

# کلاس های abstract و interface

Interface ها و کلاس های abstract راه حلی ساخت یافته  
برای جداسازی واسط از پیاده سازی را فراهم می کنند



# متدها و کلاس های Abstract

□ گاهی اوقات متدهای کلاس والد ساختگی و ظاهری هستند. در واقع هدف چنین کلاسی ایجاد یک رابط مشترک برای همه کلاس های مشتق شده از آن است.

□ زبان برنامه نویسی جاوا از مکانیزمی به نام متد abstract برای انجام این کار استفاده می کند. این متد، متدی ناقص است که تنها قسمت تعریف دارد و بدنه متد در اینجا تعریف نمی شود.

```
abstract void f( );
```

# متدها و کلاس های Abstract

□ کلاسی که شامل متدهای abstract باشد کلاس abstract نامیده می شود.

□ در صورتی که کلاسی دارای یک و یا چند متد abstract باشد، خود کلاس نیز باید به طریقی مشخص شود که از نوع کلاس abstract است. (در غیر این صورت، کامپایلر پیغام خطا می دهد.)

# متدها و کلاس های Abstract

```
abstract class Instrument {  
    private int i;  
    public abstract void play(Note n);  
    public String what() {  
        return "Instrument";  
    }  
    public abstract void adjust();  
}
```

```
class Wind extends Instrument {  
    public void play(Note n) {  
        print("Wind.play() " + n);  
    }  
    public String what() { return "Wind"; }  
    public void adjust() {  
        //Implementation  
    }  
}
```

# interface ها

□ کلید واژه interface یک کلاس abstract میسازد که هیچ یک از متود های آن پیاده سازی ای ندارد.

□ از این طریق برنامه نویس می تواند نام متد، لیست آرگومان ها و نوع برگشتی را تعیین کند. اما متدها بدنه ای ندارند.

□ یک interface می گوید: "همه کلاس هایی که این interface خاص را پیاده سازی می کنند، شبیه به این کلاس خواهند بود." بنابراین interface برای ایجاد پروتکلی بین کلاس ها استفاده می شود.

# interface

```
interface Instrument {  
    int VALUE = 5; //static & final  
    //Cannot have method definitions:  
    void play(Note n);  
    // Automatically public void adjust();  
}  
  
class Wind implements Instrument {  
    public void play(Note n) {  
        print(this + ".play() " + n);  
    }  
    public String toString() { return "Wind"; }  
    public void adjust() { print(this + ".adjust()"); }  
}
```

# interface ها

□ برای ایجاد یک interface، از کلیدواژه interface به جای کلیدواژه class استفاده کنید.

□ یک interface می تواند شامل فیلدها هم باشد. اما هر فیلد آن به صورت ضمنی static و final هستند.

□ برای آنکه کلاسی به یک interface خاص (یا گروهی از interface ها) نسبت داده شوند، از کلیدواژه implements استفاده می شود.

□ interface می گوید به چه چیزی شبیه است اما implements می گوید که چگونه عملیات انجام می گیرد .

# interface ها

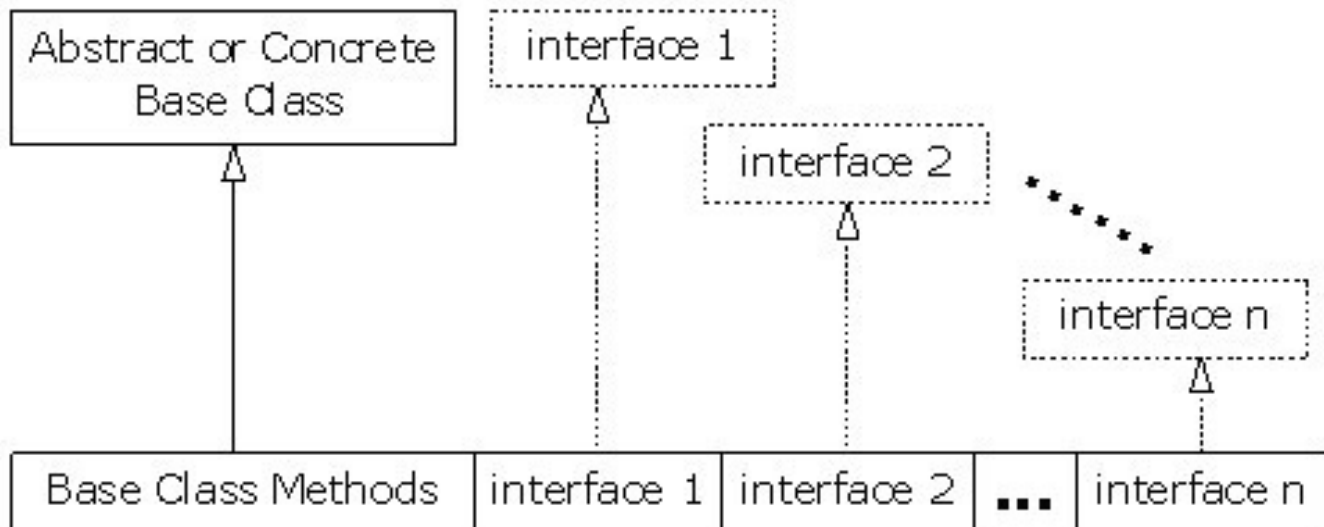
□ یک interface چیزی بیشتر از یک کلاس abstract است، چراکه از طریق آن می توان "ارث بری چندگانه" را پیاده سازی کرد بدین ترتیب با ایجاد کلاسی می توان آن را به چند نوع پایه ، upcast کرد.



# ارث بری چندگانه در جاوا

یک interface هیچ پیاده سازی ای ندارد و این به این معناست که هیچ حافظه ای به interface تخصیص داده نمی شود.

هیچ محدودیتی برای تلفیق interface ها با هم وجود ندارد و این هنگامی ارزشمند است که نیاز داریم بگوییم: "یک x در واقع یک a و یک b و یک c است."



# ارث بری چندگانه در جاوا

□ به یاد داشته باشید که:

□ یکی از دلایل اصلی استفاده از interface همانطور که قبلاً گفته شد: برای upcast به بیش از یک نوع پایه (و انعطاف پذیری ای که این روش ایجاد می کند).

□ دلیل دوم استفاده از interface ها همانند استفاده از یک کلاس abstract است: برای جلوگیری از ایجاد object ای از این کلاس توسط برنامه نویس client و برای تاکید بر این مفهوم که این تنها یک interface است

# Extend کردن یک interface با استفاده از ارث بری

□ به راحتی می توان تعاریف جدیدی از متد را از طریق ارث بری به یک interface نسبت داد. همچنین می توان از طریق ارث بری، چندین interface را با هم تلفیق کرد و interface جدیدی ایجاد نمود.

```
interface Monster {  
    void menace();  
}  
interface DangerousMonster extends Monster {  
    void destroy();  
}
```

# تصادم نام ها در هنگام تلفیق interface ها

□ هنگام پیاده سازی چند interface ممکن است با مشکلی

غیرمنتظره روبرو شویم. در صورتی که متد دارای امضا و نوع بازگشتی متفاوت باشد. چه اتفاقی می افتد؟

```
interface I1 { void f(); }  
interface I2 { int f(int i); }  
interface I3 { int f(); }  
class C implements I1 , I2 , I3{  
//Name Collisions and compile error  
}
```

# تطابق با interface

- یکی از استفاده های معمول از interface، الگوی طراحی Strategy است. از این طریق متدی را می نویسیم که می تواند اعمال خاصی را انجام دهد و interface ای داشته باشد که از طریق ما مشخص می شود.
- می توان گفت: “می توانید از متد من با هر object ای که دوست دارید استفاده کنید. البته تا زمانی که object شما با interface من مطابقت دارد.” این روش متدها را انعطاف پذیرتر، عمومی تر می کند و متدها دارای قابلیت استفاده مجدد خواهند بود.

# فیلدها در Interface

از آنجا که فیلدهایی که در interface قرار می دهیم به صورت خودکار static و final هستند، interface ابزار مناسبی برای ایجاد گروه هایی با مقادیر constant خواهد بود.

فیلدی که در interface تعریف می شود نمی تواند به blank field باشد. اما می توان آن ها را با عبارات غیر ثابتی مقدار دهی اولیه کرد.


```
public interface Months {  
    int JANUARY = 1 , FEBRUARY = 2, MARCH = 3,  
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,  
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,  
        NOVEMBER = 11, DECEMBER = 12;  
}
```

# Interface های تو در تو

Interface ها می توانند به صورت تو در تو در کلاس و یا در داخل interface دیگری جای گیرند

```
class A {  
    interface B { void f(); }  
    public class BImp implements B {  
        public void f() {}  
    }  
    private class BImp2 implements B {  
        public void f() {}  
    }  
}
```

# factory و Interface

Interface در واقع به عنوان دروازه ای به پیاده سازی های مختلف است و یک راه 

مرسوم برای ایجاد object های متناسب با interface استفاده از الگوی طراحی Factory

Method. می باشد

به جای آنکه مستقیماً یک constructor را فراخوانی کنید، می توانید متدی را با 

Factory object فراخوانی کنید که یک پیاده سازی برای interface ایجاد می کند.

در تئوری، کد شما به صورت کاملاً جدا از پیاده سازی interface قرار دارد. 