



Java程序设计

2019春季

彭启民

pengqm@aliyun.com


抽象类Vs.接口

- 对于**abstract class**和**interface**的选择反映出对于问题领域本质的理解、对于设计意图的理解是否正确、合理。



■ 一个类可以实现多个接口


- `implements`后面列出接口名，以逗号分隔。
- 当没有缺省的实现用来继承时，通常使用接口而不使用抽象类。



```
interface OnEarthFunc{
    void earthDo();
}
interface InWaterFunc{
    void waterDo();
}
interface OnAirFunc{
    void airDo();
}
```

```
class Vulture implements OnEarthFunc, OnAirFunc{
    void earthDo(){}
    void airDo(){}
}
```


```
class Crocodile implements OnEarthFunc, InWaterFunc{
    void earthDo(){}
    void waterDo(){}
}
```

- 
- 抽象类可以有私有方法或私有变量
 - 接口是公开的，里面不能有私有的方法或变量
 - 抽象类中可以赋予方法的默认行为
 - 接口中方法不能拥有默认行为(Java8.0开始允许，且允许定义静态方法)

■ Java 8允许给接口添加一个非抽象的方法实现

- 使用 **default**关键字，即虚拟扩展方法
- 虚拟扩展方法不能是抽象方法，会被接口的实现类继承或者覆写

```
interface Formula {  
    double calculate(int a);  
  
    default double sqrt(int a) {  
        return Math.sqrt(a);  
    }  
}
```

- 
- 实现抽象类可以有选择地重写需要用到的方法
 - 实现接口一定要实现接口里定义的所有方法
 - 一个类只能继承一个父类，但可以通过继承多个接口实现多重继承

```
interface func{  
    public void print();  
}
```


```
interface show extends func{  
    public void list();  
}
```

```
class demo implements show{  
  
    public void print() {  
        System.out.println("print");  
    }  
    public void list() {  
        System.out.println("list");  
    }  
  
}
```

```
public interface InterfaceC extends InterfaceA, interfaceB {  
}
```

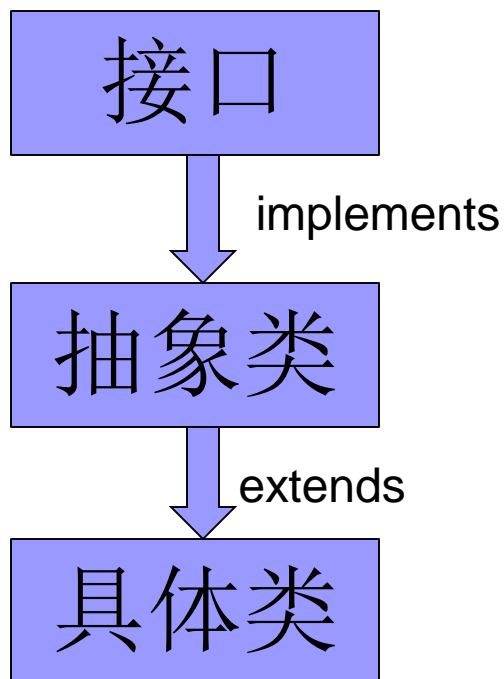
接口的继承

- 抽象类中可以有自己的数据成员，默认是 **friendly** 型，其值可以在子类中重新定义，也可以重新赋值。
- 接口中只能够有静态的不能被修改的数据成员，即必须是 **static final**（默认为 **public static final**）的并必须给其初值（接口中一般不定义数据成员）

- 
- 抽象类中可以有**abstract**的成员方法，也可以有非 **abstract**的成员方法
 - 接口中所有的成员方法都是**abstract**的，默认都是 **public abstract** 类型

- 
- 两者都不能被实例化
 - 从某种意义上说，**interface**是一种特殊形式的 **abstract class**

- 一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现




- 抽象类主要用来进行类型隐藏，体现了一种继承关系，要想使得 继承关系合理，父类和派生类之间必须存在"**is-a**"关系，即父类和派生类在概念本质上应该是相同的。


- is-a

- 对于接口，则并不要求接口的实现者和接口定义在概念本质上是一致的。

- like-a

- 
- 通过接口定义行为能够更有效地分离行为与实现，为代码的维护和修改带来方便，其中的关键在于区分对象的行为和对象的实现
 - 行为模型应该总是通过接口而不是抽象类定义

```
interface Flyanimal{
    void fly();
}
class Insect {
    int legnum=6;
}
class Bird {
    int legnum=2;
    void egg(){};
}
class Ant extends Insect implements Flyanimal {
    public void fly(){
        System.out.println("Ant can fly");
    }
}
class Pigeon extends Bird implements Flyanimal {
    public void fly(){
        System.out.println("pigeon can fly");
    }
    public void egg(){
        System.out.println("pigeon can lay eggs ");
    }
}
public class InterfaceDemo{
    public static void main(String args[]){
        Ant a=new Ant();
        a.fly();
        System.out.println("Ant's legs are"+ a.legnum);
        Pigeon p= new Pigeon();
        p.fly();
        p.egg();
    }
}
```



访问控制符与修饰符（2）

final修饰符的作用

■ 防止方法重写或类的继承

- 与方法重写和继承的强大功能与广泛用途一样，有时也需要阻止它们。例如，可能有一个封装了对某些硬件设备控制的类。而且，这个类可能提供给用户初始化设备、使用私有信息的能力。这种情况下，您会不希望类的用户重写初始化方法。



■ final三种用途:

- 类--使用final防止继承
- 方法--使用final防止重写
- 数据--创建常量

final类:

- final类的数据可以是final的，也可以不是final的。
- 如果一个类被声明为final，意味着它不能再派生出新的子类，不能作为父类被继承。
 - 一个类不能既被声明为abstract的，又被声明为final的。
- final类不能被继承，不能被覆盖
 - String，Math等是final类。



final类在执行速度方面比一般类快。

- 不涉及继承和覆盖。

- 其地址引用和装载在编译时完成。

- 在运行时不要求JVM执行因覆盖而产生的动态地址引用而花费时间和空间

。

- 与继承链上的一般对象相比，垃圾回收器在收回**final**对象所占据的地址空间时也相对简单快捷。

final方法

■ 使用目的：

- 为方法上“锁”，禁止派生类进行修改。设计程序时，若希望一个方法的行为在继承期间保持不变，而且不可被覆盖或改写，就可以采取这种做法。
- 效率。如果方法是final的，编译器在调用时转换成inline的。

■ private方法都隐含有final的意思。

■ final参数：可以读，但是不能改参数。

使用final方法不保证可能提高执行速度

- 并不是所有final方法其地址的装载和引用在编译时间完成。
 - 假设类C继承了B，B继承了A，在类A中有final方法。对类C来讲，调用A的final方法的确是inline编译，即装载在编译时间完成；但对A和B来讲，可能没有调用final方法。而在执行期间，JVM动态装载的方法有可能并不是C所调用的final方法。这种情况下，则不能够取得提高执行速度的结果。
- 如果final方法在编译时间装载到JVM，而且没有在执行期间覆盖的，可以取得inline效益，提高执行速度。

- **final**对象引用：一旦连到某个对象就不能连别的了，但是对象本身可以修改。

- **final**常量

- 编译时的常量，再也不能改,必须在定义时赋值。
 - 运行时初始化的值，以后不想再改。

空白的final数据

- 把数据成员声明为**final**的，但是却没给初始化的值。使用之前，必须先进行初始化，这一要求是强制性的。
- 对象里的**final**数据就能在保持不变性的同时又有不同了，更灵活。

静态修饰符、静态字段和方法

❖ 静态修饰符static

称为静态修饰符，可以修饰类中的属性和方法，被static修饰的属性称为静态属性。

这类属性一个最本质的特点是：它们是类的属性，而不属于任何一个类的具体对象。换句话说，对于该类的任何一个具体对象而言，静态属性是一个公共的存储单元，任何一个类的对象访问它时，取到的都是相同的数值，同样任何一个类的对象去修改它时，也都是在对同一个内存单元做操作。

```
class Employee
{
    public Employee(String n, double s)
    {
        name=n;
        salary=s;
    }

    public Employee()
    {
        name=" ";
        salary=0;
    }
    //对象初始化模块
    {
        id=nextId;
        nextId++;
    }
    private String name;
    private double salary;
    private int id;
    private static int nextId;
}
```

静态修饰符、静态字段和方法

❖ 静态修饰符

可满足两方面的要求：

（1）一种情形是只想用一个存储区域保存一个特定的数据——无论要创建多少个对象，甚至根本不创建对象；

（2）另一种情形是需要一个特殊的方法，它没有与这个类的任何对象关联。也就是说，即使没有创建对象，也需要一个能调用的方法。

静态方法

声明一个方法为static:

- (1) 使用时以类名做前缀，而不是一个具体的对象名；
- (2) static的方法是属于整个类的，它在内存中的代码段将随着类的定义而分配和装载，不被任何一个对象专有；
非static的方法是属于某个对象的方法，在这个对象创建时对象的方法在内存中拥有自己专用的代码段。
- (3) static方法属于整个类，不能操纵和处理属于某个对象的成员变量，只能处理属于整个类的成员变量。

■ main方法

- main方法并不对任何对象施加操作。当程序开始执行时还不存在任何对象，静态方法被执行并构造程序所需的对象。

■ 静态字段

如果把一个字段定义为static(静态的)，那么在一个类中只能有一个这类字段。

■ 静态常量

静态变量是很少见的，静态常量却很普遍。例如，Math类中的定义：

```
public class Math
{
.....
public static final double PI=3.1.4159265358979323846;
.....
}
```

- 实例字段和实例方法是属于某一具体实例对象的字段和方法，必须先创建这个实例对象，然后才能使用这些字段和方法。
- 对于同一个类创建的不同的实例对象，其字段可以有不同的取值，以反映该对象的不同状态。

■ 最终属性

- 无论static还是final字段，都只能存储一个数据，而且不得改变。
 - 对象里的final数据在保持不变性的同时又有所不同。
- 在对象被构造时， **final**字段(不能改变的) 必须被初始化。即，必须保证在每一个构造函数结束之前其值已被设定。以后字段的值不能改变。

访问控制符	当前类 (D)	当前类的所有子类 (C)	当前类所在的包 (B)	所有类 (A)
private	✓			
protected	✓	* ✓	✓	
public	✓	✓	✓	✓
private \vee protected \vee	✓	✓		

类及其成员修饰符关系：

类修饰符		public 公共类	缺省	abstract 抽象类无 对象	final 最终类
类成员					
成员 访问 控制 符	public	所有其他类皆可访问	本包中的类可以访问		
	protected	本包中其他类和所有其他包中的子类	本包中的其他类		
	private protected	仅该类的所有子类	仅该类当前包中子类		
	private	仅该类本身可以访问		非法	
	缺省	本包中的类可以访问			



内部类

```
class OuterClass
{
    static class A { } //静态内部类
    class B { } //非静态内部类
    public void disp( )
    {
        class C{ } //局部内部类
    }
}
```

- 编译后将产生下面的一些类文件:

```
OuterClass.class
OuterClass$A.class
OuterClass$B.class
OuterClass$1$C.class
```

内部类

■ 定义在另一个类内部的类

- 一个内部类的对象能够访问创建它的对象的实现，包括私有数据。
- 内部类作为一个成员，可以被任意修饰符修饰（外部类只有**public**和默认的修饰符）
- 对于同一个包中的其他类来说，内部类不管方法的可见性如何，那怕是**public**，除了包容类，其他类都无法使用它。
- 编译器在编译时，内部类的名称为 `OuterClass$InnerClass.class`

- 隐藏你不想让别人知道的操作，也即封装性。
- 内部类可以嵌套
- 内部类可以被定义为抽象类
- 当一个类继承自一个内部类时，缺省的构造器不可用，必须使用如下语法：

内部类对象的引用`.super()`;

- **Java**里用如下格式表达外部类的引用：

外部类名`.this`

（内部类的成员变量可与外部类的成员变量同名）

```
class Outer {
    private int index = 10;
    class Inner {
        private int index = 20;
        void print() {
            int index = 30;
            System.out.println(this);
            System.out.println(Outer.this);
            System.out.println(index);
            System.out.println(this.index);
            System.out.println(Outer.this.index);
        }
    }
    void print() {
        Inner inner = new Inner();//得到内部类的引用
        inner.print();
    }
}

class Test {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.print();
    }
}
```

```
class Outer {
    private int index = 10;
    class Inner {
        private int index = 20;
        void print() {
            int index = 30;
            System.out.println(this); // the object created from the Inner
            System.out.println(Outer.this); // the object created from the Outer
            System.out.println(index); // output is 30
            System.out.println(this.index); // output is 20
            System.out.println(Outer.this.index); // output is 10
        }
    }
    void print() {
        Inner inner = new Inner();//得到内部类的引用
        inner.print();
    }
}

class Test {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.print();
    }
}
```

■ 非静态内部类

- 不能有静态数据，静态方法或者又一个静态内部类
- **Java**编译器在创建内部类对象时，隐式的将其外部类对象的引用也传了进去并一直保存着，一个内部类对象可以访问创建它的外部类对象的内容，甚至包括私有变量。
- 在创建非静态内部类对象时，一定要先创建起相应的外部类对象。
- 非静态内部类对象有着指向其外部类对象的引用



■ 静态内部类

- 和非静态内部类相比，区别就在于静态内部类没有了指向外部的引用。

■ 局部内部类

- 定义在一个方法甚至一个代码块之内。
- 不能使用访问控制符和**static**修饰符修饰。
- 只能使用方法中的**final**常量。

```
class OuterClass{
    static class A{//静态内部类
        public A( ){
            System.out.println("Test$A !");
        }
    }
    class B{//非静态内部类
        public B(){
            System.out.println("Test$B !");
        }
    }
    public void disp( ) {
        final int a=10;  int b;
        class C{ //成员函数中的局部内部类
            public C( ) {
                System.out.println("in class C a="+a);
            }
        }
        C c=new C( );
    }
}
```

```
public class Test extends OuterClass
{
    public static void main(String args[])
    {
        OuterClass.A a=new OuterClass.A();           //建立静态内部类对象

        B b=new OuterClass( ).new B();

        //建立非静态内部类的对象
        //注意这个OuterClass().new B();相当于生成一个外部类的对象,然后在利用外部类对象生成内部类对象

        OuterClass t=new OuterClass( );
        t.disp( );
        //通过外部对象调用一个对象方法的形式,新建立了对象C.
    }
}
```

内部类的继承

当一个类继承自一个内部类时，缺省的构造器不可用。

```
class WithInner
{
    class Inner
    {
        public void sayHello()
        {
            System.out.println("Hello.");
        }
    }
}

public class Test extends WithInner.Inner
{
    Test(WithInner wi)
    {
        wi.super();
    }
    public static void main(String[] args)
    {
        WithInner wi=new WithInner();
        Test test=new Test(wi);
        test.sayHello();
    }
}
```

每一个内部类都有一个指向外部类的引用，继承一个内部类，必须先创建一个外部类，通过这个外部类引用来调用其内部类的构造方法。

如果继承的内部类是一个静态内部类则直接`super()`调用。

匿名类

- 只需创建一个类的对象而无需用到其名字
- 语法规则：

`new interfacename(){.....};`

或

`new superclassname(){.....};`

- 必须在创建时，作为new语句的一部分来声明它们

- 匿名内部类必须扩展一个基类或实现一个接口，但是不能有显式的**extends**和**implements**子句

- 匿名内部类必须实现父类以及接口中的所有抽象方法

- 匿名内部类总是使用父类的无参构造方法来创建实例。如果是实现了一个接口，则其构造方法是**Object()**

- 匿名内部类编译后的命名为：**OuterClass\$n.class**，其中**n**是一个从**1**开始的整数，如果在一个类中定义了多个匿名内部类，则按照他们的出现顺序从**1**开始排号。

- 匿名内部类可以很方便的定义回调；
 - 内部类是用来在某个时刻调用外面的方法而存在的一一回调。
 - 使用内部类可以非常方便的编写事件驱动程序
- 对一个给定的类进行扩展；
- 实现一个给定的接口。



```
interface pr
```

```
{  
    void print1();  
}
```

```
public class noNameClass
```

```
{  
    public pr dest()  
    {  
        return new pr(){  
            public void print1()  
            {  
                System.out.println("Hello world!!");  
            }  
        }  
    }  
}
```

```
public static void main(String args[])
```

```
{  
    noNameClass c = new noNameClass();  
    pr hw = c.dest();  
    hw.print1();  
}  
}
```


- 当一个内部类的类声名只是在创建此类对象时用了一次，而且产生的新类需继承于一个已有的父类或实现一个接口，才能考虑用匿名类。
- 所谓的匿名就是该类连名字都没有，只是显示地调用一个无参的父类的构造方法。
 - 由于匿名类本身无名，因此它也就不存在构造方法，可重写父类的方法。

■ 匿名内部类的初始化

- 匿名内部类没有构造函数，但是如果这个匿名内部类继承了一个只含有带参数构造函数的父类，创建它的时候必须带上这些参数，并在实现的过程中使用`super`关键字调用相应的内容。
 - 如果是在一个方法中的匿名内部类，可以利用这个方法传进你想要的参数，不过记住，这些参数必须被声明为`final`。
 - 将匿名内部类改造成有名字的局部内部类以拥有构造函数
- 在匿名内部类中使用初始化代码块。

■ Java8对匿名类的改进

- 有Lambda表达式允许将函数当成参数传递给某个方法，或者把代码本身当作数据处理
 - 比匿名类更好用。从Lambda表达式可访问外部变量，匿名对象只能访问外部访问有final定义的变量。
 - 最简单的Lambda表达式可由逗号分隔的参数列表、->符号和语句块组成，例如：

```
Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );
```

■ Lamda与匿名类

- Lambda表达式是匿名内部类的一种简化，因此它可以取代匿名内部类的作用。
- 相同点
 - 都可以直接访问"**effectively final**"的局部变量，以及外部类的成员变量（包括实例变量和类变量）
 - 创建的对象都可以直接调用从接口中继承的默认方法。

■ Lamda与匿名类

□ 区别

- 匿名内部类可以为任意接口创建实例——不管有多少个抽象方法，只要匿名内部类实现了所有方法即可，**Lambda**表达式只能为函数式接口创建实例。
- 匿名内部类可以为抽象类甚至普通类创创建实例，**lambda**表达式只能为函数式接口创建实例。
- 匿名内部类实现的抽象方法体允许调用接口中的默认方法，**Lambda**表达式的代码块不允许调用接口中的默认方法。

Java语言中方法的调用有两类：

- 一类是需要程序书写专门的调用命令来调用的方法，称为**程序调用方法**，例如split ()；
- 另一类是运行过程中系统自动调用的方法，不需要在程序里书写专门的调用方法的命令，称为**系统调用方法**，例如事件处理方法的调用等。



Java API中的方法

Java API（Application Program Interface，也称为Java类库）提供了丰富的类和方法。

例如：常见的算术运算、字符串操作、字符操作、输入输出、错误检查等操作。

Java API中的方法--Math类的方法

方法	说明	方法	说明
abs(x)	x 的绝对值(这个方法还有 float,int 和 long 型值的版本)	max(x,y)	取 x 和 y 中较大者。
ceil(x)	不小于 x 的最小整数(向上取整)	min (x,y)	取 x 和 y 中较小者。
cos(x)	x 的余弦函数值(x 以弧度为单位)	pow (x,y)	x 的 y 次幂
exp(x)	指数方法 e 的 x 次幂	sin (x)	x 的正弦函数值(x 以弧度为单位)
floor(x))	不大于 x 的最大整数(向下取整)	sqrt (x)	x 的平方根
log(x)	x 的自然对数(以 e 为底)		

Java API中的方法--Math方法的调用实例

例:计算并且打印出900的平方根, 调用方法的语句格式:

```
System.out.println(Math.sqrt(900));
```



包

包

- 在**Java**语言中，包的使用是它的一大特色。使用**Java**提供的大量包，程序员可以轻松、方便地编写出复杂的、功能强大的应用程序。
- 这些包就是**Java**应用程序编程界面，即**Java API**。它是为用户开发自己的类、小应用程序和应用程序而设计。

■ java-package包的使用

- 当一个大型程序交由数个不同的程序人员开发时，**java**中为了避免相同的类名的冲突事件，提供了一个包的概念（**package**）



■ java跨平台特性的需求

- java中的所有资源也是以文件方式组织，大量的类文件需要组织管理。
- java中同样采用了目录树形结构。虽然各种常见操作系统平台对文件的管理都是以目录树的形式组织，但是它们对目录的分隔表达方式不同，为了区别于各种平台，java中采用了"."来分隔目录。



Java API Packages

■ API文档

- 查看JavaAPI提供的类的简介
- 提供类的说明
- 类的继承关系
- 类的属性
- 类的方法

联机文档: **JavaSE 8.0 Documentation**

■ 在Java语言，最基本的包就是Java语言核心API，基本内容包括：

- ☐ java.lang
- ☐ java.lang.reflect
- ☐ java.bean
- ☐ java.rmi、java.rmi.registry和java.rmi.server
- ☐ java.security、java.security.acl和java.security.interfaces
- ☐ java.io
- ☐ java.util
- ☐ java.util.zip
- ☐ java.net
- ☐ java.awt
- ☐ java.awt.image
- ☐ java.awt.peer
- ☐ java.awt.datatransfer
- ☐ java.awt.event
- ☐ java.applet
- ☐ java.sql
- ☐ java.text



■ java.lang

- Java语言中最核心的包，提供了最基本的数据类型，编程框架。

■ java.io

- io，就是input/output的缩写，Java语言的标准输入/输出库。

■ java.util

- 由几个基本的实用类组成的，如日期类。

■ java.net

- 具备网络处理功能的类，使用这个包就可轻松地创建、连接套接字（socket）、可以编写出自己的telnet、FTP等程序。

■ java.awt.*

- 代表java.awt以及java.awt.image等其它包。这些包就是通常说的Java Abstract Window Toolkit（抽象窗口工具包，简称AWT）。它是由一些能够为用户的小应用程序创建丰富、有吸引力的、实用的界面资源组成的。它是使用Java开发图形界面程序时必不可少的一个包。使用它就可创建出美丽的图形世界。

■ java.applet

- 要写Applet程序就要使用它：import java.applet

■ java.sql

- 包含了JDBC（Java DataBase Connect，Java数据库连接），使用它可以开发出数据库应用程序。



■ java中包结构和平台的衔接

- java中的资源存在于不同平台下时必然会有很大差异。因此跨平台的java包结构和平台之间必须通过一种方式来衔接到一起。事实上它们就是通过我们很熟悉的classpath的设置来衔接到一起的。

包

■ 导入包

- `import 包名.类名;` //单类型导入
- `Import 包名.*;` //按需类型导入，不是导入一个包中的所有类！

注：不使用import时要显示指定类所属的包名，如
`java.sql.date`

■ 定义包

- java源文件的第一条语句：
`package 包名;`
例： `package ucas.test;`

Cautions

- *Import* does NOT copy code
 - ONLY to help with resolving the names
 - Simply makes the names known
 - Exactly identical to typing the fully-qualified name everywhere
- You get some for free
 - Current package
 - Default package
 - `java.lang.*`

Collisions

- What happens if two libraries are imported and they include the same names?
 - package java.util has a Date
 - so does java.sql
- If you don't use any Date, you're fine
- If you do, you need to specify which one

```
java.util.Date dt = new java.util.Date();
```

正确使用"包"

- 对类路径的设置通常有两种方法:

- i) 在系统的环境变量中设置, 设置方法依据平台而变;
- ii) 以命令参数的形式来设置。

- 如:

```
javac -classpath d:\jdk8\lib d:\cjm\edu\test\TestFile.java  
java -classpath .;d:\jdk8\lib; d:\cjm edu.test.TestFile
```

- 类的目录结构要和类中第一句“包声明”一致。
 - 如类TestFile.class对应的.java文件的第一句：
`package cas.test;`
- 为了便于工程发布，可以将自己的类树打成.jar文件。
- 对其它资源的使用，如图标文件，文本等资源文件的使用必须要注意，查找资源文件不应从类文件所在的目录开始，而是应该从package指定的类路径的起点开始。

创建独一无二的包名

- 把package名称分解为机器上的一个目录，利用操作系统的层次化的文件结构来解决这个问题
- package名称的第一部分是反顺序的类的创建者的internet域名
- 运行方法：
 java pack1.Test
 □ 包名. 类名

■ 确保类的存放路径和类中指明的"包路径"一致

- i)编写.java文件时存放的目录事先确定好，如TestFile.java就直接放在cas\test目录下，然后用下面的语句编译：

```
javac -classpath d:\jdk8\lib  
d:\ac\cas\test\TestFile.java
```

当编译完成后，产生的TestFile.class文件会出现在编译命令中java文件的描述路径中。即出现在d:\ac\cas\test中

- ii)通过-d参数的使用来编译程序。如使用下面的语句来编译：

```
javac -d d:\ac d:\temp\TestFile.java
```

将在-d后指定的目录d:\ac下面自动按照packagek中指定的目录结构来创建目录，并且将产生的.class文件放在这个新建的目录下，即在d:\ac下面建立\cas\test目录，然后产生的TestFile.class放在d:\ac\cas\test目录下。

- **java**开发过程中，第三方的包文件只有放置到正确的文件夹下才能被正确的使用
 - 已经打包成**jar**文件，则将该文件放到 **jdk\jre\lib\ext**文件夹下
 - 没有打包，则将文件夹放到**jdk\lib**文件夹下
 - 加上 **-Xbootclasspath/a:**第三方**jar**包路径;
java -Xbootclasspath/a:d:\jxl.jar; -jar d:\test.jar

■ Java5.0增加的静态引用

以前:

```
import java.lang.Math; //程序开头处
```

...

```
double x = Math.random();
```

新:

```
import static java.lang.Math.random; //程序开头处
```

...

```
double x = random();
```

有益的建议

(1) 保持数据私有。

(2) 初始化数据。

Java并不对本地变量初始化，但会初始化对象中的实例字段，但是永远不要依赖于默认值。

(3) 不要在一个类中使用太多的基本类型。

(4) 次序约定：

公开部件；包作用域部件；私有部件

在每一部分内我们列出次序如下：

实例方法；静态方法；实例字段；静态字段

(5) 分解, 简化。



容器类



■ 用途：保存对象

- 容器可以管理对象的生命周期、对象与对象之间的依赖关系

■ 整个Java容器类的基础是容器接口

□ Collection

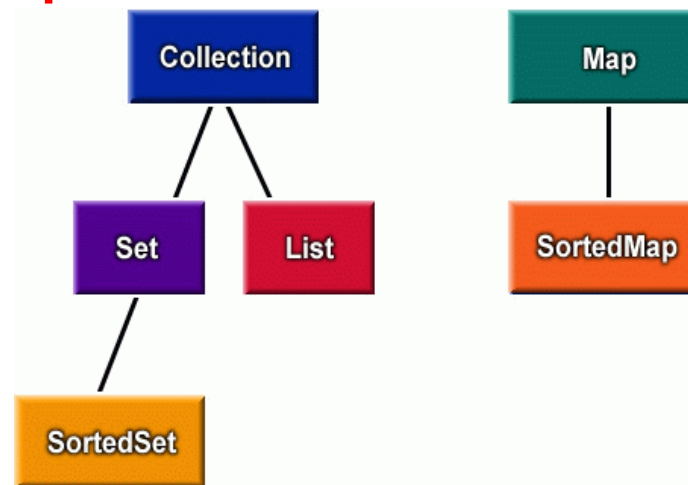
■ Set (集) → SortedSet

■ List (列表)

□ Map (映射) → SortedMap

□ Iterator → ListIterator

□ Comparator



集合接口

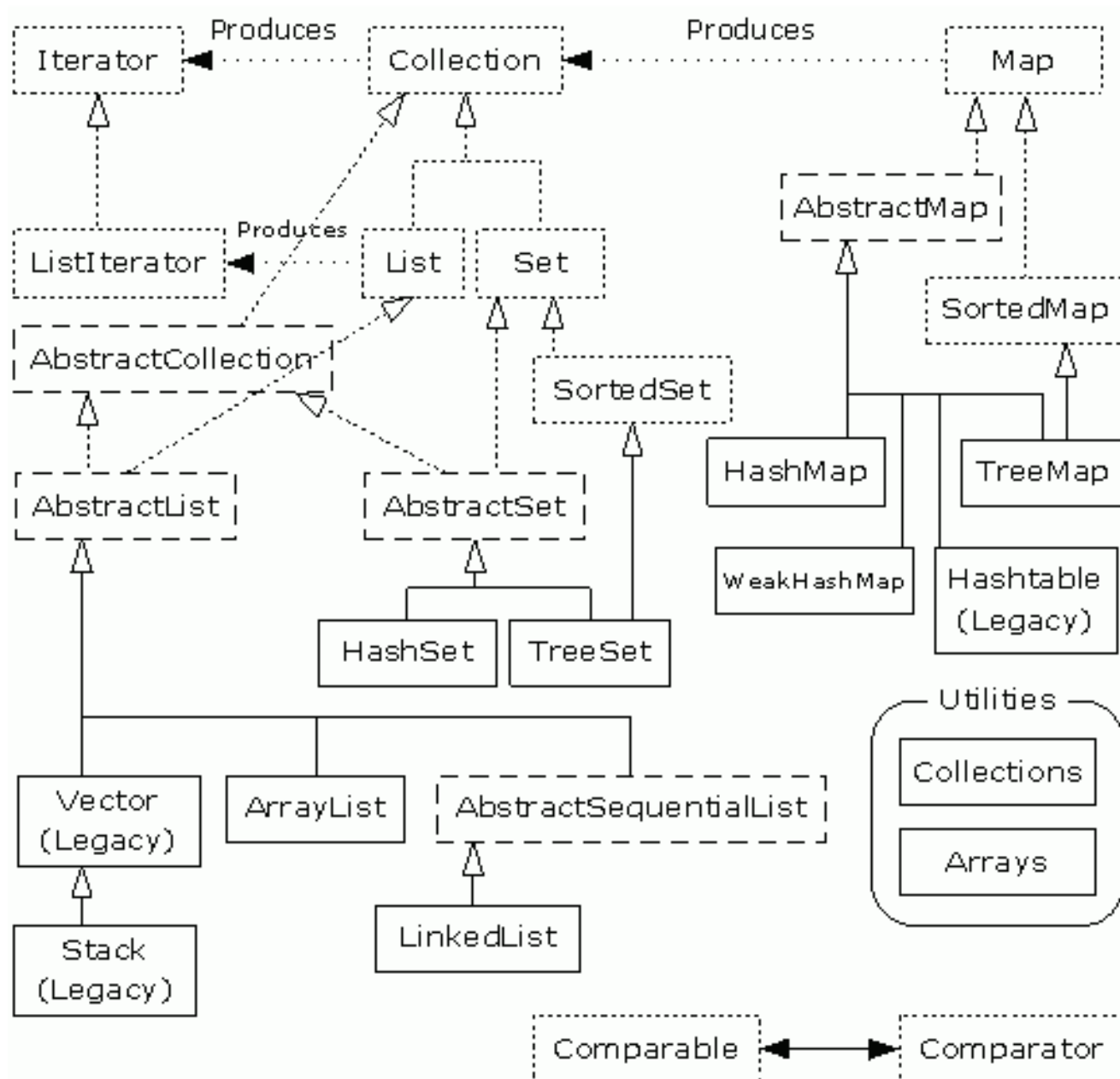
■ Collection:

- 每个元素都是单一对象。
 - List以特定顺序容纳元素。
 - Set中不能有重复的元素。

■ Map:

- 每个元素都是一对**key-value**（键值 / 实值）对象，且每个元素中的键值都不能与其他元素中的键值相同。

三个不可改变的静态变量：EMPTY_SET ,EMPTY_LIST
， EMPTY_MAP。



集合类

Collection

- └List 接口

 - └LinkedList 链表

 - └ArrayList 顺序结构动态数组类

 - └Vector 向量

 - └Stack 栈

- └Set

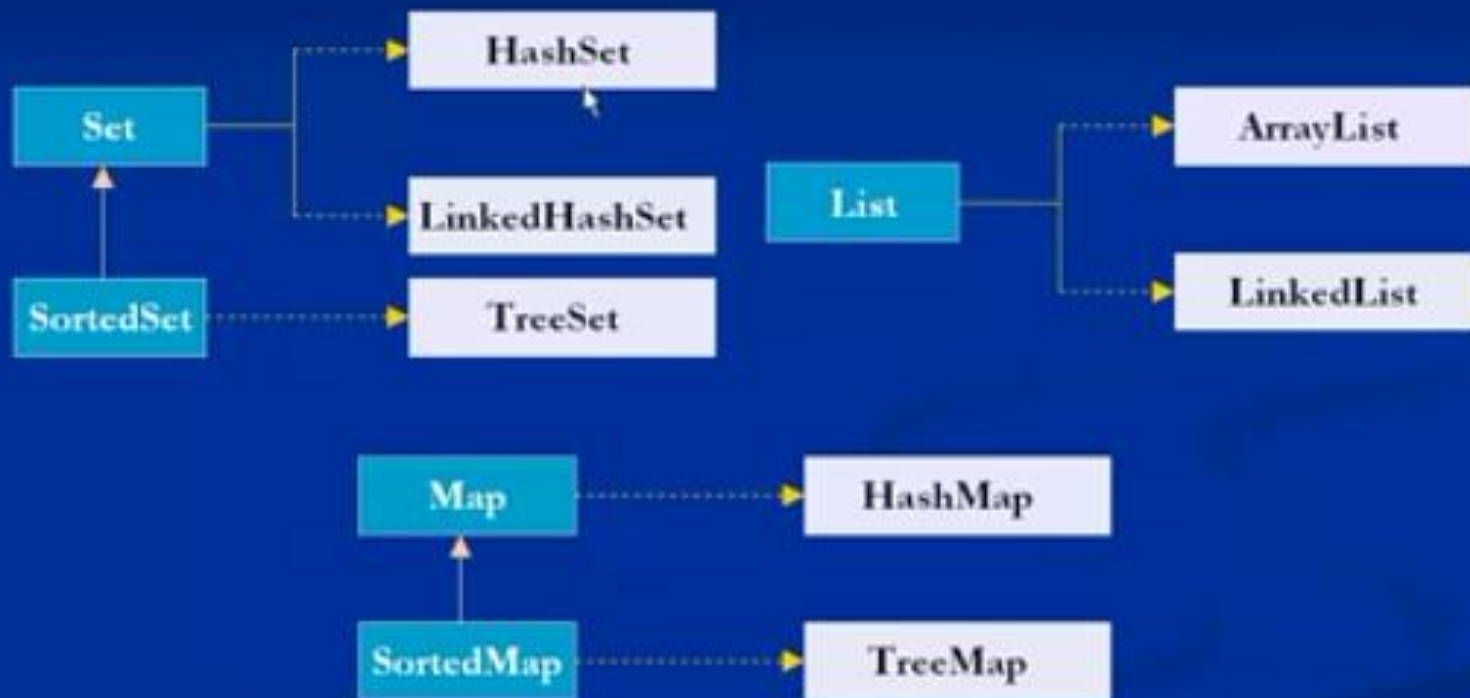
Map

- └Hashtable

- └HashMap

- └WeakHashMap List接口

集合框架中的实现类



接口	特性	实现类	实现类特性	成员要求
List	线性、有序的存储容器，可通过索引访问元素	ArrayList	数组实现。非同步。	
		Vector	类似ArrayList，同步。	
Map	保存键值对成员	LinkedList	双向链表。非同步。	任意Object对象，如果修改了equals方法，需同时修改hashCode方法 键成员要求实现Comparable接口，或者使用Comparator进行构造TreeMap。键成员一般为同一类型。
		HashMap	基于哈希表的Map接口的实现，满足通用需求	
		TreeMap	默认根据自然顺序进行排序，或者根据创建映射时提供的Comparator进行排序	
		LinkedHashMap	类似于HashMap，但迭代遍历时取得“键值对”的顺序是其插入顺序或者最近最少使用的次序	
		IdentityHashMap	使用==取代equals()对“键值”进行比较的散列映射	
		WeakHashMap	弱键映射，允许释放映射所指向的对象	
Set	成员不能重复	ConcurrentHashMap	线性安全的Map	元素必须定义hashCode() 元素必须实现Comparable接口 元素必须定义hashCode()
		HashSet	为快速查找设计的Set	
		TreeSet	保持次序的Set，底层为树结构	
		LinkedHashSet	内部使用链表维护元素的顺序（插入的次序）	

类描述

AbstractCollection

实现了大部分的集合接口。

AbstractList

继承于AbstractCollection 并且实现了大部分List接口。

AbstractSequentialList

继承于 AbstractList ， 提供了对数据元素的链式访问而不是随机访问。

LinkedList

继承于 AbstractSequentialList， 实现了一个链表。

ArrayList

通过继承AbstractList， 实现动态数组。

AbstractSet

继承于AbstractCollection 并且实现了大部分Set接口。

HashSet

继承了AbstractSet， 并且使用一个哈希表。

LinkedHashSet

具有可预知迭代顺序的 Set 接口的哈希表和链接列表实现。

TreeSet

继承于AbstractSet， 使用元素的自然顺序对元素进行排序。

AbstractMap

实现了大部分的Map接口。

HashMap

继承了HashMap， 并且使用一个哈希表。

TreeMap

继承了AbstractMap， 并且使用一颗树。

WeakHashMap

继承AbstractMap类， 使用弱密钥的哈希表。

LinkedHashMap

继承于HashMap， 使用元素的自然顺序对元素进行排序。

IdentityHashMap

继承AbstractMap类， 比较文档时使用引用相等。

序号 类描述

- 1 **Vector**
Vector类实现了一个动态数组。和ArrayList和相似，但是两者是不同的。
- 2 **Stack**
栈是Vector的一个子类，它实现了一个标准的后进先出的栈。
- 3 **Dictionary**
Dictionary 类是一个抽象类，用来存储键/值对，作用和Map类相似。
- 4 **Hashtable**
Hashtable是原始的java.util的一部分， 是一个Dictionary具体的实现 。
- 5 **Properties**
Properties 继承于 Hashtable.表示一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。
- 6 **BitSet**
一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。

Collection

public interface **Collection** extends **Iterable** {

// Basic properties

int size();

boolean isEmpty();

boolean contains(Object element); // use equals() for
comparison

boolean equal(Object);

int hashCode(); // new equals() requires new hashCode()

// basic operations

boolean add(Object); // Optional; return true if this
changed

boolean remove(Object); // Optional; use equals() (not ==)

// Bulk Operations

boolean containsAll(Collection c); // true if $c \subseteq \text{this}$

// the following 4 methods are optional, returns true if contents changed

boolean addAll(Collection c); // Optional; this = this \cup c

boolean removeAll(Collection c); // Optional; this = this - c

boolean retainAll(Collection c); // Optional; this = this \cap c

void clear(); // Optional; this = {};

// transformations

Iterator iterator(); // collection \rightarrow iterator ; imple. of Iterable

Object[] toArray(); // collection \rightarrow array

<T> T[] toArray(T[] a);

// if $|\text{this}| \leq |a| \Rightarrow$ copy this to a and return a ; else \Rightarrow create a new array b

// of the type T[] and copy this to b and return b;

}

set(interface)

- 最简单的一种容器，它的对象不按特定方式排序，只是简单的把对象加入容器中。
- 对**Set**（集）中成员的访问和操作是通过集中对象的引用进行的，所以集中不能有重复对象。

- 存入**Set**的每个元素必须是唯一的，不保存重复元素
- 加入**Set**的**Object**必须定义**equals()**方法以确保对象的唯一性。
- **Set**与**Collection**有完全一样的接口。
- **Set**接口不保证维护元素的次序
 - 里面的元素是无序的，不能通过索引操作**Set**对象。

List

- 主要特征是其对象以线性方式存储，没有特定顺序，只有一个开头和一个结尾
- 列表在数据结构中分别表现为：数组和向量、链表、堆栈、队列。



```
public interface List extends Collection {
```

```
// Positional Access
```

```
Object get(int index); // 0-based
```

```
Object set(int index, Object element); // Optional; return old  
value
```

```
void add([int index,] Object element); // Optional
```

```
Object remove(int index); // Optional; remove(Object) given in  
Collection
```

```
boolean addAll(int index, Collection c); // Optional
```

```
// Search
```

```
int indexOf(Object o);
```

```
int lastIndexOf(Object o);
```


```
// Range-view
```

```
List subList(int from, int to);
```

```
// Iteration ; Iterator iterator() given in Collection
```

```
ListIterator listIterator([int f] ); // default f = 0; return a listIterator with  
cursor set to position f
```

```
}
```

- 
- **Set**和**List**都是由公共接口**Collection**扩展而来，所以都可以使用一个类型为**Collection**的变量来引用。
 - 把一个列表或集传递给方法的标准途径是使用**Collection**类型的参数。

Map

- 每个项都是成对的。存储的每个对象都有一个相关的关键字（**Key**）对象，关键字决定了对象在**Map**（映射）中的存储位置，检索对象时必须提供相应的关键字，就像在字典中查单词一样。
 - 关键字应该是唯一的。
 - 关键字本身并不能决定对象的存储位置，它需要对过一种散列（**hashing**）技术来处理，产生一个被称作散列码（**hash code**）的整数值，散列码通常用作一个偏置量，该偏置量是相对于分配给映射的内存区域起始位置的，由此确定关键字/对象对的存储位置。
 - 理想情况下，散列处理应该产生给定范围内均匀分布的值，而且每个关键字应得到不同的散列码。

■ Map接口

- 一个Map可以返回的东西包括它的键值构成的一个Set、由它的值构成的一个集合或者由它的键值对构成的一个Set，可创建一个Set用来表达Map的某一部分。
 - 并不是一种由键值对构成的集合
- 一个 Map 容器中的键对象不允许重复，对于值对象则没有唯一性的要求。
- 用 put(Object key,Object value) 方法即可将一个键与一个值对象相关联。用 get(Object key) 可得到与此 key 对象所对应的值对象。



```
public interface Map { // Map does not extend  
    Collection
```

```
// Basic Operations
```

```
    // put or replace, return replaced object
```

```
Object put(Object key, Object value); // optional
```

```
Object get(Object key);
```

```
Object remove(Object key);
```

```
boolean containsKey(Object key);
```

```
boolean containsValue(Object value);
```

```
int size();
```

```
boolean isEmpty();
```


// Bulk Operations

void putAll(Map t); //optional

void clear(); // optional

// Collection Views;

// backed by the Map, change on either will be reflected on the other.

public Set keySet(); // cannot duplicate by definition!!

public Collection values(); // can duplicate

public Set entrySet(); // no equivalent in Dictionary

// nested Interface for entrySet elements

public interface Entry {


Object getKey();

Object getValue();

Object setValue(Object value);


}


}




■ Map接口基本方法

- Object get(Object key)
- Object put(Object key, Object balue)
- Set keySet()
- Set entrySet()


- 
- A Map can be viewed as a Collection in three ways:
 - **keySet**: the Set of keys contained in the Map.
 - **values**: The Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.
 - **entrySet**: The Set of key-value pairs contained in the Map.
 - The Map interface provides a small nested interface called **Map.Entry** that is the type of the elements in this Set.



```
public interface SortedMap extends Map {  
    Comparator comparator();  
    // range-view operations  
    SortedMap subMap(Object fromKey,  
        Object toKey);  
    SortedMap headMap(Object toKey);  
    SortedMap tailMap(Object fromKey);  
    // member access;  
    // Don't forget bulk of other Map operations  
    // available  
    Object firstKey();  
    Object lastKey();  
} // throws NoSuchElementException if  
    m.isEmpty()
```



```
SortedMap m = new TreeMap();  
m.put("Sneezy", "common cold");  
m.put("Sleepy", "narcolepsy");  
m.put("Grumpy", "seasonal affective  
disorder");  
System.out.println( m.keySet() );  
System.out.println( m.values() );  
System.out.println( m.entrySet() );
```

- 
- Running this snippet produces this output:
[Grumpy, Sleepy, Sneezzy]
[seasonal affective disorder, narcolepsy, common cold]
[Grumpy=seasonal affective disorder,
Sleepy=narcolepsy,
Sneezzy=common cold]

课后作业

- 利用已学知识，完成有关设计与编码：
- 1.完成对汽车的描述，要求体现继承机制，包含**3**个以上属性，**3**个以上方法；
- 2.根据定义的类，生产**5**个实体，选择容器类进行管理；
- 3.4月27日24时前课程网提交.java文件。