انجمن جاوا کاپ تقدیم می‌کند

دوره برنامه‌نویسی جاوا

واسط
Interface

صادق علی‌اکبری

حقوق مؤلف

- کلیه حقوق این اثر متعلق به انجمن جاواکاپ است
- بازنشر یا تدریس آن‌چه توسط جاواکاپ و به صورت عمومی منتشر شده است، با ذکر مرجع (جاواکاپ) بلامانع است
- اگر این اثر توسط جاواکاپ به صورت عمومی منتشر نشده است و به صورت اختصاصی در اختیار شما یا شرکت شما قرار گرفته، بازنشر آن مجاز نیست
- تغییر محتوای این اثر بدون اطلاع و تأیید انجمن جاواکاپ مجاز نیست

سرفصل مطالب

- واسط (Interface)
- کاربرد واسط در طراحی نرم‌افزار
- واسط در نمودار UML
- وراثت چندگانه (Multiple Inheritance)
- کلاس داخلی (Inner Class)
  - کلاس داخلی بی‌نام (Anonymous Inner Class)

واسط (interface)

یادآوری: کلاس‌ها و متدهای انتزاعی (abstract)

- **متد انتزاعی**: متدی که برای همه اشیاء یک کلاس وجود دارد،

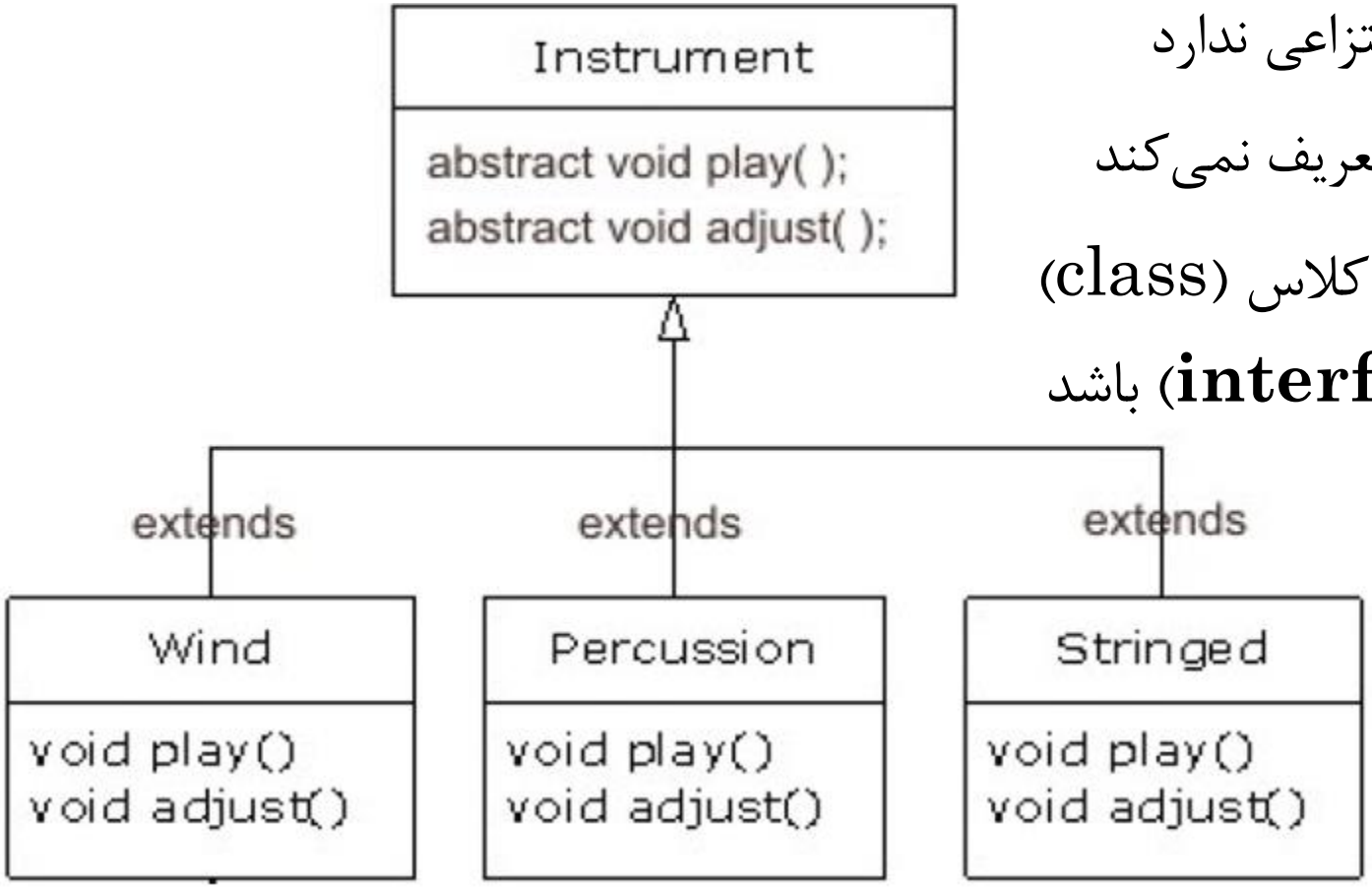
  اما پیاده‌سازی این متد در آن کلاس ممکن نیست

  و باید در هر زیرکلاس پیاده‌سازی شود

- مثل متد getArea در کلاس Shape
- کلاس انتزاعی: کلاسی که قرار نیست شیئی مستقیماً از این نوع باشد
  - هر شیء از این نوع، نمونه‌ای از یکی از زیرکلاس‌های این نوع خواهد بود
- گاهی همه متدهای یک کلاس انتزاعی هستند
  - و هیچ ویژگی (Field) مشترکی در این اَبَرکلاس تعریف نمی‌شود
- چنین کلاسی عملاً یک واسط (interface) از عملکرد و رفتارهای زیرکلاس‌ها است

مثال

- کلاس Instrument یک کلاس **کاملاً انتزاعی** (pure abstract) است
  - هیچ متد غیر انتزاعی ندارد
  - ویژگی خاصی تعریف نمی‌کند
- بهتر است به جای کلاس (class) یک **واسط** (**interface**) باشد



واسط (interface)

Public Interface
Hidden Implementation

- مفهوم عام واسط را قبلاً دیده بودیم:
  - واسط شیء: مجموعه عملکرد و رفتارهای عمومی شیء
    که برخلاف پیاده‌سازی پنهان، قابل استفاده و فراخوانی است
- **معنای خاص واسط (کلیدواژه interface در جاوا)**
  - واسط مانند یک کلاس کاملاً انتزاعی است (pure abstract clas)
  - هر متدی که در واسط (inerface) تعریف شود:
    به صورت ضمنی: عمومی (public) و انتزاعی (abstract) است
  - هر متغیری که در واسط تعریف شود:
    به صورت ضمنی: عمومی، استاتیک و ثابت (final) است
- واسط نشان می‌دهد که شیء چه رفتارها و عملکردی دارد
  - ولی نحوه اجرای این رفتارها را توصیف نمی‌کند (پیاده‌سازی ندارد)

مثال:

```
public interface Shape {
   double getArea();
   double getPerimeter();
}
```

- معنا و کاربرد این واسط تقریباً مشابه این کلاس انتزاعی است:

```
public abstract class Shapes {
   public abstract double getArea();
   public abstract double getPerimeter();
}
```

- مثل کلاس انتزاعی: ایجاد نمونه (شیء) از واسط ممکن نیست
  - عملگر new قابل اجرا روی یک واسط نیست
- البته تفاوت‌هایی هم در عمل وجود دارد (که خواهیم دید)

پیاده‌سازی واسط

- برای ارث‌بری از یک کلاس، از کلیدواژه **extends** استفاده می‌شود
- برای ارث‌بری یک کلاس از یک واسط، از کلیدواژه **implements** استفاده می‌شود

```
class Rectangle implements Shape{...}
```

- بین یک کلاس و واسطی که پیاده‌سازی کرده، رابطه is-a برقرار است
  - Rectangle **is a** Shape
- اگر کلاسی یک واسط را پیاده‌سازی کند: باید همه متدهای آن را هم تعریف کند
- وگرنه این کلاس، متدهای انتزاعی را به ارث برده و خودش هم باید انتزاعی شود
- مثال: این کلاس باید به صورت انتزاعی تعریف شود

```
abstract class Rectangle implements Shape{
  public double getArea() { return ...}
}
```

زیرا متد انتزاعی getPerimiter را پیاده‌سازی نکرده است

مثال

```
class Rectangle implements Shape{
    private double width, length;
    private int color;
    public Rectangle(double width, double length
            , int color) {
        this.width = width;
        this.length = length;
        this.color = color;
    }
    public int getColor() {
        return color;
    }
    public double getArea() {
        return width * length;
    }
    public double getPerimeter() {
        return 2*(width + length);
    }
}
```

```
public interface Shape {
  double getArea();
  double getPerimeter();
}
```

ارث‌بری واسط از واسط

- یک واسط می‌تواند از واسط (یا واسط‌های) دیگری ارث‌بری کند
  - همه متدهای (انتزاعی) اَبَرواسط به زیرواسط به ارث می‌رسند
  - این کار هم با کلیدواژه extends انجام می‌شود
  - رابطه is a برقرار خواهد بود
- یک واسط نمی‌تواند از یک کلاس ارث‌بری کند (غیرمجاز: خطای کامپایل)

```
interface CanRun{
  void run();
}
```

```
interface CanThink{
  void think();
}
```

```
interface CanTalk extends CanThink{
  void talk();
}
```

```
interface Human extends CanRun, CanTalk{
  void think();
}
```

خلاصه: وراثت و واسط

- یک کلاس، فقط و فقط از یک کلاس می‌تواند ارث‌بری کند (extends)
  - هر کلاس، اَبَرکلاس مشخص دارد که با کلیدواژه extends مشخص می‌شود
  - وگرنه (اگر اَبَرکلاس تصریح نشود) زیرکلاس Object خواهد بود
- یک کلاس هیچ و یا چند واسط را پیاده‌سازی می‌کند (implements)
- یک واسط از هیچ و یا چند واسط ارث‌بری می‌کند (implements)

```
interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}
```

```
class ActionCharacter {
    private String name;
    public String getName() {
        return name;
    }
}
```

مثال

```
class Hero
    extends ActionCharacter
    implements CanFight, CanSwim, CanFly {

    public void swim() {
    }

    public void fly() {
    }

    public void fight() {
    }
}
```

کوییز

سؤال

```
interface CanFight {
    void fight();
    void move();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
    void move();
}

class ActionCharacter {
    public void fight() {
    }
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
```

- کلاس ActionCharacter باید چه متدهایی را پیاده‌سازی کند تا انتزاعی نباشد؟

پاسخ:

move, fly, swim

نکات تکمیلی درباره واسط‌ها

تضاد اسامی

```
interface A{
  int f();
}
interface B{
  int f();
}
abstract class C implements A,B{ }
```

```
interface A{
  void f();
}
interface B{
  int f();
}
abstract class C implements A,B{ }
```

The return types are incompatible for the inherited methods A.f(), B.f()

چرا در جاوا ارث‌بری چندگانه برای کلاس‌ها پشتیبانی نمی‌شود؟

- چرا یک کلاس نمی‌تواند فرزند (زیرکلاس) چند کلاس باشد؟
  - طراحی پیچیده می‌شود (فهم آن هم سخت می‌شود)
  - وجود ویژگی‌های هم‌نام در اَبَرکلاس‌ها مشکل‌ساز می‌شود
    - **Multiple Inheritance of State**
  - وجود متدهایی که در چند اَبَرکلاس پیاده‌سازی شده‌اند، مشکل‌ساز می‌شود
    - **Multiple Inheritance of Behavior**

*نکته: این شکل از وراثت، به نوعی در **جاوا ۸** ممکن شده است*

- چرا ارث‌بری چندگانه برای واسط‌ها پشتیبانی می‌شود؟
  - مشکلات کلاس در واسط وجود ندارد
  - در واسط متدها پیاده‌سازی نمی‌شوند و ویژگی (Property) وجود ندارد
  - وراثت چندگانه از واسط‌ها، کاربردهای بسیار مهمی دارد
    - **Multiple Inheritance of Type**

واسط: متغیرها و سازنده‌ها

- تعریف متغیر در یک واسط رایج نیست
- در صورت تعریف متغیر:
  - متغیرها به طور ضمنی ثابت، استاتیک و عمومی (public) خواهند بود
  - مثال:

```
interface Humans{
 int MAX_AGE=150; ≈ public static final int MAX_AGE=150;
}
```

- خلاصه: واسط، وضعیت و حالت (state) اشیاءش را توصیف نمی‌کند
- امکان تعریف سازنده (constructor) در واسط وجود ندارد (چرا؟)
  - هدف سازنده، مقداردهی اولیه ویژگی‌های شیء است (Field یا Property)
  - سازنده: حالت (وضعیت) اولیه شیء را آماده می‌کند
  - اما ویژگی خاصی در واسط تعریف نمی‌شود (واسط، حالت شیء را توصیف نمی‌کند)

مثال

```
interface A {
     int a = 5;
     int f();
}
public class TestClass implements A{
     public static void main(String[] args) {
          System.out.println(a);
     }

...

}
```

کوییز

سؤال

همه دستورات زیر خطای کامپایل ایجاد می‌کنند:

```
c = d;
d = c;
b = d;
d = b;
a.f();
```

همه دستورات زیر بدون خطا کامپایل می‌شوند:

```
d = new D();
d = new E();
c= new E();
b = new E();
a = b;
b.f();
```

فرض کنید این ارجاع‌ها تعریف شده‌اند:

```
A a;
B b;
C c;
D d;
```

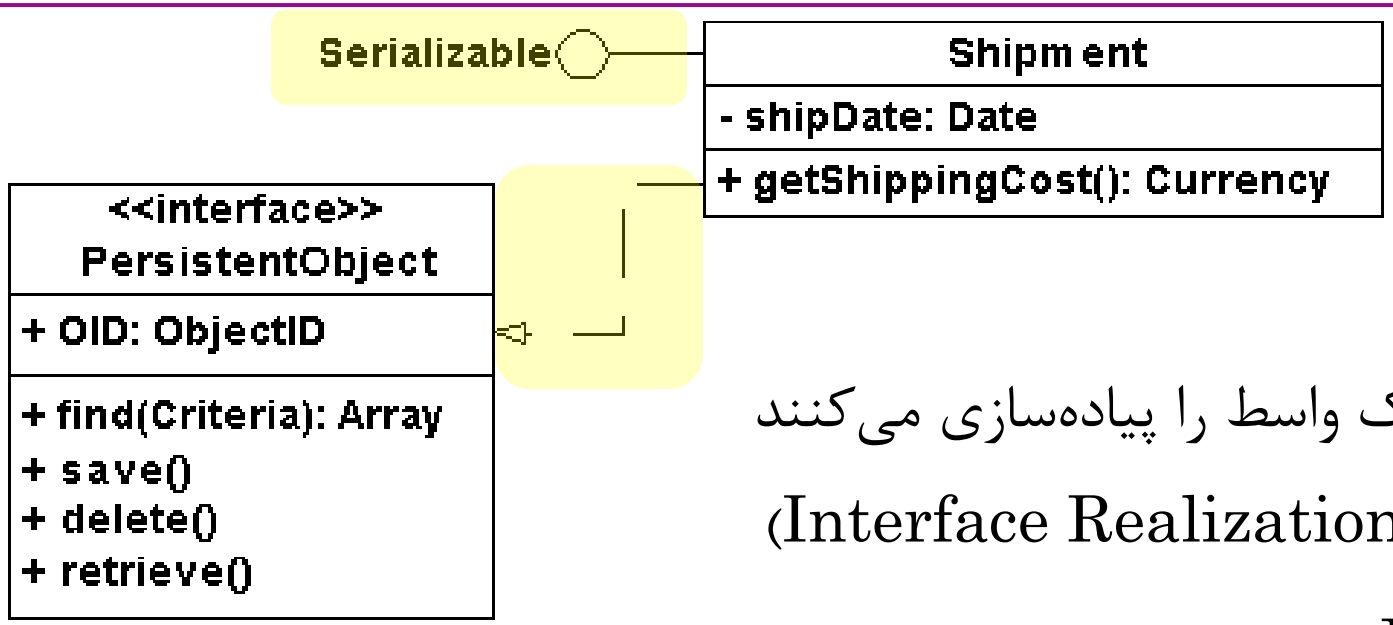
- به نظر شما از بین A،B،C،D و E کدام‌ها کلاس و کدام‌ها واسط هستند؟

کاربرد واسط در طراحی

کاربرد واسط

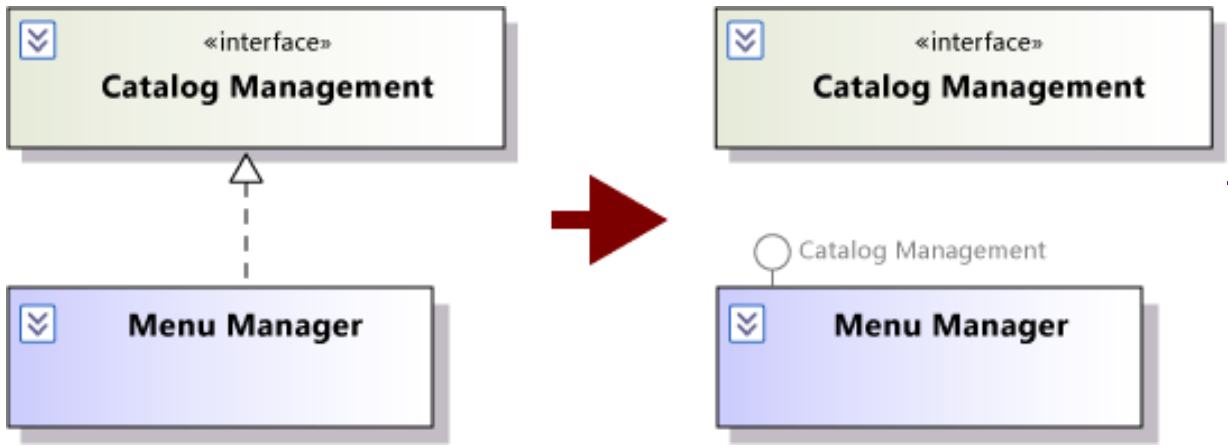
- واسط: کلاس مجرد خالص (کاملاً انتزاعی): pure abstract
- واسط درگیر جزئیات پیاده‌سازی نمی‌شود و کلیات رفتار شیء را توصیف می‌کند
- از واسط می‌توانیم به عنوان «توصیف‌کننده طراحی» استفاده کنیم
  - مثلاً طراح سیستم واسط‌ها را طراحی کند و برنامه‌نویس آن‌ها را پیاده‌سازی کند
  - مثلاً برای کدی که نوشتیم و در اختیار کاربران عمومی قرار می‌دهیم:
    فقط واسط را توضیح دهیم (کاربران نحوه کار کلاس را خواهند فهمید)
- با کمک واسط به صورت قدرتمند از ارث‌بری و چندریختی بهره می‌گیریم
- بهتر است حتی‌الامکان طراحی کلاس‌ها و متدهای ما به واسط‌ها وابسته باشند
  - مثلاً پارامتر یک متد، بهتر است واسط List باشد و نه زیرکلاس ArrayList
  - زیرا وابستگی به یک زیرکلاس خاص، تغییر و نگهداری برنامه را پرهزینه‌تر می‌کند

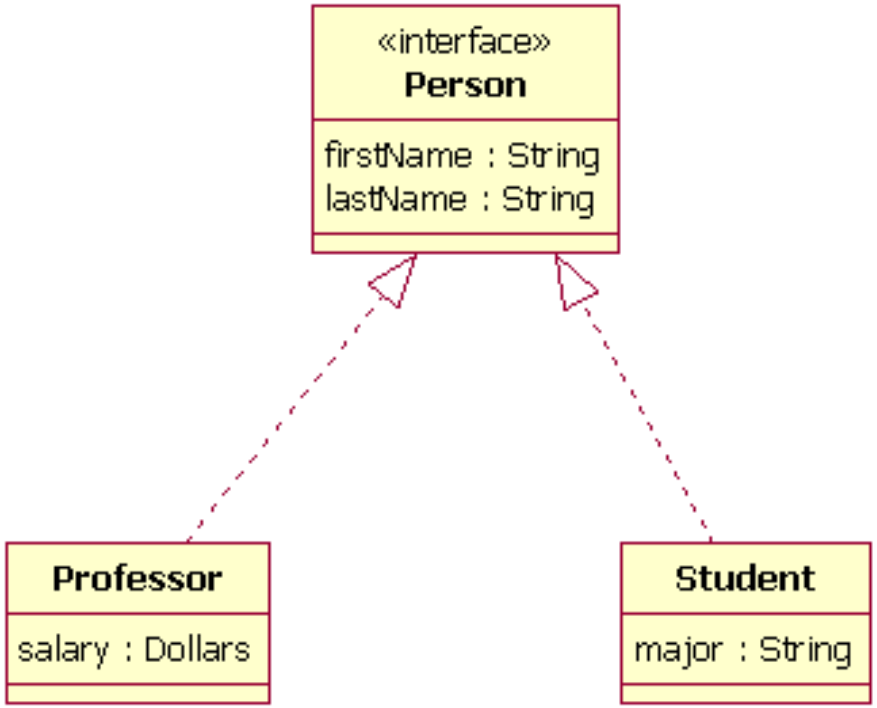
نمایش واسط در UML

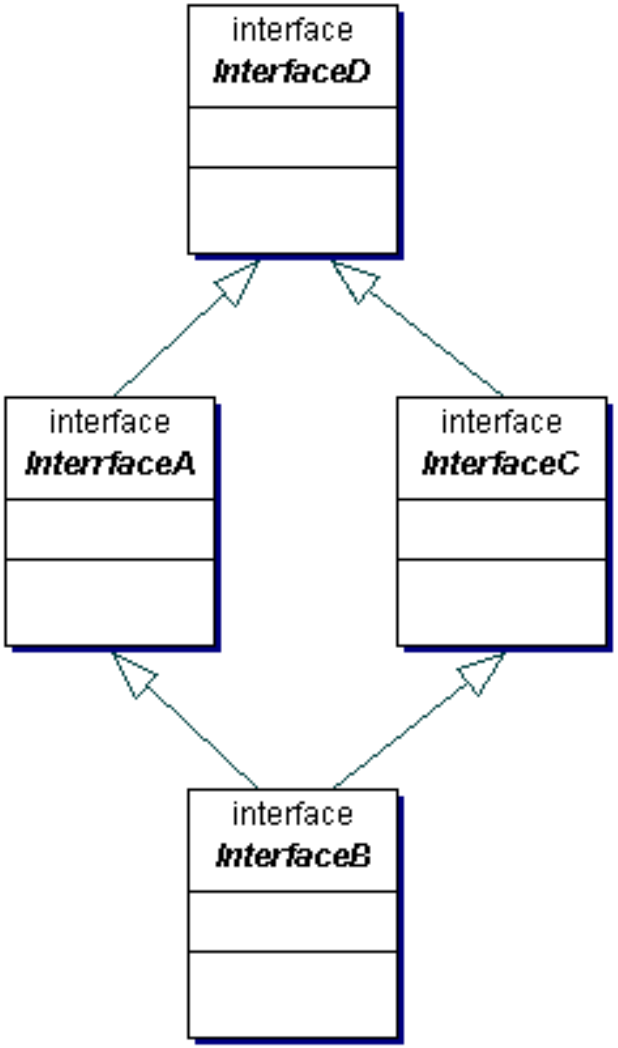


- کلاس‌هایی که یک واسط را پیاده‌سازی می‌کنند
  - تحقق واسط (Interface Realization)
- واسط در UML به دو شکل قابل نمایش است (هر دو شکل صحیح است) :
  - واسط‌های ساده‌ای که متدهای مهمی ندارند : مثل واسط Serializable
    - این‌گونه واسط‌ها اصطلاحاً فقط یک پروتکل تعریف می‌کنند
    - مثال: فقط اشیائی قابل ذخیره در فایل هستند که واسط Serializable را پیاده کنند
  - واسط‌هایی که متدهای خاصی دارند: مثل واسط PersistentObject

مثال


«interface»
Catalog Management
Menu Manager
«interface»
Catalog Management
Catalog Management
Menu Manager


«interface»
Person
firstName : String
lastName : String
Professor
salary : Dollars
Student
major : String


interface
InterfaceD
interface
InterfaceA
interface
InterfaceC
interface
InterfaceB

جاوا ۸ و متدهای پیش‌فرض برای واسط‌ها

- **از جاوا ۸ به بعد**: یک واسط می‌تواند متدهای غیرانتزاعی داشته باشد
- به این متدها، **متد پیش‌فرض** (**Default Method**) گفته می‌شود.
- مثال:

```
interface Person {
  Date getBirthDate();
  default Integer age(){
    long diff = new Date().getTime()-getBirthDate().getTime();
    return (int) (diff / (1000L*60*60*24*365));
  }
}
```

- همچنان یک کلاس می‌تواند چند واسط را پیاده‌سازی کند
- بنابراین امکان وراثت چندگانه در جاوا ۸ (به شکلی محدود) وجود دارد
- *بعداً به صورت مستقل در این زمینه (امکانات جاوا ۸) صحبت خواهیم کرد*

کلاس داخلی

کلاس داخلی

- در جاوا می‌توانیم کلاسی را درون کلاس دیگر تعریف کنیم: Inner Class
  - در کنار متدها و متغیرهای درون کلاس: کلاس‌های داخلی تعریف کنیم

```
public class OuterClass {
    private int value;
    ...
    public class Inner {
        public void f(){...}
        ...
    }
}
```

Inner Class

کلاس داخلی (Inner Classes)

- درون کلاسی دیگر تعریف می‌شود
- تعیین سطح دسترسی برای کلاس داخلی ممکن است
  - public, protected, package access, private
  - کلاس داخلی public، بیرون از کلاس بیرونی (کلاس دربرگیرنده) قابل استفاده است
  - کلاس داخلی private، فقط داخل همین کلاس بیرونی قابل استفاده است
- کلاس داخلی می‌تواند final باشد
  - در این صورت قابل ارث‌بری نخواهد بود
- کلاس داخلی می‌تواند استاتیک باشد یا نباشد
  - کلاس عادی نمی‌تواند استاتیک باشد. استاتیک فقط برای کلاس داخلی معنی دارد

کلاس داخلی عادی (غیراستاتیک)

- در یک کلاس داخلی عادی:
  - یک ارجاع ضمنی (پنهان) به شیئی از کلاس بیرونی وجود دارد
  - اگر اسم کلاس بیرونی `OuterClass` باشد:
    این ارجاع ضمنی با `OuterClass.this` در کلاس داخلی قابل استفاده است
  - (مثل `this` که ارجاعی به شیء جاری از همین کلاس است)
- با کمک این ارجاع ضمنی (وحتی بدون ذکر و تصریح آن) :
  - در کلاس داخلی به ویژگی‌ها و متدهای کلاس بیرونی (کلاس دربرگیرنده) دسترسی داریم
- بنابراین برای ایجاد شیئی از کلاس داخلی، نیازمند شیئی از کلاس بیرونی هستیم
  - نمونه‌سازی از کلاس داخلی با کمک یک شیء مرجع از کلاس بیرونی انجام می‌شود

مثال

```
public class OuterClass {
 private int value = 2;
 class Inner{
  public void innerMethod(){
   OuterClass.this.value = 5;
  }
 }
 public void outerMethod(){
  Inner inner = new Inner();
  inner.innerMethod();
 }
 public static void main(String[] a){
  OuterClass outer = new OuterClass();
  System.out.println(outer.value);
  outer.outerMethod();
  System.out.println(outer.value);
 }
}
```

این نمونه، در یک متد غیراستاتیک در کلاس بیرونی ساخته شده

یک ارجاع به شیء بیرونی در این متد وجود دارد

این ارجاع به صورت ضمنی (پنهان) به شیء کلاس داخلی پاس می‌شود

این ارجاع با عنوان OuterClass.this در کلاس داخلی قابل استفاده است

مثال دیگر

```
public class OuterClass {
  private int value  = 2;
  class Inner{
    public void f(){
       OuterClass.this.value = 5;
    }
  }
  public static void main(String[] args)  {
    OuterClass outer = new OuterClass();
    OuterClass.Inner inner = outer.new Inner();
    System.out.println(outer.value);
    inner.f();
    System.out.println(outer.value);
  }
}
```

این نمونه، در یک متد غیراستاتیک در کلاس بیرونی ساخته شده

یک ارجاع به شیء بیرونی در این متد وجود دارد

این ارجاع به صورت ضمنی (پنهان) به شیء کلاس داخلی پاس می‌شود

این ارجاع با عنوان OuterClass.this در کلاس داخلی قابل استفاده است

کلاس داخلی استاتیک

- به ارجاعی به شیئی از کلاس بیرونی دسترسی ندارد
  - مثل متد استاتیک که به this دسترسی ندارد،
    کلاس داخلی به outer.this دسترسی ندارد
- بنابراین هنگام نمونه‌سازی هم به شیئی از کلاس بیرونی نیازی نیست

```
class OuterClass {
 static class Inner{
  public void f(){…}
 }
}
```

```
public class MainClass {
 public static void main(String[] args)  {
  OuterClass.Inner in = new OuterClass.Inner();
  in.f();
 }
}
```

کلاس داخلی بی‌نام
Anonymous Inner Class

کلاس داخلی بی‌نام

- (یکی از پرکاربردترین شکل‌های استفاده از کلاس‌های داخلی)
- گاهی یک کلاس را می‌سازیم تا فقط یک شیء از آن ایجاد کنیم
  - این کلاس معمولاً از کلاس یا واسط خاصی ارث‌بری می‌کند: رفتار خاصی را پیاده کرده
- در این صورت، می‌توانیم ایجاد کلاس را خلاصه کنیم
- در محلی که به شیء جدید نیاز داریم، کلاس را هم تعریف کنیم
- چنین کلاسی یک بار مصرف است: نام ندارد

```
interface Protocol{
  void behavior();
}
```

```
Protocol inner = new Protocol() {
    public void behavior() {
      value = 5;
    }
};
inner.behavior();
```

کلاس بی‌نام یک کلاس داخلی است

```
interface Protocol{
  void behavior();
}
```

```
public class OuterClass {
  private int value  = 2;
   public void outerMethod(){
   Protocol inner = new Protocol() {
     public void behavior() {
       value = 5;
     }
   };
   inner.behavior();
}
```

معادل این است:

```
OuterClass.this.value = 5;
```

- به ویژگی‌ها و متدهای کلاس بیرونی دسترسی دارد
- معمولاً یک واسط را پیاده‌سازی می‌کند (و گاهی یک کلاس که معمولاً انتزاعی است)
  - و بعضی از متدهای این واسط یا کلاس را Override می کند

کاربرد رایج کلاس داخلی بی‌نام

```
public class MyFrame extends JFrame{
   JButton button;
   JTextField textbox;
   public MyFrame(){
        button = new JButton("Click!");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
              showMessageDialog(null, textbox.getText());
            }
        });
        add(button);
        textbox = new JTextField(10);
        add(textbox);
   }
}
```

چند نکته درباره کلاس داخلی بی‌نام

- در بسیاری از موارد، هدف از ایجاد کلاس بی‌نام، تعریف یک متد است
  - توصیف یک رفتار
  - مثلاً وقتی این دکمه زده شد، چه رفتاری انجام شود؟
- برای این منظور، زبان‌های برنامه‌نویسی مختلف، امکانات دیگری ارائه می‌کنند
  - اشاره‌گر به تابع (Pointer to Function)
  - مفهوم Delegate
- امکانات جدیدی هم در جاوا ۸ ارائه شده است
  - عبارت لامبدا (Lambda Expression)
  - ارجاع به متد (Method Reference)

کوییز

خروجی این برنامه چیست؟

```
abstract class Protocol{
   public abstract void f();
}
public class OuterClass {
  private int value = 2;
  public void outerMethod() {
    Protocol inner = new Protocol() {
      public void f() {
        OuterClass.this.value = 5;
      }
    };
    inner.f();
  }
  public static void main(String[] args) {
    OuterClass outer = new OuterClass();
    System.out.println(outer.value);
    outer.outerMethod();
    System.out.println(outer.value);
  }
}
```

پاسخ:

2

5

تمرین عملی

تمرین عملی

- تعریف یک واسط: CanMove
  - تعریف زیرواسط‌های CanRun و CanSwim و CanFly
  - تعریف زیرواسط دوزیست (Amphibious): CanSwim & CanMove
  - کلاس Bird و کلاس Airplane (که CanFly) را پیاده می‌کنند
    - ابتدا زیرکلاسی متدهای واسط را پیاده‌سازی نمی‌کند: انتزاعی می‌شود
- عدم امکان ایجاد شیء (نمونه‌سازی) از واسط
- ایجاد شیء با کمک زیرکلاس بی‌نام

جمع‌بندی

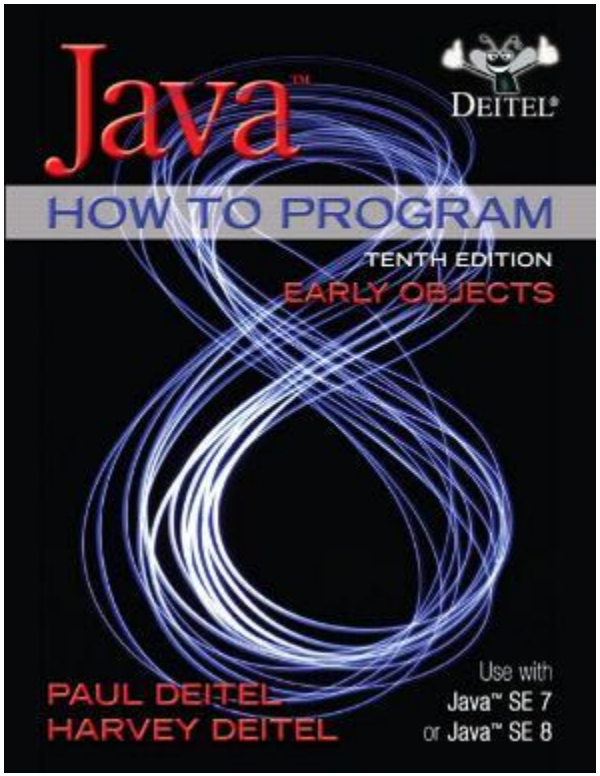
جمع‌بندی

- واسط
- کاربرد واسط در طراحی
- نمایش واسط در UML Class Diagram
- وراثت چندگانه
- کلاس داخلی و کلاس داخلی بی‌نام

مطالعه کنید

- بخشی از فصل ۱۰ کتاب دایتل

Java How to Program (Deitel & Deitel)



10- Object-Oriented Programming: Polymorphism and **Interfaces**

- تمرین‌های همین فصل از کتاب دایتل

تمرین

- واسط‌های زیر را تعریف کنید:
  - Nameable (شامل متد getName)
  - CanThinkg (شامل متد think)
  - CanTalk که زیرواسط CanThink است (شامل متد talk)
- کلاس انتزاعی NamedObject را تعریف کنید
  - ویژگی name و getter/setter های متناظر دارد
- کلاس Person را تعریف کنید
  - این کلاس Nameable و CanTalk و NamedObject است
- زیرکلاس Student را تعریف کنید
- مثال

```
CanTalk c = new Person();
c.talk();
```

```
Nameable n = new Person();
String r = n.getName();
```

```
NamedObject no = new Student();
System.out.println(no.getName());
```

جستجو کنید و بخوانید

- محدودیت‌های کلاس داخلی
- واسط‌های مهم زبان جاوا
  - Serializable, AutoCloseable, Runnable, ...
- امکانات جدید در جاوا ۸
  - واسط تابعی
  - عبارت لامبدا (Lambda Expression)
  - ارجاع به متد
  - متد پیش‌فرض (Default Method)
  - امکان وراثت چندگانه از چند واسط (که متد پیش‌فرض دارند)

پایان

سایر مطالب

تاریخچه تغییرات

نسخه	تاریخ	توضیح
۱.۰.۰	۱۳۹۴/۳/۲۸	نسخه اولیه ارائه آماده شد