Java高级特征

孙聪

网络与信息安全学院

2019-09-30

课程内容

- Java概述
- 面向对象程序设计概念
- Java语言基础
- Java面向对象特性
- Java高级特征
- 容器类
- 常用预定义类
- 异常处理
- 输入输出
- 线程

提要

提要

静态变量(类变量)

静态变量

在类的成员变量声明中带有static关键字的变量

静态变量(类变量)

- 该类的所有的对象实例之间共享使用方法区中的静态变量

```
class Employee{
    private int id;
    public static int serialNum = 1; //静态成员变量
    Employee(){ id=serialNum ++; }
    public static void main(String[] args){
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
    }
}
```

静态变量(类变量)

- 该类的所有的对象实例之间共享使用方法区中的静态变量

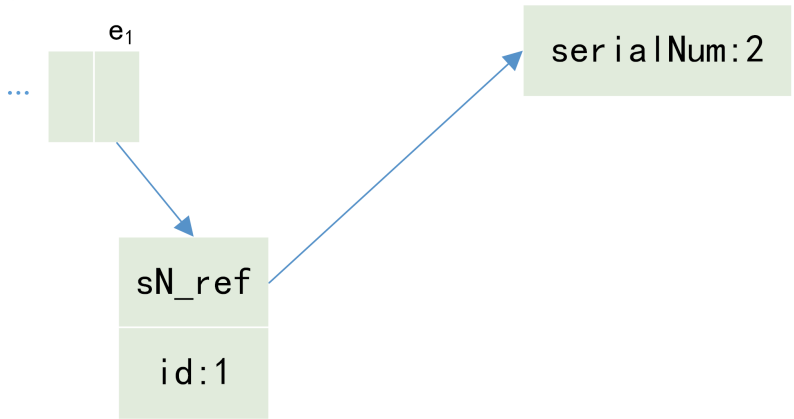
```
class Employee{
    private int id;
    public static int serialNum = 1; //静态成员变量
    Employee(){ id=serialNum ++; }
    public static void main(String[] args){
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
    }
}
```

```
serialNum:1
```

静态变量(类变量)

- 该类的所有的对象实例之间共享使用方法区中的静态变量

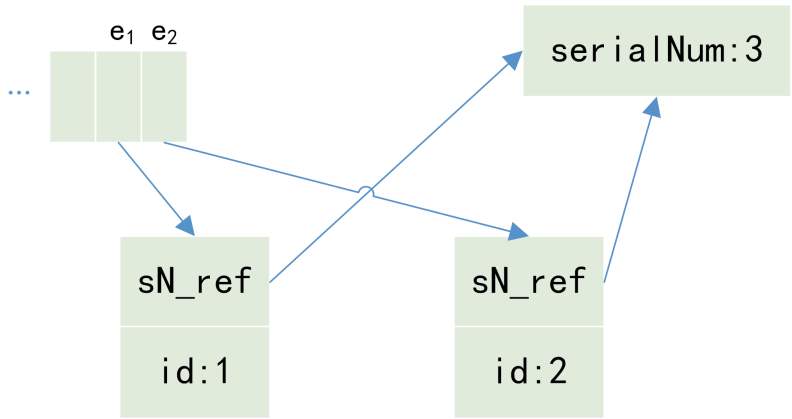
```
class Employee{
    private int id;
    public static int serialNum = 1; //静态成员变量
    Employee(){ id=serialNum ++; }
    public static void main(String[] args){
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
    }
}
```



静态变量(类变量)

- 该类的所有的对象实例之间共享使用方法区中的静态变量

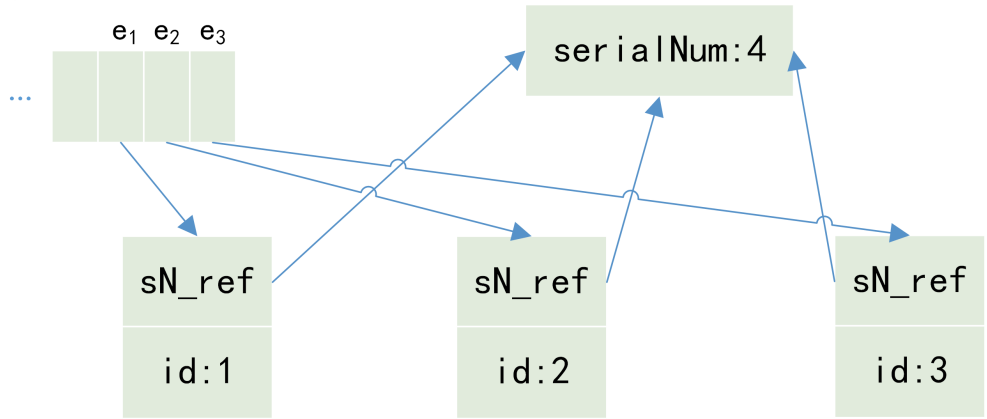
```
class Employee{
    private int id;
    public static int serialNum = 1; //静态成员变量
    Employee(){ id=serialNum ++; }
    public static void main(String[] args){
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
    }
}
```



静态变量(类变量)

- 该类的所有的对象实例之间共享使用方法区中的静态变量

```
class Employee{
    private int id;
    public static int serialNum = 1; //静态成员变量
    Employee(){ id=serialNum ++; }
    public static void main(String[] args){
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
    }
}
```



静态变量的创建

- 静态变量的创建与实例对象无关
- 只在系统加载其所在类时分配空间并初始化，且在创建该类的实例对象时不再分配空间
  - 什么时候加载其所在类？
    - 运行到不得不加载该类的时候
  - 什么是“不得不加载该类的时候”？
    - 即将创建该类的第一个对象时
    - 首次使用该类的静态方法或静态变量时
    - 加载一个类之前要先加载其父类
- 例：StaticInitialization.java

静态变量的访问

- 非private的静态变量，可在类外用类名访问

```
class Employee{
    private int id;
    public static int serialNum = 1;
    Employee(){ id=serialNum ++; }
}
class OtherClass{
    public static void main(String[] args){
        System.out.println(Employee.serialNum);
        // Employee e=new Employee();
        // System.out.println(e.serialNum);
    }
}
```

静态方法(类方法)

静态方法

在类的成员方法声明中带有static关键字的方法

```
class GeneralFunction {
    public static int add(int x, int y) { return x + y; }
}
public class UseGeneral {
    public static void main(String[] args) {
        int c = GeneralFunction.add(9, 10);
        System.out.println("9 + 10 = " + c);
    }
}
```

- main方法是静态方法、程序入口点，JVM不创建实例对象就可以执行该方法
- 任何类的构造方法实际上都是静态方法，当首次创建一个类的对象实例时，如果类还没有被装载，就会首先装载该类，然后再进行对象的实例化

静态方法(类方法)

- 静态方法中没有this引用，因此静态方法不能直接调用实例方法，也不能直接访问所属类的实例成员变量

```
public class TestStaticMethod{
    public static void main(String[] args){
        StaticMethod obj=new StaticMethod();
        StaticMethod.sPrintXAndY(obj);
    }
}
class StaticMethod{
    int x=0;
    static int y=1;
    public void iPrintAndIncreaseY(){
        sPrintY();
        y++;
    }
    public static void sPrintY(){
        //System.out.println(this.x);        //不能访问实例成员变量
        //iPrintAndIncreaseY();              //不能访问实例方法
        System.out.println(StaticMethod.y); //可以访问静态变量
    }
    public static void sPrintXAndY(StaticMethod o){
        System.out.println(o.x);            //可以通过o引用访问实例成员变量
        o.iPrintAndIncreaseY();             //可以通过o引用调用实例方法
        sPrintY();                          //可以直接调用静态方法
    }
}
```

静态方法的重写

- 回顾方法重写的规则：
  - 不改变方法的名称、参数列表和返回值，改变方法的内部实现
  - 子类中重写方法的访问权限不能缩小
  - 子类中重写方法不能抛出新的异常
  - 父类中private的成员，在子类中不能被隐藏（重写）
- 本节加入以下重写规则：
  - 子类不能把父类的静态方法重写为非静态
  - 子类不能把父类的非静态方法重写为静态
  - 子类可以声明与父类静态方法相同的方法
  - 静态方法的重写不会导致多态性
  - 构造方法本质上是static方法，不具有多态性

静态方法的重写

- 回顾方法重写的规则：
  - 不改变方法的名称、参数列表和返回值，改变方法的内部实现
  - 子类中重写方法的访问权限不能缩小
  - 子类中重写方法不能抛出新的异常
  - 父类中private的成员，在子类中不能被隐藏（重写）
- 本节加入以下重写规则：
  - 子类不能把父类的静态方法重写为非静态
  - 子类不能把父类的非静态方法重写为静态
  - 子类可以声明与父类静态方法相同的方法
  - 静态方法的重写不会导致多态性
  - 构造方法本质上是static方法，不具有多态性

```
public class StaticPolymorphism{
    public static void main(String[] args){
        Sup s=new Sub();
        s.mtd1();
        s.mtd2();                          //静态方法的重写不会导致多态性
    }
}
class Sup {
    public void mtd1() { System.out.println("Sup.instanceMethod"); }
    public static void mtd2() { System.out.println("Sup.staticMethod"); }
}
class Sub extends Sup{
    //public static void mtd1() {}        //静态方法不能隐藏实例方法
    //public void mtd2() {}               //实例方法不能重写静态方法
    public void mtd1(){ System.out.println("Sub.instanceMethod"); }
    public static void mtd2() { System.out.println("Sub.staticMethod"); }
}
```

静态初始化程序块

- 类定义中不属于任何方法体且以static关键字修饰的语句块
  static {...}

- 在加载该类时执行且只执行一次
- 如果类中定义了多个静态语句块，则这些语句块按照在类中出现的次序运行

```
public class TestStaticInit{
    public static void main(String[] args){
        System.out.println(StaticInit.j);
    }
}
class StaticInit{
    static int i;
    static int j=2;
    static { System.out.println(i+ “;” +j); }
    int k=4;
    static { i=3; }
    static { System.out.println(i+ “;” +j); }
}
```

例4-5：写出以下程序的输出

```
class T1 {
    static int s1 = 1;
    static { System.out.println("static block of T1: " + T2.s2); }
    T1() { System.out.println("T1(): " + s1); }
}
class T2 extends T1 {
    static int s2 = 2;
    static { System.out.println("static block of T2: " + T2.s2); }
    T2() { System.out.println("T2(): " + s2); }
}
public class InheritStaticInit {
    public static void main(String[] args) {
        new T2();
    }
}
```

提要

final的使用位置

- 在类声明中使用：表示类不能被继承
- 在成员方法声明及方法参数中使用：成员方法不能被重写，参数变量值不能更改
- 在成员变量和局部变量声明中使用：表示变量的值不能更改

在类声明中使用final关键字

- 被定义成final的类不能再派生子类
- 例

```
final class Employee { ... }
class Manager extends Employee { ... }  //错误
```

在成员方法声明中使用final关键字

- 被定义成final的方法不能被重写
- 将方法定义为final可使运行时的效率优化（早期）
  - 对于final方法，编译器直接产生调用方法的代码，而阻止运行时刻对方法调用的动态绑定
- private的方法都隐含指定为是final的，对子类不可见就无所谓被重写

在方法参数中使用final关键字

- 将方法参数指明为final，则无法在方法中更改参数的值

```
class Gizmo {
    public void spin() { }
}
public class FinalArg {
    void with(final Gizmo g) {
        // g=new Gizmo();        //错误,g是final的
    }
    void without(Gizmo g) {
        g = new Gizmo();
        g.spin();
    }
    int g(final int i) { return i + 1; }
    public static void main(String[] args) {
        FinalArg bf = new FinalArg();
        bf.without(null);
        bf.with(null);
    }
}
```

在成员变量中使用final关键字

- 被定义成final的成员变量，一旦被赋值就不能改变
  - 对于基本类型的成员变量：
    - 数值不变
    - 用来定义常量：声明final变量并同时赋初值
  - 对于引用类型的成员变量：
    - 引用不变，但被引用对象可被修改
    - 这一约定同样适用于数组

```
class Value { int i;
    public Value(int i) { this.i = i; }
}
public class FinalData {
    private static Random rand = new Random(47);
    private final int valueOne = 9;                    // 基本类型编译时常量
    private final int i4 = rand.nextInt(20);           // 运行时不可变,但非编译时常量
    private Value v1 = new Value(11);
    private final Value v2 = new Value(22);            // 引用类型常量
    private final int[] a = { 1, 2, 3, 4, 5, 6 };      // 数组类型常量
    public static void main(String[] args) {
        FinalData fd1 = new FinalData();
        //fd1.valueOne++; fd1.i4++;                    // valueOne, i4不能更改
        fd1.v1 = new Value(9);                         // v1不是final的, 可引用到新的对象
        // fd1.v2=new Value(0);                        // v2是final的,不能引用到新的对象
        fd1.v2.i++;                                    // v2引用不能更改,但对象本身可更改
        // fd1.a=new int[3];                           // 数组a为final,不能引用到新的数组
        for (int i = 0; i < fd1.a.length; i++) {
            fd1.a[i]++;                                // 数组a是final的,但数组元素不是final的
        }
    }
}
//又例: FinalVariables.java
```

在成员变量中使用final关键字

- 空白final：若final成员变量声明时未赋初值，则在所属类的每个构造方法中都必须对该变量赋值

```
public class BlankFinals{
    public static void main(String[] args){
        Car[] cars=new Car[]{new Car("BMW"), new Car("Rolls-Royce"),
                             new Car("Toyota")};
        for(Car c: cars){ c.Print(); }
    }
}
class Car{
    final int carID;              //空白final的基本类型成员
    final Brand brand;            //空白final的引用类型成员
    static int counter=10000;
    public Car(String s){         //构造方法中必须对carID和brand赋初值
        carID=counter++;
        brand=new Brand(s);
    }
    public void Print(){
        System.out.println(brand.brandName+" No." +carID);
    }
}
class Brand{
    String brandName;
    public Brand(String s){ brandName=s; }
}
```

在局部变量中使用final关键字

- 被定义成final的局部变量可以在所属方法的任何位置被赋值，但只能赋一次

提要

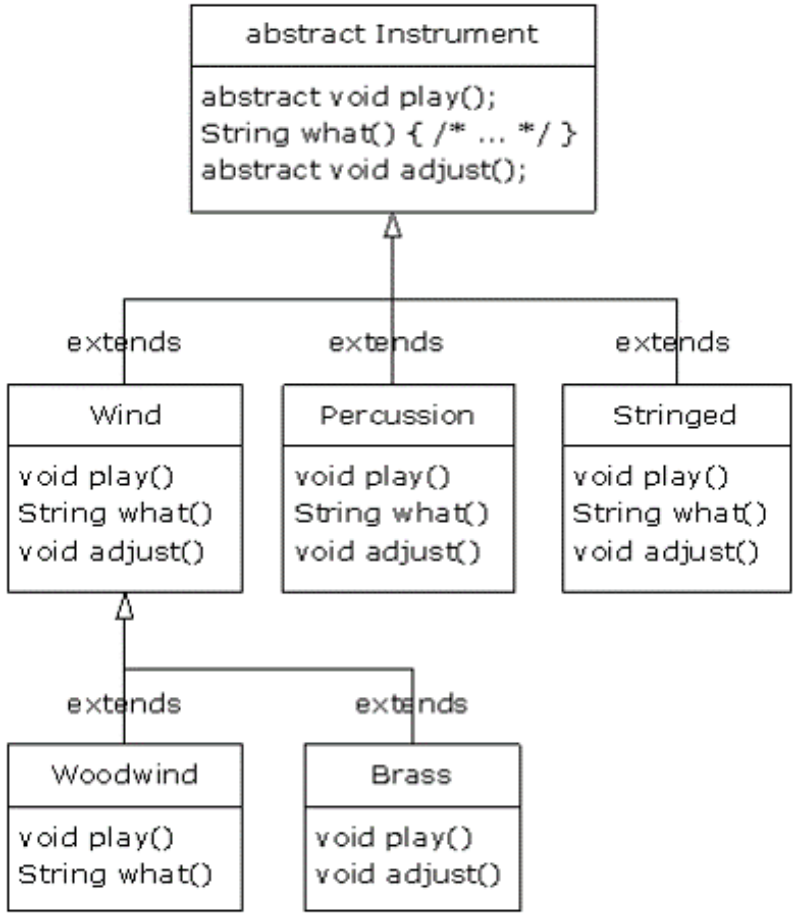
抽象方法

- 抽象方法：Java允许在类中只声明方法而不提供方法的实现，这种只有声明而没有方法体的方法称为抽象方法
- 抽象方法的声明中需加关键字abstract

抽象类

- 由关键字abstract声明的类称为抽象类
- 与抽象方法的关系
  - 包含抽象方法的类必须是抽象类
  - 抽象类可以不包含抽象方法（但一般会包括）
- 可以包含构造方法、一般成员变量和成员方法
- 核心特点：不能创建抽象类的实例
  - 若抽象类的子类实现了所有抽象方法，则可以创建子类的实例对象，否则该子类也是抽象类
- 抽象类的作用：为一类对象建立抽象的模型
- 既然不能创建实例，为何还需要构造方法？
  - 抽象类的构造方法通常声明为protected（只给子类用）

抽象类



- 又例：TestShapesAbsClass.java

接口

- 接口（Interface）：确定了对特定对象所能发出的请求，或者对象接收消息的方式
- 接口中只声明方法（“做什么”，抽象方法），但不定义方法体（“怎么做”）
  - 将“做什么”与“怎么做”分离
  - 只规定了类的基本形式，不涉及任何实现细节，实现了接口的类具有该接口规定的行为
  - 接口可看作使用类的“客户”代码与提供服务的类之间的契约或协议
- 抽象类是介于普通类与接口之间的中间状态

接口的定义

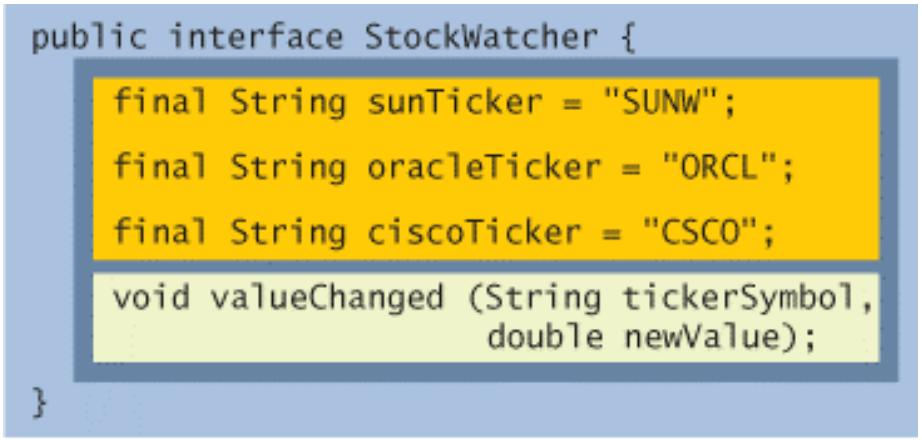
- 接口定义 = 接口声明 + 接口体

接口声明

- [public] interface 接口名 [extends 父接口列表] { 接口体 }
  - public / default: 任意类均可使用 / 与该接口在同一个包中的类可使用
  - 一个接口可以有多个父接口，子接口继承父接口的所有常量和方法

接口体

- 接口体=常量定义+方法定义



- 常量默认具有final，static属性
  - 类型 常量名=常量值;
- 方法默认具有public，abstract属性
  - 返回类型 方法名 ([参数列表]);

接口体

- 注意
  - 常量不能为空白final的
  - 在实现接口的类中，接口方法必须实现为public的（权限不能降低）
  - 接口中成员不能使用的修饰符：transient, volatile, synchronized, private, protected

接口的使用——用类实现接口

- 类声明中的implements关键字
- 类可以使用接口定义的常量
- 类必须实现接口定义的所有方法（否则为抽象类）
- 一个类可以实现多个接口，例：

```
interface  I1{ … }
interface  I2{ … }
class Sup { ... }
class C extends Sup implements I1,I2 { ... }
```

接口的使用——作为数据类型，支持多态

- 实现该接口的类可看作该接口的“子类”，接口类型的变量可指向该“子类”的实例

```
Interface Human{
    void showNameInNativeLanguage();
}
class Chinese implements Human{ ... }
class Japanese implements Human{ ... }
...
Human e1 = new Chinese( );
Human e2 = new Japanese( );
e1.showNameInNativeLanguage();
e2.showNameInNativeLanguage();
...
```

- 又例： TestShapesInterface.java

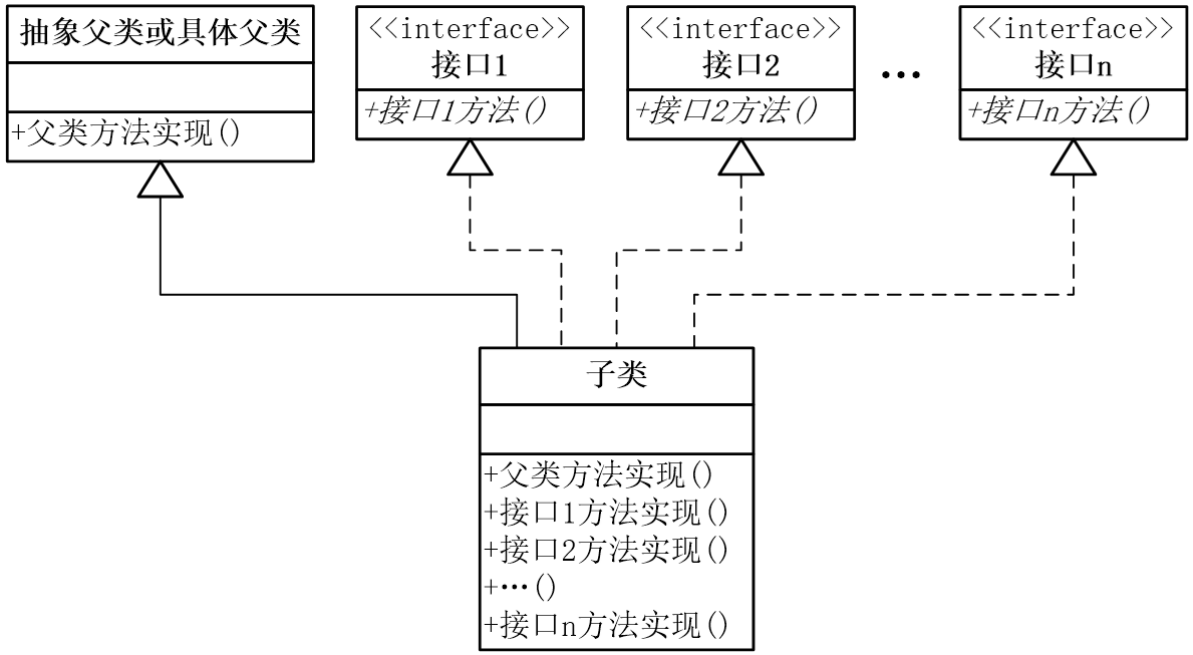
接口与多重继承

类继承与接口继承

- 类继承只支持单继承（子类从单个直接父类继承），不支持多重继承（子类从多个直接父类继承）
  - extends后的类名只能有一个
  - 类的继承关系形成树型层次结构
- 接口继承支持多重继承
  - 父接口中的常量可被子接口中的同名常量隐藏
  - 父接口中的方法可被子接口中的方法重写（没有意义）

接口与多重继承

- 类可以通过实现多个接口达到多重继承的效果
- 在进行单个父类与多个接口合并时，只有单个父类具有实现细节，从而避免代码冲突



```
public class Sportsman extends Person implements Runner, Swimmer, Jumper{
    public void run() { print ("Sportsman running"); }
    public void swim() { print("Sportsman swimming"); }
    public void jump() { print("Sportsman jumping"); }
    public static void toRun(Runner r) { r.run(); }
    public static void toSwim(Swimmer s) { s.swim(); }
    public static void toJump(Jumper j) { j.jump(); }
    public static void toEatAndDrink(Person p) { p.eat(); p.drink(); }
    public static void main(String[] args){
        Sportsman s=new Sportsman();
        toRun(s);
        toSwim(s);
        toJump(s);
        toEatAndDrink(s);
    }
}
class Person{
    public void eat(){ print("Person eating"); }
    void drink() { print("Person drinking"); }
}
interface Runner{
    void run();
    void eat();
}
interface Swimmer{ void swim(); }                    interface Jumper{ void jump(); }
```

接口与抽象类的区别

- 接口中的所有方法都是抽象的，而抽象类可以定义非抽象方法
- 一个类可以实现多个接口，但只能继承一个抽象父类
- 接口与实现它的类不构成类的继承体系，即接口不是类体系的一部分，不相关的类也可以实现相同的接口；抽象类属于类的继承体系，且一般位于类体系的较高层

通过继承扩展接口

- 直接向接口中扩展方法可能带来问题：所有实现原来接口的类将因为接口的改变而不能正常工作
- 不能向interface定义中随意增加方法，需要通过继承扩展接口

```
public interface StockWatcher {
    final String sunTicker = “SUNW”;
    final String oracleTicker = “ORCL”;
    void valueChanged(String tickerSymbol, double newValue);

}
```

通过继承扩展接口

- 直接向接口中扩展方法可能带来问题：所有实现原来接口的类将因为接口的改变而不能正常工作
- 不能向interface定义中随意增加方法，需要通过继承扩展接口

```
public interface StockWatcher {
    final String sunTicker = “SUNW”;
    final String oracleTicker = “ORCL”;
    void valueChanged(String tickerSymbol, double newValue);
    void currentValue(String tickerSymbol, double newValue); //不能直接添加
}
```

通过继承扩展接口

- 直接向接口中扩展方法可能带来问题：所有实现原来接口的类将因为接口的改变而不能正常工作
- 不能向interface定义中随意增加方法，需要通过继承扩展接口

```
public interface StockWatcher {
    final String sunTicker = “SUNW”;
    final String oracleTicker = “ORCL”;
    void valueChanged(String tickerSymbol, double newValue);

}
public interface StockTracker extends StockWatcher { //通过子接口进行扩展
    void currentValue(String tickerSymbol, double newValue);
}
```

下列接口的定义中，哪些是正确的？

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

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

- ```
  abstract interface Printable extends Interface1, Interface2{
      void print() {};
  }
  ```

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

提要

枚举类型

- 通过关键字enum将一组具名值的有限集合创建为一种类型
  - 这些具名值又称为枚举常量
- 一个枚举类型实际定义了一个类，该类可以包含方法和其他属性，以支持对枚举值的操作，还可以实现任意的接口
- 枚举类型变量属于引用变量，变量取值范围为所有可能的枚举常量，例如对于枚举类型

```
public enum Grade{
    FRESHMAN, SOPHOMORE, JUNIOR, SENIOR
}
```

枚举变量的定义

```
Grade grade = Grade.JUNIOR;
```

枚举类型

枚举类型的定义

```
[public] enum 枚举类型名 [implements 接口名表] {
    枚举常量定义
    [枚举体定义]
}
```

枚举类型

枚举类型的定义

```
[public] enum 枚举类型名 [implements 接口名表] {
    枚举常量定义
    [枚举体定义]
}
```

- 枚举声明：
  - public/default：可被包外类访问 / 只能在同一包中访问
  - 所有枚举类型都隐含继承java.lang.Enum类，故不能再继承其他任何类

枚举类型

枚举类型的定义

```
[public] enum 枚举类型名 [implements 接口名表] {
    枚举常量定义
    [枚举体定义]
}
```

- 枚举常量定义：常量1[,常量2[, … 常量n]…]] [;]
  - 如果没有枚举体部分，则“;”可省略
  - 枚举常量实际上是枚举类型的static和final的实例，加载枚举类型时，调用枚举类型的构造方法创建这些实例
  - 如果在枚举体中定义了带参构造方法，则在定义枚举常量时可采用“常量(参数1,参数2, …)”的形式

枚举类型

枚举类型的定义

```
[public] enum 枚举类型名 [implements 接口名表] {
    枚举常量定义
    [枚举体定义]
}
```

- 枚举体定义：
  - 可以包含变量、构造方法和成员方法
  - 构造方法只能为private，保证用户不会创建新的枚举常量

枚举类型

- 枚举类型的方法
  - 每个枚举类型都具有java.lang.Object类和java.lang.Comparable接口中可以被继承的方法
  - 编译器在创建枚举类型时也自动加入一些方法，如：
    - ElementType[] values()：返回一个数组，数组包含该枚举类型的所有枚举常量，且数组中的元素严格保持其在枚举类型中的声明顺序
    - String name()：返回当前枚举常量的名字
    - int ordinal()：返回该枚举常量在声明中的次序值
    - EnumType valueOf(String)：获得枚举常量名字符串对应的枚举常量实例

```
public class TestEnum{
    public static void main(String[] args){
        for(Season s: Season.values()){
            if(s.ordinal()==1)        //s为SUMMER
                System.out.println(s.name()+“;” +s.toString());
            if(s.ordinal()==2)        //s为FALL
                System.out.println(s.getDeclaringClass());
            if(s.equals(Season.valueOf(“WINTER”)))
                System.out.println(s);
        }
    }
}
enum Season{ SPRING, SUMMER, FALL, WINTER }
```

```
enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    private final int value;
    Coin(int value) { this.value = value; }
    public int value() { return value; }
}
enum CoinColor { COPPER, NICKEL, SILVER }
public class CoinTest {
    public static void main(String[] args) {
        for (Coin c : Coin.values()) {
            System.out.print(c + ": " + c.value() + ", ");
            switch (c) {
                case PENNY:
                    System.out.println(CoinColor.COPPER); break;
                case NICKEL:
                    System.out.println(CoinColor.NICKEL); break;
                case DIME:
                case QUARTER:
                    System.out.println(CoinColor.SILVER); break;
            }
        }
    }
}
```