";  
    }  
}
```

```
class Tiger extends Animal {  
}  
  
abstract class Fruit implements Edible {  
    // Data fields, constructors, and methods omitted here  
}  
  
class Apple extends Fruit {  
    public String howToEat() {  
        return "Apple: Make apple juice";  
    }  
}  
  
class Orange extends Fruit {  
    public String howToEat() {  
        return "Orange: Make orange juice";  
    }  
}
```

Chicken: Fry it
Apple: Make apple juice

مثال ساده از Interface در جاوا

در این مثال یک متده دارد و پیادهسازی آن توسط کلاس A انجام می‌گیرد.

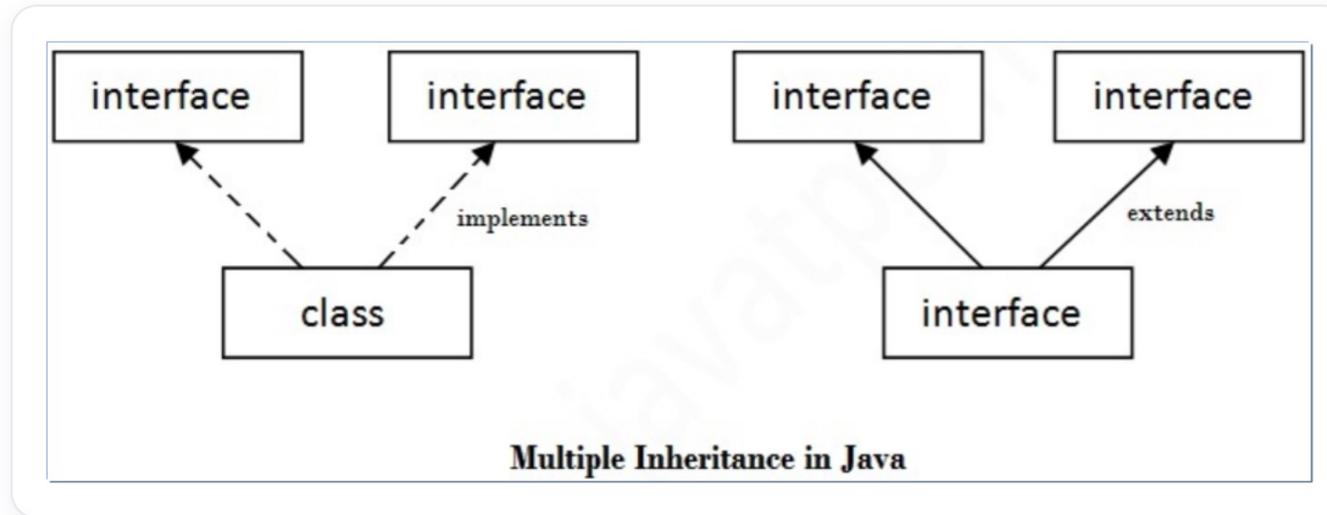
```
interface printable {
    void print();
}

class A6 implements printable {
    public void print() {
        System.out.println("Hello");
    }
}

public static void main(String args[]) {
    A6 obj = new A6();
    obj.print();
}
```

Hello

ارثبری چندگانه در جاوا به کمک Interface



ارثبری چندگانه در جاوا (**Multiple Inheritance in Java**) مفهومی است که به یک کلاس یا یک اینترفیس اجازه می‌دهد از چندین اینترفیس ارثبری کند یا آن‌ها را پیاده‌سازی نماید.

در جاوا، **کلاس‌ها** نمی‌توانند از چند کلاس دیگر ارثبری کنند زیرا این کار باعث ایجاد **ابهام** (مانند مسئله الماس) می‌شود. اما جاوا این امکان را با استفاده از **اینترفیس‌ها** فراهم کرده است تا یک کلاس بتواند چندین اینترفیس را **پیاده‌سازی** کند یا یک اینترفیس از چندین اینترفیس دیگر **ارثبری** نماید.

تعریف: اگر یک **کلاس** بیش از یک اینترفیس را پیاده‌سازی کند یا یک اینترفیس از چندین اینترفیس دیگر ارثبری کند، به این ویژگی **ارثبری چندگانه** گفته می‌شود.

در شکل نمایش داده شده:

- در سمت چپ، یک **کلاس** وجود دارد که چندین **اینترفیس را پیاده‌سازی** کرده است. این نشان می‌دهد که یک کلاس می‌تواند رفتارهای تعریف شده در چندین اینترفیس را در خود پیاده‌سازی نماید.
 - در سمت راست، یک **اینترفیس** از چندین **اینترفیس دیگر ارث بری** می‌کند. به این ترتیب، این اینترفیس جدید تمام متدهای انتزاعی (abstract) تعریف شده در اینترفیس‌های والد را نیز به ارث می‌برد.
- مزیت اصلی:** این روش اجازه می‌دهد بدون مشکل ابهام در ارث بری، یک کلاس یا اینترفیس چندین رفتار مختلف را از منابع گوناگون به ارث ببرد یا پیاده‌سازی کند.
- نکته:** این رویکرد مخصوصاً برای مدیریت **چندین قابلیت** (مثل قابل خوردن بودن، قابل چاپ بودن، و غیره) در یک کلاس مفید است، بدون اینکه نیاز باشد آن قابلیتها را از کلاس‌های مختلف به ارث ببریم.

```
interface Printable {
    void print();
}

interface Showable {
    void show();
}

class A7 implements Printable, Showable {
    public void print() {
        System.out.println("Hello");
    }
    public void show() {
        System.out.println("Welcome");
    }

    public static void main(String args[]) {
        A7 obj = new A7();
        obj.print();
        obj.show();
    }
}
```

Hello
Welcome

ارث بری چندگانه در جاوا به کمک Interface

در زبان **جاوا**، **ارث بری چندگانه** (Multiple Inheritance) برای **کلاس ها** پشتیبانی نمی شود. به این معنی که یک کلاس نمی تواند به صورت مستقیم از چند کلاس مختلف ارث بری کند. دلیل این محدودیت جلوگیری از بروز **ابهام** در ساختار برنامه و پیچیدگی های احتمالی است.

اما با استفاده از **interface** می توان این محدودیت را برطرف کرد. زمانی که یک کلاس چندین **interface** را **پیاده سازی** (implements) می کند، هیچ ابهامی ایجاد نمی شود زیرا **interface** ها تنها شامل **تعریف متدها** هستند و پیاده سازی ندارند.

بنابراین، با کمک **interface** می توان به نوعی **ارث بری چندگانه** دست یافت، بدون اینکه مشکلات ناشی از ارث بری چندگانه کلاس ها به وجود آید. این ویژگی یکی از مهم ترین کاربردهای **interface** در جاوا محسوب می شود.

```
interface Printable {
    void print();
}

interface Showable extends Printable {
    void show();
}

class Testinterface2 implements Showable {

    public void print() {
        System.out.println("Hello");
    }

    public void show() {
        System.out.println("Welcome");
    }

    public static void main(String args[]) {
        Testinterface2 obj = new Testinterface2();
        obj.print();
        obj.show();
    }
}
```

Hello
Welcome

```
interface A {  
    void a();  
    void b();  
    void c();  
    void d();  
}  
  
abstract class B implements A {  
    public void c() {  
        System.out.println("I am C");  
    }  
}  
  
class M extends B {  
    public void a() {  
        System.out.println("I am a");  
    }  
    public void b() {  
        System.out.println("I am b");  
    }  
    public void d() {  
        System.out.println("I am d");  
    }  
}
```

```
class Test5 {  
    public static void main(String args[]) {  
        A a = new M();  
        a.a();  
        a.b();  
        a.c();  
        a.d();  
    }  
}
```

I am a
I am b
I am C
I am d

مقایسه Abstract class و Interface

Interface	Abstract Class
Methods can be declared	Methods can be declared
No method bodies	Method bodies can be defined
Constants can be declared	All types of variables can be declared
Has no constructors	Can have constructors
Multiple inheritance possible	Multiple inheritance not possible
Has no top interface	Always inherits from Object
Multiple parent interfaces	Only one parent class

marker or tagged interface?

در جاوا، زمانی که یک **interface** هیچ متدهای فیلیدی ندارد، به آن **Tagged Interface** یا **Marker Interface** گفته می‌شود. این نوع اینترفیس‌ها تنها برای انتقال اطلاعات متادیتا به **JVM** استفاده می‌شوند و هیچ رفتار خاصی را تعریف نمی‌کنند.

برخی از مثال‌های معروف از این نوع اینترفیس‌ها عبارت‌اند از: **Remote**, **Serializable**, **Cloneable** و **Serializable**. این اینترفیس‌ها برای مشخص کردن یک ویژگی خاص در کلاس‌ها استفاده می‌شوند؛ مثلاً وقتی یک کلاس **Serializable** را پیاده‌سازی می‌کند، در واقع به **JVM** اطلاع می‌دهد که این شیء قابل ذخیره‌سازی یا انتقال است.

در زمان اجرا، **JVM** با بررسی اینکه آیا یک کلاس، یک **Marker Interface** را پیاده‌سازی کرده است یا خیر، تصمیم می‌گیرد عملیات خاصی را انجام دهد. برای نمونه، اگر کلاسی **Serializable** باشد، **JVM** اجازه‌ی نوشتتن و خواندن آن شیء در جریان داده را می‌دهد.

```
//How serializable interface is written
public interface Serializable{

}
```

برخی از واسطه‌های محبوب در جاوا

برای پیمایش بر روی مجموعه‌ای (**collection**) از اشیاء بدون نیاز به دانستن نحوه ذخیره‌سازی آنها استفاده می‌شود. برای مثال در **list** یا از این رابط استفاده می‌شود تا بتوان به ترتیب به عناصر دسترسی پیدا کرد. توضیحات کامل‌تر آن در بخش **Collections** ارائه می‌شود.

برای ایجاد یک کپی از شیء موجود از طریق متدهای **clone()** که در کلاس **Object** قرار دارد استفاده می‌شود. اگر یک کلاس این اینترفیس را پیاده‌سازی کند، نشان می‌دهد که اجازه‌ی کپی‌سازی ایمن از آن شیء وجود دارد.

برای **بسته‌بندی (pack)** و ذخیره‌سازی اشیاء به گونه‌ای که بتوان آنها را از طریق شبکه انتقال داد یا بر روی دیسک ذخیره کرد، استفاده می‌شود. هدف این است که شیء به بایت‌ها تبدیل شده و سپس قابل بازیابی (بازسازی) باشد. **JVM** با استفاده از این اینترفیس می‌تواند عملیات مربوط به ورودی/خروجی (**I/O**) را بر روی اشیاء انجام دهد.

برای ایجاد یک ترتیب کلی (**total order**) بین اشیاء به کار می‌رود. این اینترفیس معمولاً برای مرتب‌سازی مجموعه‌ها یا مقایسه‌ی دو شیء با هم استفاده می‌شود.

واسط Iterator

یکی از رابطه‌های مهم در پکیج **java.util** است که برای پیمایش بر روی عناصر ذخیره‌شده در مجموعه‌ها (**collections**) مورد استفاده قرار می‌گیرد. این واسط روشی استاندارد برای دسترسی ترتیبی به اشیاء بدون نیاز به دانستن ساختار داخلی مجموعه فراهم می‌کند. با استفاده از **Iterator** می‌توان به ترتیب عناصر را بررسی، خوانده و حتی حذف کرد.

```
package java.util;

public interface Iterator {
    // the full meaning is public abstract boolean hasNext()
    boolean hasNext();
    Object next();
    void remove(); // optional throws exception
}
```

Methods with Description	SN
boolean hasNext() بازمی‌گرداند true اگر عنصر بعدی وجود داشته باشد، در غیر این صورت false.	1
Object next() عنصر بعدی را بازمی‌گرداند. در صورت نبود عنصر بعدی، استثنای NoSuchElementException پرتاب می‌کند.	2
void remove() عنصر فعلی را حذف می‌کند. اگر قبل از فراخوانی next() متدهای remove() و next() صدا زده شود، استثنای IllegalStateException پرتاب می‌کند.	3

متدهایی که آیا در مجموعه (collection) عنصر دیگری برای پیمایش وجود دارد یا خیر. اگر عنصر بعدی وجود داشته باشد مقدار بررسی می‌کند که آیا در مجموعه `hasNext()` برای پیمایش وجود دارد یا خیر. این متدهایی کنترل ادامهی حلقه در هنگام پیمایش مجموعه استفاده می‌شود.

متدهایی که آیا در مجموعه را بازمی‌گرداند. در صورتی که دیگر عنصری برای بازگشت وجود نداشته باشد، این متدهای استثنای `next()` عنصر بعدی موجود در مجموعه را بازمی‌گردانند. در صورتی که دیگر عنصری برای بازگشت وجود نداشته باشد، این متدهای استثنای `NoSuchElementException` را پرتاب می‌کند. معمولاً این متدهای همراه با `hasNext()` به کار می‌روند تا در زمان پیمایش، از وجود عنصر بعدی اطمینان حاصل شود.

متدهایی که عنصر فعلی از مجموعه استفاده می‌شود. این متدهای فعال در صورتی مجاز به اجراست که پیش از آن متدهای `remove()` فراخوانی شده باشند. در غیر این صورت، اجرای آن باعث پرتاب استثنای `IllegalStateException` می‌شود. استفاده از این متدهای امکان می‌دهد تا در حین پیمایش، عناصر خاصی را از مجموعه حذف کنید بدون آنکه به ساختار داخلی مجموعه دست بزنید.

واسط Clonable

یک واسط نشانه‌گذاری در جاوا است که برای مشخص کردن قابلیت **کپی‌سازی (cloning)** در کلاس‌ها استفاده می‌شود. این واسط به **JVM** اطلاع می‌دهد که اشیای ساخته شده از آن کلاس می‌توانند با اطمینان کپی شوند.

این واسط هیچ متدهای فیلدی ندارد و تنها به عنوان یک علامت برای **JVM** عمل می‌کند. اگر کلاسی این واسط را پیاده‌سازی نکرده باشد و متدهای **clone()** روی شیء آن فراخوانی شود، برنامه با استثنای **CloneNotSupportedException** مواجه خواهد شد.

در مثال کلاس **Car**، واسط **Cloneable** پیاده‌سازی شده تا اشیای این کلاس قابل کپی باشند. متدهای **clone()** در این کلاس با ایجاد یک شیء جدید از نوع **Car** و کپی کردن مقادیر ویژگی‌های **make**, **model** و **price** از شیء فعلی تولید می‌کند.

```
// Car example revisited
public class Car implements Cloneable {
    private String make;
    private String model;
    private double price;

    // default constructor
    public Car() {
        this("", "", 0.0);
    }

    // give reasonable values to instance variables
    public Car(String make, String model, double price) {
        this.make = make;
        this.model = model;
        this.price = price;
    }

    // the Cloneable interface
    public Object clone() {
        return new Car(this.make, this.model, this.price);
    }
}
```

خودمون رو بسنجیم

این بخش برای این طراحی شده که در پایان مطالعه این اسلاید، بتونی خودت رو محک بزنی و ببینی آیا مفاهیم رو به خوبی یاد گرفتی یا نه. سوالات زیر را مرور کن و سعی کن بدون نگاه کردن به متن درس، به اون ها پاسخ بدی.

- مفهوم انتزاع رو توضیح بدید.
- آیا از یک **کلاس abstract** میتوان یک شی ایجاد کرد؟
- واسطه‌ها و کلاس‌های انتزاعی را با هم مقایسه کنید.
- ارثبری چندگانه در جاوا به چه صورت امکان‌پذیر است؟

پایان

در صورت هرگونه سوال یا پیشنهاد میتوانید با من در ارتباط باشید
(:)

gmail: mr.mohamad.hoseini05@gmail.com
telegram: @MHosseiniR