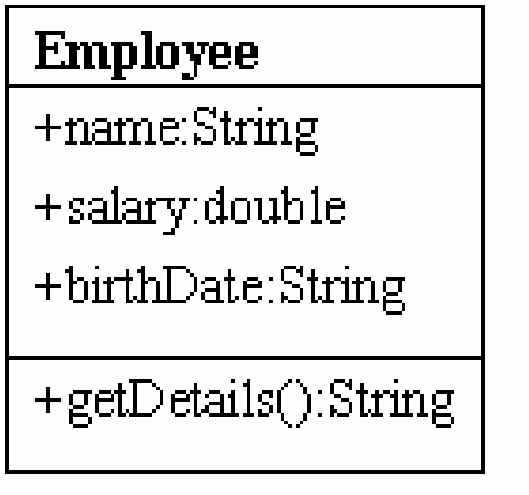
JAVA程序员培训-2

讲师：陈伟俊

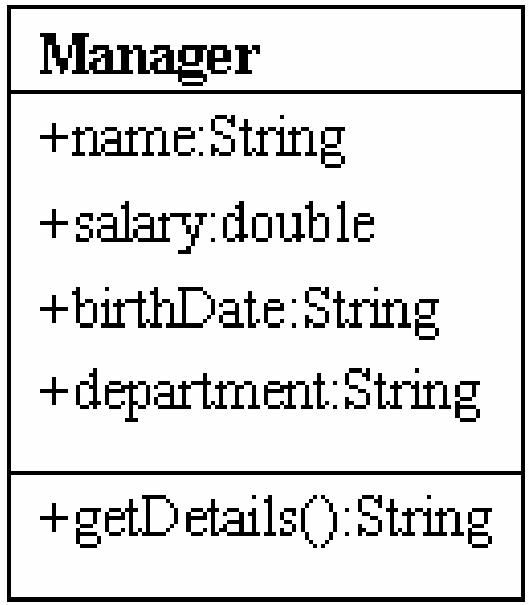
第六章 类设计

继承

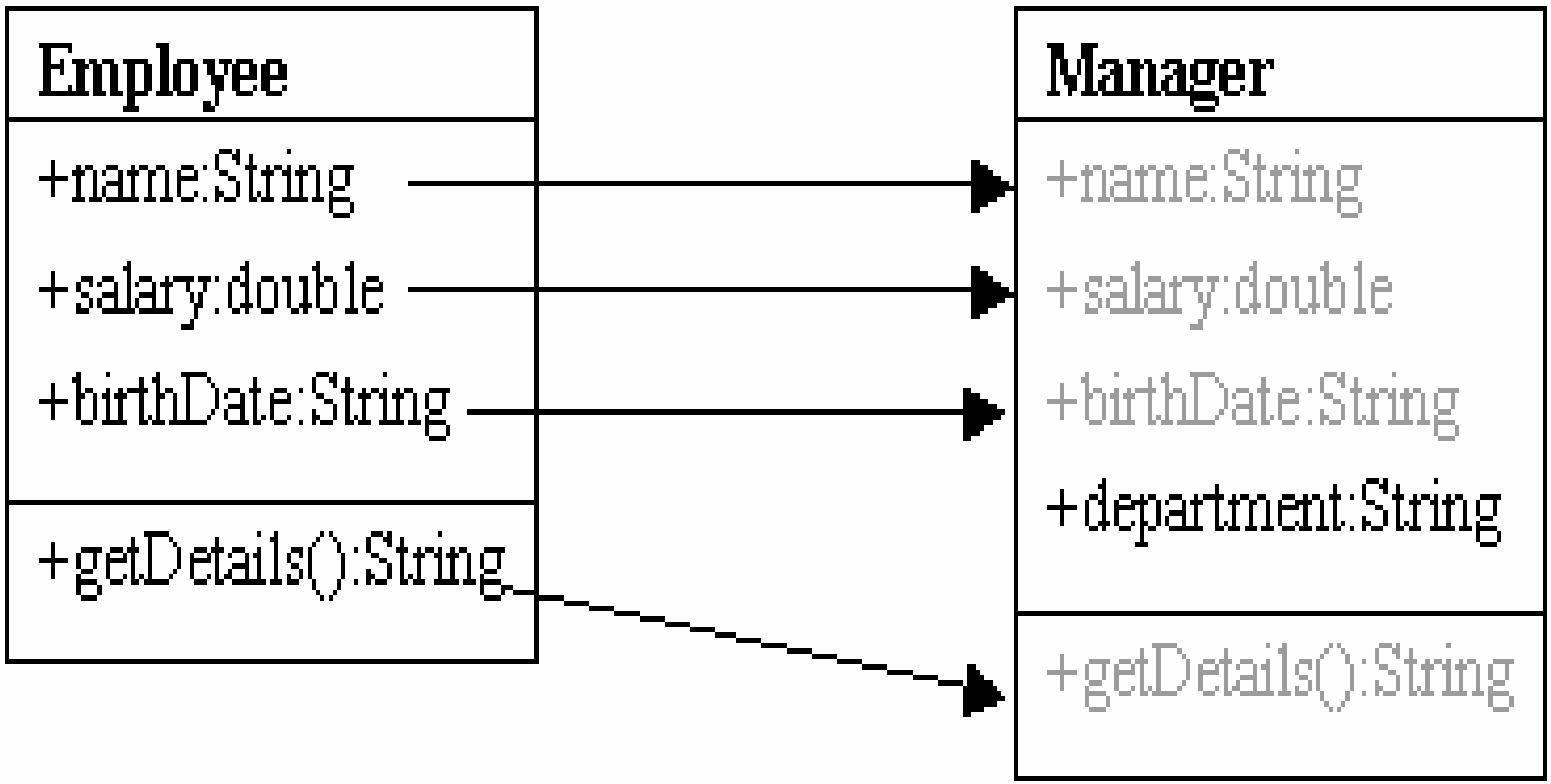
- Employee类



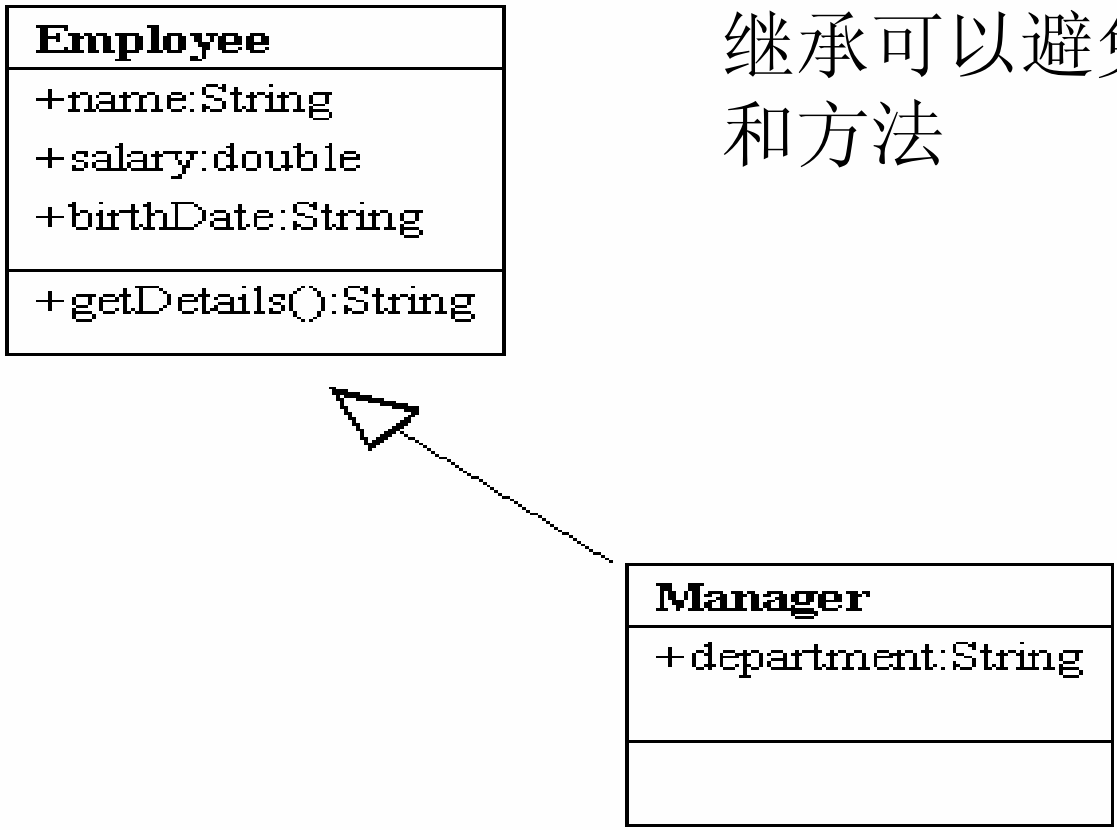
- Manager类



继承


Employee
+name:String
+salary:double
+birthDate:String
+getDetails():String
Manager
+name:String
+salary:double
+birthDate:String
+department:String
+getDetails():String

继承



继承可以避免重复定义属性和方法

继承

- 语法规则：

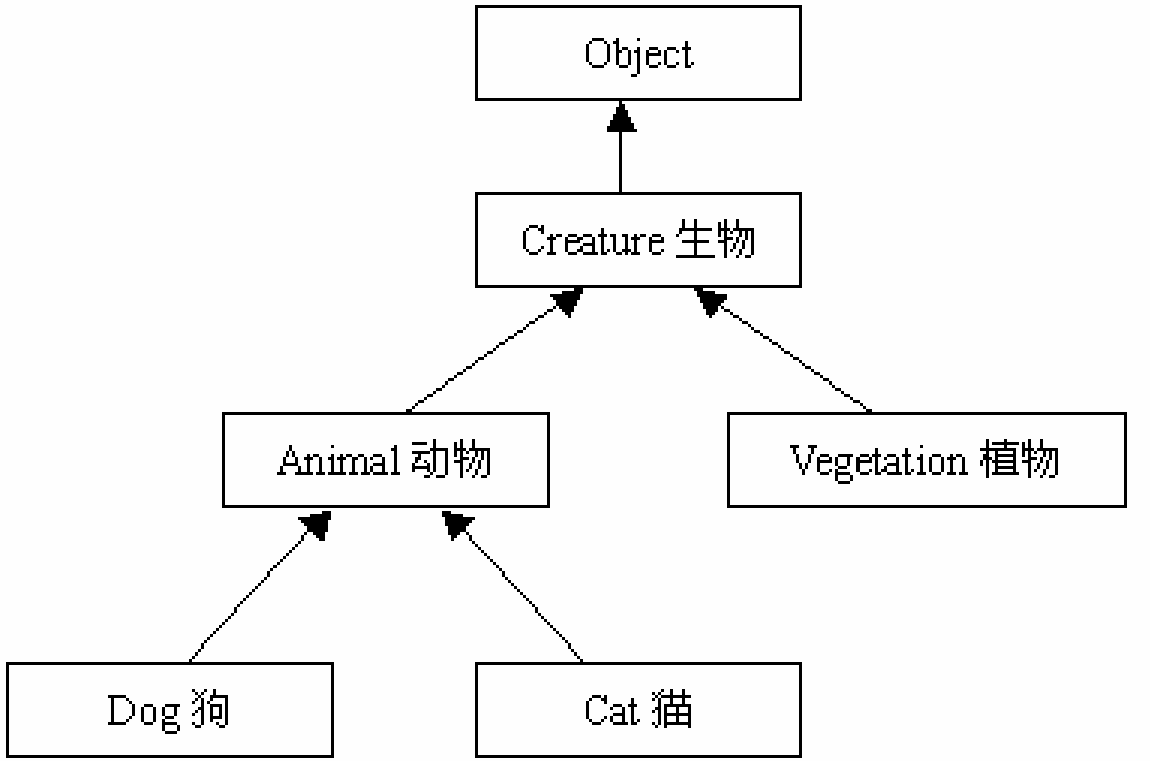
  <修饰符> class <子类名称> [extends 父类]{

  ……

  }

- JAVA中只有单继承
  - 一个父类可以有多个子类
  - 一个子类只能继承一个父类
- 所有的类都直接或间接的继承了Object类

单继承示例



访问控制

修饰符	同一个类	同一个包	子类	整体
private	是			
default	是	是		
protected	是	是	是	
public	是	是	是	是

方法的覆盖(override)

- 子类可以将父类的方法加以改造
- 覆盖的规则
  - 子类覆盖方法和父类被覆盖方法的方法返回类型，方法名称，参数列表必须相同
  - 子类覆盖方法的访问权限必须大于等于父类的方法的访问权限
  - 方法覆盖只能存在于子类和父类之间

方法的覆盖举例

- Employee类

```
public String getDetails(){
    return no + "/" + name + "/" + year;
}
```

- Manager类

```
public String getDetails(){
    return no + "/" + name + "/" + year + "/" + dept;
}
```

方法的重载(overload)

- 在类中同一种功能有不同的实现形式
- 方法重载的规则
  - 方法名相同
  - 方法的参数类型,个数,顺序至少有一项不同
  - 方法的返回类型和修饰符均不受限制
- 具体调用哪个方法,完全取决于调用时给定的参数(实参)

方法的重载举例

```
public class Math {
    public int sum(int n1,int n2){
        return n1 + n2;
    }
    public int sum(int n1,int n2,int n3){
        return n1 + n2 + n3;
    }
    public static void main(String[] args) {
        Math m = new Math();
        System.out.println(m.sum(4,5)); //9
        System.out.println(m.sum(1, 2, 3)); //6
    }
}
```

- 思考：重载sum方法，使之能够计算两double型数字的和

比较覆盖和重载的不同

	覆盖	重载
方法名	相同	相同
参数列表	相同	必须不同
返回类型	相同	可以不同
访问权限	大于等于被覆盖方法	无特殊要求
位置	在子类中覆盖父类	可以重载父类的，也可以重载在一个类的
次数	覆盖一次	重载多次
抛出异常	不能抛出更多异常	无特殊要求

可变长参数

- 可变长参数可以接收任意个数的实参，形参实际上是一个数组
- 可变长参数必须是最后一个参数
- 语法形式：
  - 方法名称（类型 参数1，类型 参数2，……，类型 … 变长参数）

可变长参数

```
public int getSum(int... a) {
        int sum = 0;

        for (int i = 0; i < a.length; i++) {
                sum += a[i];
        }

        return sum;
}

……

int sum = t.getSum(1,2,3,4,5);
System.out.println(sum);
```

回顾this

- 使用this的场合
  - 使被屏蔽的成员变量变为可见
  - 调用本类的其他构造方法（即其他重载方法）

this与重载构造器举例

```
public class Employee {
    ......
    public Employee(){

    }
    public Employee(String no){
           this(no,null,null);//调用其他构造器
    }
    public Employee(String no,String name){
           this(no,name,null);//调用其他构造器
    }
    public Employee(String no,String name,String year){
           this.no = no;
           this.name = name;
           this.year = year;
    }
    ......
```

super关键字

- 类中用super来指向它的父类
- 使用super的场合
  - 在子类中访问父类被屏蔽的属性和方法
  - 在子类构造方法中调用父类的构造方法

super举例

- 在类的构造方法中调用这个类的父类的构造方法

```
public Manager(String no,String name,String
   year,String dept){
    super(no,name,year); //调用父类的构造方法
    this.dept = dept;
}
```

- 调用父类的方法举例

```
public String getDetails(){
    return super.getDetails() + "/" + dept;
}
```

关于构造方法的几点说明

- 子类不能继承父类的构造器。
- 如果要调用其他构造器（无论子类本身还是父类），必须写在构造器中的第一行。
- 如果构造器中没有显式的this或super调用，则系统默认在构造器第一行插入一个默认父类无参构造器super（），此时如果父类显式的定义了构造器，但没有定义无参构造器会提示错误。
- 对父类的构造器调用会一直延伸到Object类。

对象构造和初始化步骤

- 分配存储空间和默认初始化
- 按下列步骤初始化实例变量

1. 绑定构造方法参数
2. 如果有this调用，则调用相应的重载构造方法，然后跳到步骤5
3. 显示或隐式的递归调用父类的构造方法（Object类除外）
4. 进行实例变量的显式初始化
5. 执行当前构造方法的方法体

思考

- 为什么必须在子类的构造器中加入调用父类的构造方法？

多态

- 宏观的说就是有不同形式的能力
- 具体表现之一就是一个子类的实例可以赋给一个父类类型的变量
- 例如
  - Employee e = new Employee();一个父类类型的变量引用一个父类类型的实例
  - Manager m = new Manager();一个子类类型的变量引用一个子类类型的实例
  - Employee e = new Manager(); 一个父类类型的变量引用子类类型的实例，这就是多态
  - Manager m = new Employee();一个子类类型的变量引用父类类型的实例，不可以

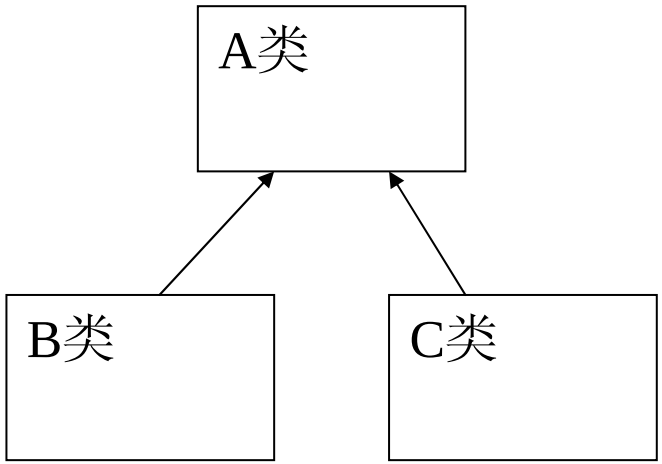
多态性

- 一个父类类型的变量如果引用的实际是子类的对象，那它是可以说是一个多态对象
- 多态对象只能使用父类的成员
- 但如果父类方法被子类覆盖，则调用子类方法也称为虚拟方法调用
  - 例：

```
Employee e1 = new Manager("101","jack","1998-8-1","NEC");
System.out.println(e1.getDetails());//调用的是子类的方法
```

多态的实际用途

- 传递多态参数



假如定义一个方法

```
public void method(A a)
```

那么A类B类C类的实例都可以传进去(多态参数)

否则就要重载三次方法

```
public void method(A a)
public void method(B c)
public void method(C c)
```

instanceof操作符

- 测试对象类型，判断一个引用类型的变量所引用的实例是否是一个类的实例。
- 语法格式： x instanceof A
  - x为引用变量，A为类名或接口名
  - 检验x引用的实例是否是类A的对象实例
  - x引用的实例如果是A类或A的子类或A类的间接子类都返回true
- x引用的实例类型必须是和A类在同一个继承分支上，否则会编译错误

instanceof使用

```
Employee e1 = new Employee();
Manager m1 = new Manager();
System.out.println(e1 instanceof Employee);//true
System.out.println(m1 instanceof Manager);//true
System.out.println(m1 instanceof Employee);//true
System.out.println(e1 instanceof Object);//true
System.out.println(e1 instanceof Manager);//false
System.out.println(e1 instanceof Cat);//编译错误
```

对象类型转换

- 引用数据类型的转换
  - 自动转换：从子类到父类的类型可以自动转换
  - 强制转换：从父类到子类的类型要强制转换
- 例：

```
Employee e1 = new Manager();//自动转换
Manager m1 = (Manager) e1;//强制转换
```

- 无继承关系的类型转换是非法的
- 强制转换的实际是对象的对象列表而并非是对象的实例

Object类

- Object类是所有java类的根类
- 如果类的声明中没有显式的定义extends继承某个类则默认继承Object类
- 例如：
  - public class Employee
  - 等同于
  - public class Employee extends Object
- 任何引用类型的实例都可以赋给Object类型的引用变量
  - Object obj = 任何引用类型的实例

== 和 equals方法的使用

- ==操作符
  - 基本类型 比较值
  - 引用类型 比较的是引用的地址（是否引用的是同一个实例）
  - ==两边的操作元类型必须保持一致或可自动转换，否则编译出错
- equals方法
  - 由根类Object提供，在Object中实现的功能和==相同
  - 格式：obj1.equals(obj2)返回boolean类型值
  - 有一些类已经覆盖了equals方法：String 、File 、Date，以及所有的包装类
  - 可以在自定义类中覆盖equals方法，但同时也应该覆盖hashcode方法

覆盖equals方法举例

```
public class MyPoint {
    protected int x;
    protected int y;
    ......
    public boolean equals(Object obj){
        if (obj == null) return false;//如果为空返回假
        if (this == obj) return true;//如果是用一个实例返回真

        if (!(obj instanceof MyPoint)) return false;//如果不是相同
    类型返回假
        MyPoint other = (MyPoint) obj;//强制转换为当前类型

        //进行自定义比较
        if (this.x == other.x && this.y == other.y){
                return true;
        }
        return false;
    }
```

toString方法

- 由Object类提供，在Object类中实现的功能是返回“完整类名@16进制哈希码”
- 在打印时如果传入的是引用类型，则默认调用此引用类型的toString()方法
- 在和字符串做加法运算时，也默认调用引用类型的toString()方法
- 可以覆盖toString()方法,将类中数据连接后返回给用户方便易懂的信息

包装类(wrapper类)

- 主要作用将基本类型数据包装为引用类型,将基本类型用类表示
- 8种基本类型都有对应的包装类

基本类型	包装类
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double

包装类的装箱和拆箱

- 装箱：将基本类型转换为包装类引用类型
- 拆箱：从包装类引用类型中获取基本类型
- 从JDK1.5之后支持自动装箱和拆箱
- 装箱举例：
  - int i = 10;
  - Integer intObj = new Integer(i);
  - 自动装箱：Integer intObj = 10;
- 拆箱举例：
  - i = intObj.intValue();
  - 自动拆箱：int i = intObj;

包装类类型转换举例

- 可以转换数据类型
- 例如： int型转为String类型

```
int i = 1;
Integer objInt = new Integer(i);
String s1 = objInt.toString();
```

- 例如：String类型转换为int型

```
String s = “198”;
int i = Integer.parseInt(s);
```

本章结束

第七章 高级类特性

static修饰符

- static在类中可以做属性、方法和内部类的修饰符
- static修饰的属性和方法和具体实例无关，属于类本身，可以被所有实例共享
- static修饰的属性和方法也称为静态属性和方法（也称为类属性和类方法），在访问控制允许的情况下，可以用类名.属性或类名.方法的形式访问

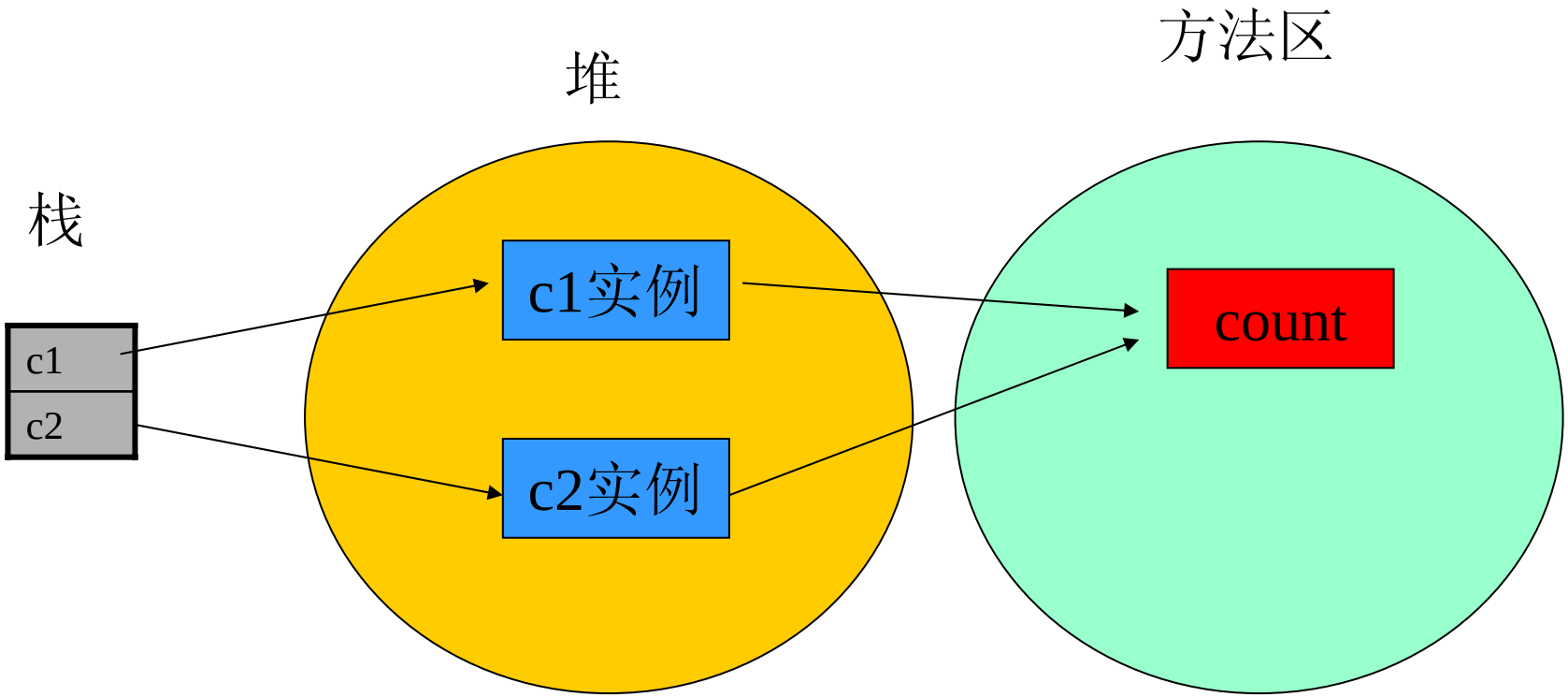
类属性

- 类似全局变量，在内存中只有一份，由该类所有实例共享
- 例如：

```
class Count{
    static int count;
    public Count(){
        count++;//每次创建新的实例就将count加1
    }
    public static int getCount(){
        return count;
    }
}
```

```
public class TestStatic {

    public static void main(String[] args) {

        Count c1 = new Count();

        System.out.println(Count.getCount());//打印1

        Count c2 = new Count();

        System.out.println(Count.getCount());//打印2
    }
}
```

类属性图例

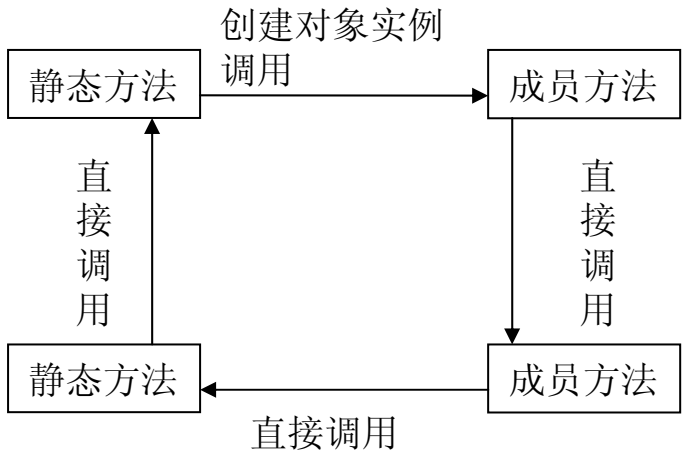


类方法

```
class Count{
    static int count;
    public Count(){
        count++;//每次创建新的实例就将count加1
    }
    public static int getCount(){
        return count;
    }
}
public class TestStatic {
    public static void main(String[] args) {
        Count c1 = new Count();
        System.out.println(Count.getCount());//直接用类名访问

        Count c2 = new Count();
        System.out.println(Count.getCount());//直接用类名访问
     }
}
```

类方法

- 在静态方法内部只能直接访问静态成员，不能直接访问实例成员（非静态成员），不能使用this关键字，因为无法确定是哪一个实例的成员。但可以创建实例调用非静态成员



static静态代码块

- 在类中,方法的外部定义
- static代码块仅会被执行一次(当类被载入时)
- 一个类中可以定义多个static代码块,按出现位置按顺序执行
- static代码块经常用于初始化静态属性

```
class Count{
    static int count;
     static{
       count = 10;
     }
}
```

单子设计模式Singleton

```
public class TestSingleton {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2);
    }
}
class Singleton{
    private static Singleton s = new Singleton();
    private Singleton(){
    }
    public static Singleton getInstance(){
        return s;
    }
}
```

静态导入语句

- 可以将类，接口，枚举中的静态属性或静态方法导入，在使用这些属性和方法的时候无需使用“类型.成员名称”，可以在当前类中直接使用成员名称
- 语法格式：
  - import static <顶层包名>.[<子包名>].<类名>.<属性名|方法名|*>
- 以上语法也适用于导入接口中或枚举中的静态成员

静态导入语句举例

```
package c;

public class UtilMath {

	public static double PI = 3.1415927;

	public static double getArea(double r){
		return PI * r * r;
	}

}
```

```
package d;

import static c.UtilMath.*;

public class Test {

	public static void main(String[] args) {

		System.out.println(PI);

		System.out.println(getArea(10));

	}

}
```

final修饰符

- 可以修饰类、方法和变量。
  - 用final修饰的类不能被继承
  - 用final修饰的方法不能被子类覆盖，但不能修饰构造方法，private修饰的方法默认就是final的
  - 用final修饰的变量表示常量，不能重复赋值。只能在声明时同时赋值或者声明时不赋值但在每个构造方法中显式赋值

final应用举例

```
public class TestFinal {
    public static void main(String[] args) {

        MyStudent s1 = new MyStudent();
        System.out.println(s1.ID);//打印1

        MyStudent s2 = new MyStudent();
        System.out.println(s2.ID);//打印2

        s2.ID = 3;//编译错误,不能重新赋值
    }
}
class MyStudent {
    private static int count;
    public final int ID; //常量

    public MyStudent(){
        ID = ++count;
    }
}
```

abstract关键字

- 可以修饰类和方法：abstract类称为抽象类 abstract方法称为抽象方法
- 抽象类：用来模型化那些那些功能无法全部实现，留给子类去实现的类。
- 抽象方法：不提供方法的具体实现，没有{}

抽象类

- 有抽象方法的类一定是抽象类
- 抽象类不一定有抽象方法
- 抽象类不可以被实例化，但可以有构造方法（子类可以调用抽象父类的构造方法来初始化父类的数据）
- 抽象类可以继承抽象类或非抽象类，单继承
- 抽象类和抽象方法不能被final修饰
- 子类如果没有覆盖且实现所有的抽象方法，那子类也必须是一个抽象类

抽象类举例1

```
//抽象类 动物类
abstract class Animal{

    private int leg;
    public Animal(int leg){
        this.leg = leg;
    }
    public void eat(){
        System.out.println("吃东西");
    }
    public abstract void run();//抽象方法
}
```

抽象类举例2

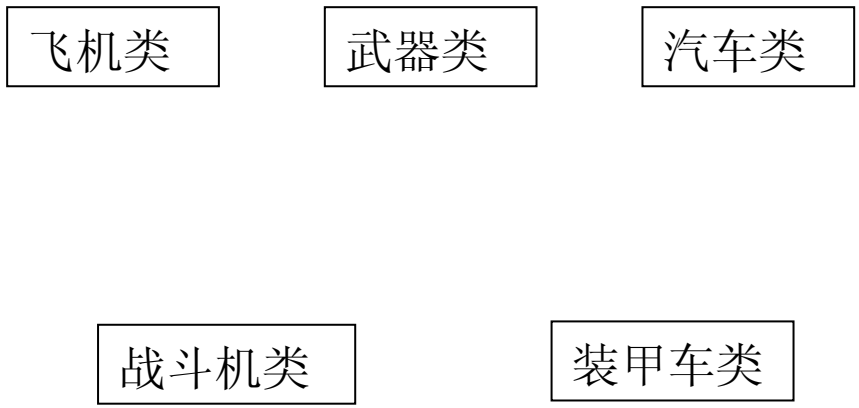
```
//子类 狗
class Dog extends Animal{
	public Dog(int leg){
		super(leg);
	}
	//实现抽象方法
	public void run(){
		System.out.println("撒腿跑");
	}
}
//子类 鱼
class Fish extends Animal{
	public Fish(int leg){
		super(leg);
	}
	//实现抽象方法
	public void run(){
		System.out.println("游来游去");
	}
}
```

修饰符搭配使用注意事项

- 以下修饰符的连用是无意义的，会导致编译错误
  - private和abstract
  - final和abstract
  - static和abstract

接口interface

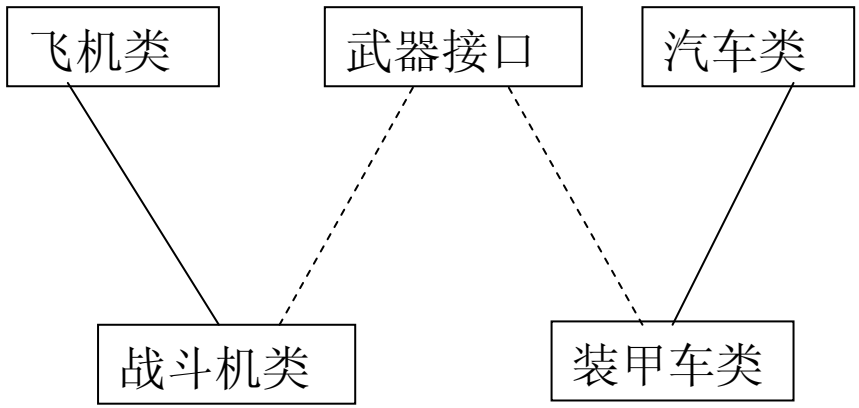
- 由于java中一个类只能有一个父类,所以如果一个类既有一中特性又有另一种特性,则无法用继承关系很好的描述。



战斗机既是一种飞机又是一种武器，而装甲车既是一种汽车也是一种武器

接口interface

- java中提供了接口（interface），一个类可以继承一个父类，同时实现多个接口



如图：可以让战斗机继承飞机类，实现武器接口，而装甲车类可以继承汽车类，而实现武器接口

接口的语法格式

- 声明定义接口
  - 〈修饰符〉 interface 接口名{.....}
- 类实现接口
  - 〈修饰符〉 class 类名 [extends 父类] implements 接口1,接口2,.....
- 例如：

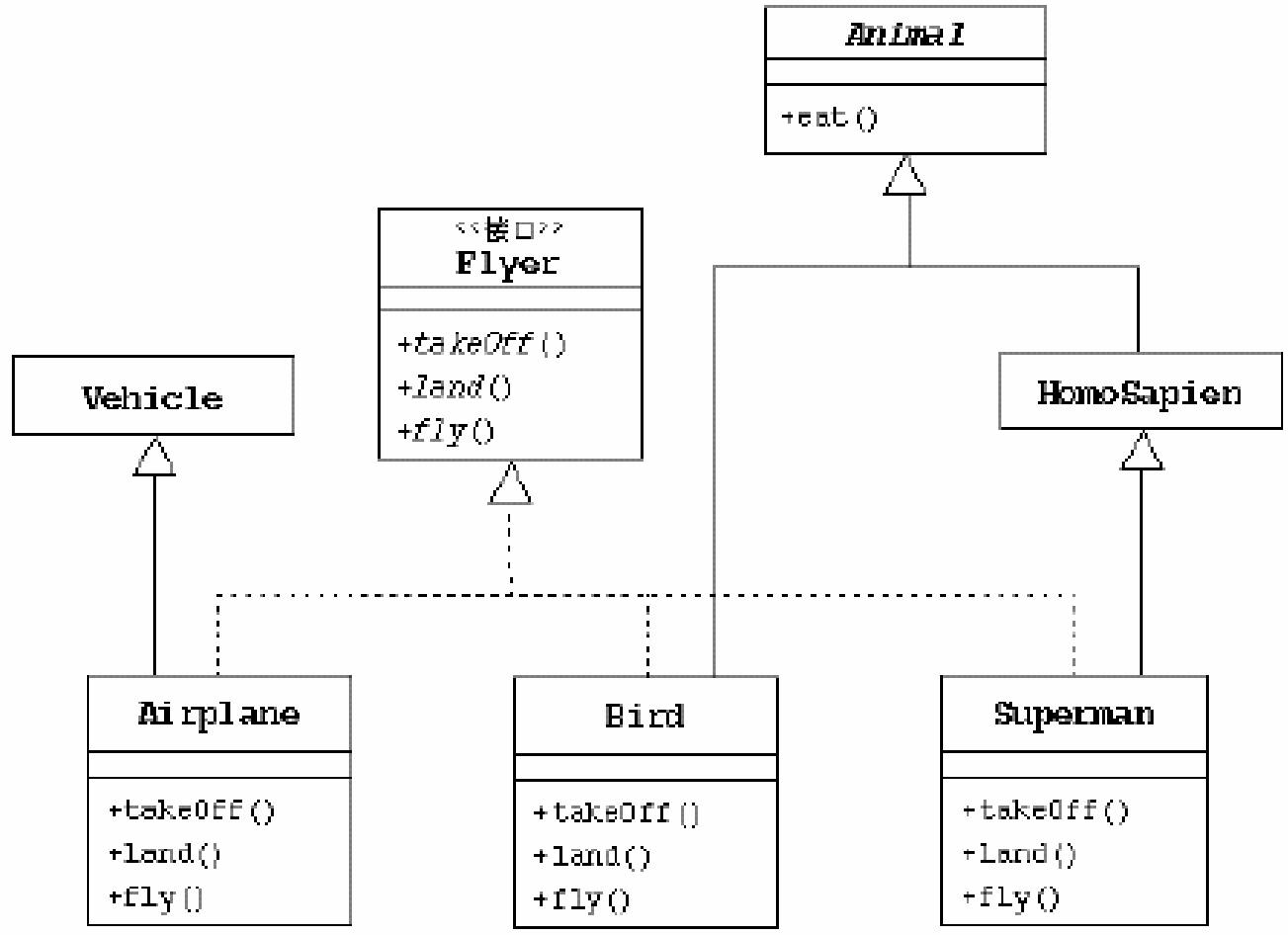
```
//定义接口
public interface Runner{
        void run();
}
```

```
//定义类实现接口
public class Person implements Runner{
        void run(){
                ……
        }
}
```

接口的中的成员

- 成员变量都是public static final修饰，不可以用其他修饰符
- 成员方法都是用public abstract修饰，不可以用其他修饰符，例如 protected, static都不可以
- 其中public static final和public abstract可以省略不写
- 接口中没有构造方法，不能被实例化

接口示例


Animal
+eat()
<<接口>>
Flyer
+takeOff()
+land()
+fly()
Vehicle
HomoSapien
Airplane
+takeOff()
+land()
+fly()
Bird
+takeOff()
+land()
+fly()
Superman
+takeOff()
+land()
+fly()

接口和类的关系

- 一个类实现了某个接口，这个类可以称为是这个接口的实现类
- 一个类可以同时实现多个接口
- 一个类实现了某个接口，必须覆盖实现所有的抽象方法，否则这个类必须定义为抽象类
- 接口之间不能实现，但可以继承，一个接口可以继承多个父接口，称为复合接口
  - 格式：extends 父接口1，父接口2，……
- 一个接口的实现类的实例可以赋给一个接口类型（多态）

接口的设计思想

- 在接口中声明方法，该方法由一个或多个类实现
- 显示对象的编程接口而不暴露类的实际主体
- 捕获无关联类之间的相似性，而无须加强类之间的关系
- 通过声明一个实现若干接口的类来模拟多继承

内部类

- 一个类声明定义在另一个类中就可以称为内部类
- 按照作用域可以分为
  - 成员内部类
    - 实例内部类
    - 静态内部类
  - 局部内部类
- 内部类和外部类名称不能重复，以下内部类的完整类名是A.B

```
public class A{  //外部类
   public class B{ //实例内部类
   }
}
```

实例内部类

- 定义在方法外部没有static修饰的内部类称为实例内部类
- 在外部类中可以直接创建内部类的对象
- 如果在其他类中创建内部类的对象，必须先创建外部类的对象实例，然后用外部类的实例去创建内部类的对象实例
- 内部类可以直接访问所有外部类的成员，外部类若想访问内部类的成员，必须建立内部类的对象实例，通过实例访问
- 在内部类中不能定义静态成员，只能定义实例成员
- 如果内部类中的成员或局部变量和外部类的成员同名
  - 外部类名.this.成员名 表示外部类的成员
  - this.成员名 表示内部类的成员

内部类举例

```
class A { //外部类
    int v1;
    int v2;
    void m1(){
        B b = new B();
        b.v1 = 100;
    }

    class B{ // 内部类
        int v1;
        static int v2; //非法
        public B(){
            A.this.v1 = 10;
            v2 = 20;
            this.v1 = 30;
        }
    }
}
```

```
public class TestA {

public static void main(String[] args){
        A a = new A();
        A.B b = a.new B();
        b.v1 = 1000;
    }
}
```

静态内部类

- 在方法外部用static修饰的内部类称为静态内部类
- 静态内部类中可以直接访问外部类的静态成员，如果要访问实例成员，必须通过实例访问
- 在其他类中创建静态内部类的对象时，可以直接用new创建，不必通过外部类实例
- 在静态内部类中可以定义静态和非静态成员
- 其他类可以通过完整类名直接访问静态内部类的静态成员

静态内部类举例

```
class A { //外部类
    int v1;
    int v2;
    void m1(){
        B b = new B();
        b.v1 = 100;
    }

    static class B{ //静态内部类
        int v1;
        static int v2;

        public B(){
            v1 = 30;
        }
    }
}
```

```
public class TestA {
    public static void main(String[] args){
        A.B b = new A.B();
        b.v1 = 100;
        A.B.v2 = 200;
    }
}
```

局部内部类

- 在一个方法内部定义的内部类称为局部内部类
- 类只在方法内部可见，方法外无法使用
- 类似局部变量，不能用 private,protected,public,static修饰
- 类中不能包含静态成员
- 可以直接访问所有外部类的成员和方法内final类型的参数和变量

局部内部类举例

```
class A { //外部类
    int v1;
    int v2;

    void m2(final int p1){ //方法
        int v3 = 10;

        class C{ //局部内部类
            int c1 = v1;
            int c2 = p1;
            int c3 = v3;//非法
        }
    }
}
```

匿名类

- 匿名类是一种没有名字的特殊内部类
- 可以定义在方法内部，也可以定义在方法外部
- 匿名类本身没有构造方法，但是可以调用父类的构造方法
- 例如：

```
A a = new A(){
    ……
};
```

  - 如果A是一个类或抽象类，则相当于定义并创建了一个A类的子类的实例并赋给A类型的变量
  - 如果A是一个接口，则相当于定义并创建了一个实现A接口的类的实例并赋给A类型的变量

匿名类举例

```
public class TestA {

    public void m1(){
          System.out.println("m1方法");
    }

    public static void main(String[] args){
          //匿名类
          TestA a = new TestA(){
                    public void m1(){
                              System.out.println("匿名类覆盖m1方法");
                    }
          };

          a.m1();//打印：匿名类覆盖m1方法
    }
}
```

枚举(enum)

- 如果有的变量只有几种可能的取值，例如表示性别的男和女，固定的12个月份，为了提高描述问题的直观性，可以使用枚举类型
- 枚举中存储了N个枚举静态常量，每一个都是本身都是枚举类型
- 枚举类型默认都继承了java.lang.Enum类型
- 可以用 “枚举类型.枚举常量”的形式来使用枚举
- 声明一个枚举的语法

  [<修饰符>] enum <枚举类型名>{

  枚举常量1，枚举常量2，……;

  }

枚举的使用

声明一个枚举:

```
public enum Sex {
    BOY,GIRL;
}
```

使用枚举：

```
Sex s = Sex.BOY;
if (s == Sex.BOY)
    System.out.println("男");
else
    System.out.println("女");
```

枚举的使用2

声明枚举

```
public enum Color {
	RED,GREE,BLUE;
}
```

使用枚举

```
Color c = Color.GREEN;

switch (c){
	case RED:
		System.out.println("红");
		break;
	case GREEN:
		System.out.println("绿");
		break;
	default:
		System.out.println("蓝");
		break;

}
```

枚举类型的常用方法

- values() 返回所有的枚举的数组
  - 例如 Color[] colors = Color.values();
- valuesOf(String 字符串) 利用字符串动态返回一个枚举
  - 例如 Color c = Color.valueOf("BLUE");
- ordinal() 返回此枚举的顺序，默认从0开始
- equals() 等同于 ==

本章结束

第八章 异常处理

- 在一些传统语言中，程序员要通过一些特殊的方法返回值来判断各种异常情况
- 在java中提供异常处理机制，可以自动捕获发生的异常并处理，如数组越界，访问文件不存在，数据库错误等
- Java程序运行过程中所发生的异常事件可分为两类：
  - 错误(Error):JVM系统内部错误、资源耗尽等严重情况
  - 异常(Exception): 其它因编程错误或偶然的外在因素导致的一般性问题，例如：
    - 对负数开平方根
    - 空指针访问
    - 试图读取不存在的文件
    - 网络连接中断

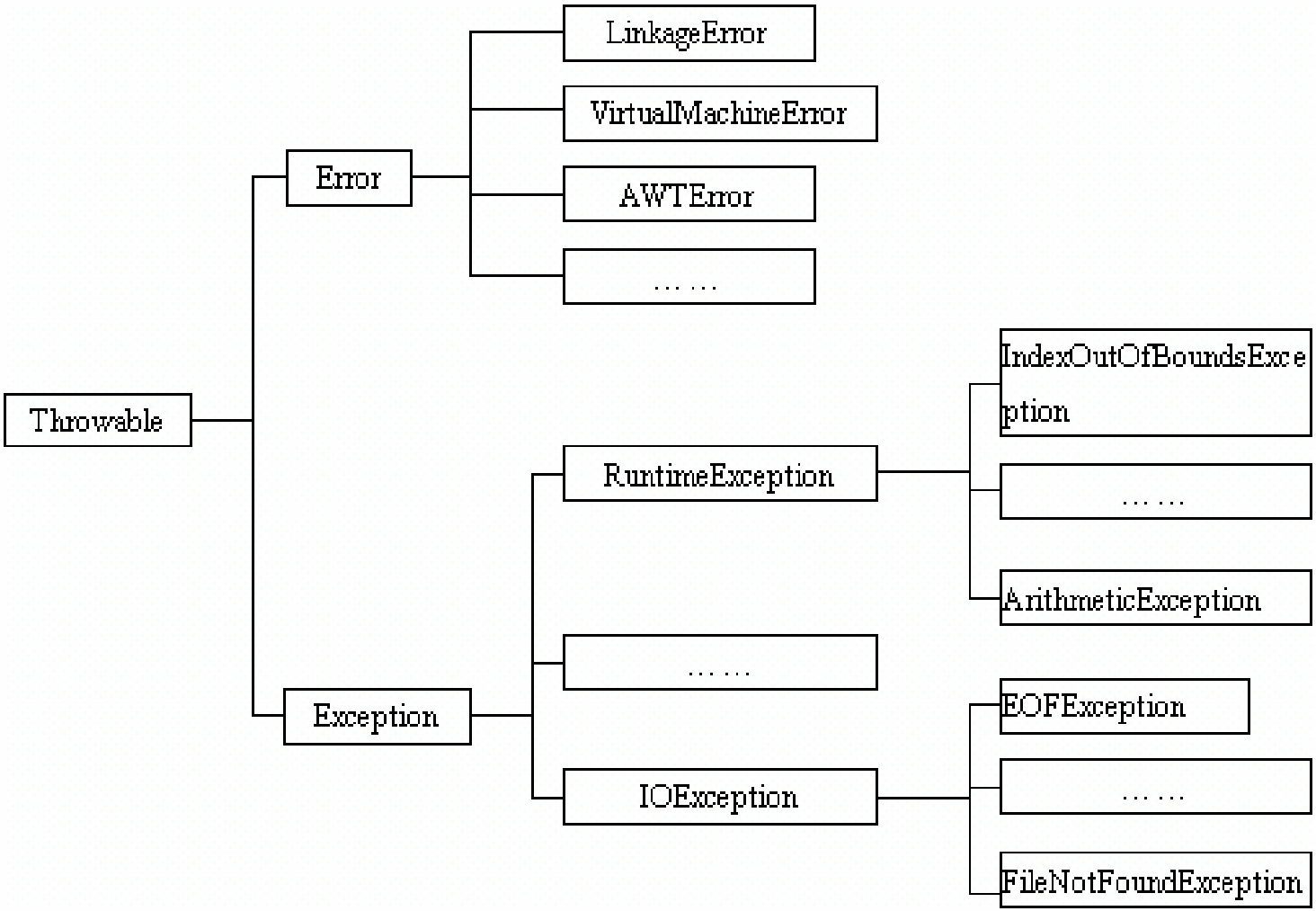
产生异常举例

```
public class Test8_1{
        public static void main(String[] args) {
            String friends[]={"lisa","bily","kessy"};
            for(int i=0;i<4;i++)  {
                        System.out.println(friends[i]); //没有friends[3]
            }
            System.out.println("\nthis is the end");
    }
}
```

通过编译，输出结果：

```
lisa
bily
kessy
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
at exception.Test8_1.main(Test8_1.java:7)
```

java异常类层次


Throwable
Error
LinkageError
VirtualMachineError
AWTError
……
Exception
RuntimeException
IndexOutOfBoundsExce
ption
……
ArithmeticException
……
IOException
EOFException
……
FileNotFoundException

常见异常

- RuntimeException
  - 错误的类型转换
  - 数组下标越界
  - 空指针访问
- IOExeption
  - 从一个不存在的文件中读取数据
  - 越过文件结尾继续读取
  - 连接一个不存在的URL

异常处理机制

- Java程序的执行过程中如出现异常，会自动生成一个异常类对象，该异常对象将被提交给Java运行时系统，这个过程称为抛出(throw)异常。
- 当Java运行时系统接收到异常对象时，会寻找能处理这一异常的代码并把当前异常对象交给其处理，这一过程称为捕获(catch)异常。
- 如果Java运行时系统找不到可以捕获异常的方法，则运行时系统将终止，相应的Java程序也将退出。
- 程序员通常只能处理异常(Exception)，而对错误(Error)无能为力

异常处理举例（1）

```
public class Test8_1 {
    public static void main(String[] args) {
        String friends[] = { "lisa", "bily", "kessy" };
        try {
            for (int i = 0; i < 4; i++) {
                System.out.println(friends[i]);
            }
        } catch (java.lang.ArrayIndexOutOfBoundsException e) {
            System.out.println("index err");
        }
        System.out.println("\nthis is the end");
    }
}
```

异常处理举例（2）

- 程序java8_2运行结果：java java8_2

lisa
bily
kessy
index err

this is the end

捕获异常

- 捕获异常是通过try-catch-finally语句实现的。

```
try{
  ......        //可能产生异常的代码
}catch( ExceptionName1 e ){
  ......        //当产生ExceptionName1型异常时的处置措施
}catch( ExceptionName2 e ){
...... //当产生ExceptionName2型异常时的处置措施
}  [ finally{
......  //无条件执行的语句
}  ]
```

捕获异常

- try
  - 捕获异常的第一步是用try{...}语句块选定捕获异常的范围。
- catch
  - 在catch语句块中是对异常对象进行处理的代码，每个try语句块可以伴随一个或多个catch语句，用于处理可能产生的不同类型的异常对象。与其它对象一样，可以访问一个异常对象的成员变量或调用它的方法。
    - getMessage( ) 方法，用来得到有关异常事件的信息
    - printStackTrace( )用来跟踪异常事件发生时执行堆栈的内容。

捕获异常

- finally语句块
  - 捕获例外的最后一步是通过finally语句为例外处理提供一个统一的出口，使得在控制流转到程序的其它部分以前，能够对程序的状态作统一的管理。不论在try代码块中是否发生了异常事件，finally块中的语句都会被执行。
  - finally语句是任选的

不捕获异常时发生的情况

- 分为两种情况
  - 如果是运行时异常（RuntimeException类及它的子类）,即使没有显式的try catch捕获，发生异常时java可以自动捕获做默认处理，但程序会终止
  - 如果是检查异常（除RuntimeException以外所有的异常），则必须显式捕获或向外抛出，否则通不过编译

IOException异常处理举例(1)

```
public class Test8_3 {
    public static void main(String[] args) {
        FileInputStream in = new FileInputStream("myfile.txt");
        int b;
        b = in.read();
        while (b != -1) {
            System.out.print((char) b);
            b = in.read();
        }
        in.close();
    }
}
```

IOException异常处理举例(2)

程序Test8_3编译结果: E:\ex\javac Test8_3.java

```
E:\ex\Test8_3.java:4: 未报告的异常 java.io.FileNotFoundException ；
    必须被捕获或被声明抛出
            FileInputStream in=new FileInputStream("myfile.txt");
                              ^
E:\ex\Test8_3.java:6: 未报告的异常 java.io.IOException ； 必须被捕获
    或被声明抛出                b = in.read();
                                  ^
E:\ex\Test8_3.java:9: 未报告的异常 java.io.IOException ； 必须被捕获
    或被声明抛出   b = in.read();
                               ^
E:\ex\Test8_3.java:11: 未报告的异常 java.io.IOException ； 必须被捕获
    或被声明抛出  in.close();
                              ^
4 个错误
```

IOException异常处理举例(3)

```
import java.io.*;
public class Test8_4 {
    public static void main(String[] args) {
        try {
            FileInputStream in = new FileInputStream("myfile.txt");
            int b;
            b = in.read();
            while (b != -1) {
                System.out.print((char) b);
                b = in.read();
            }
            in.close();
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            System.out.println(" It’s ok!");
        }
    }
}
```

向外抛出异常throws

- 如果一个方法内可能会出现异常，但当前方法不明确如何处理这个异常，则可以将这个异常抛向方法的调用者
  - 格式：方法后 throws 异常类1，异常类2，.....
- 方法的调用者如果无法处理这个异常，可以继续向上抛出
- 一般向外抛出的都是检查异常

覆盖方法关于异常的规则

- 覆盖的方法不能比被覆盖的方法抛出更多的异常

```
public class A { //父类
    public void methodA() throws IOException {
        ……
    }
}
public class B1 extends TestA {
    public void methodA() throws FileNotFoundException { //合法
        ……
    }
}
public class B2 extends TestA {
    public void methodA() throws Exception { //非法
        ……
    }
}
```

手动抛出异常

- Java异常类对象除在程序执行过程中出现异常时由系统自动生成并抛出，也可根据需要需要人工创建并抛出
  - 首先要生成例外对象，然后通过throw语句实现抛出操作(提交给Java运行环境)。
    - IOException e =new IOException();
    - throw e;
  - 可以抛弃的例外必须是Throwable或其子类的实例。下面的语句在编译时将会产生语法错误：
  - throw new String("want to throw");

创建用户自定义异常类

- 用户自定义的异常类必须继承现有的异常类
- 用户自定义例外类MyException，用于描述数据取值范围错误信息：

```
class MyException extends Exception {
        private int idnumber;
   public MyException(String message, int id) {
        super(message);
        this.idnumber = id;
   }
   public int getId() {
        return idnumber;
   }
}
```

使用用户自定义异常类

```
public class Test8_6 {
    public void regist(int num) throws MyException {
        if (num < 0) {
            throw new MyException("人数为负值，不合理", 3);
            System.out.println("登记人数" + num);
        }
    }
    public void manager() {
        try {
            regist(100);
        } catch (MyException e) {
            System.out.print("登记失败，出错种类" + e.getId());
        }
        System.out.print("本次登记操作结束");
    }

    public static void main(String args[]) {
        Test8_6 t = new Test8_6();
        t.manager();
    }
}
```

本章结束

第九章 反射和标注

反射的作用

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的属性和方法。
- 在运行时调用任意一个对象的方法。
- 构建动态代理。

反射相关的类

- Class类——代表一个类。
- Field类——代表类的属性。
- Mehtod类——代表类的方法。
- Constructor类——代表类的构造器

一个简单反射的例子

```
package a;
public class ObjImpl implements ObjInterface{
	public String getGreeting(String name) {
		return "hello," + name;
	}
}

package a;
public class TestReflect {
	public static void main(String[] args) throws ClassNotFoundException,
				InstantiationException, IllegalAccessException {
		Class objClass = Class.forName("a.ObjImpl");
		ObjImpl obj = (ObjImpl) objClass.newInstance();
		System.out.println(obj.getGreeting("tom"));

	}
}
```

标注

- 标注Annotation（注释，批注）提供在程序中与程序元素关联的任何信息或者元数据的途径。
- 可以应用在类，构造器，方法，属性中

自定义标注

[<修饰符>] @interface <标注名>{
    返回值 方法名称（）；
    返回值 方法名称（）；
    ……
}

- 方法的返回值必须是：基本类型，String，Class，枚举，Annotation和它们组成的数组

标注的标注

- @Retention(RetentionPolicy.RUNTIME) 表示JVM运行时此标注存在
- @Retention(RetentionPolicy.CLASS) 表示仅在class文件中存在，程序运行无法读取
- @Retention(RetentionPolicy.SOURCE) 表示仅在源文件中存在，编译之后会丢失

- @Target(ElementType.METHOD) 表示仅能修饰方法
- @Target(ElementType.FIELD) 表示仅能修饰属性
- @Target(ElementType.TYPE) 表示仅能修饰类

自定义标注示例

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyComment {
    int id();
    String info();
}
```

在程序中使用标注

```
public class Test {
    @MyComment(id = 1, info = "m1方法")
    public void m1() {

    }

    @MyComment(id = 2, info = "m2方法")
    public void m2() {

    }
}
```

利用反射读取标注

```
import java.lang.reflect.*;
public class TestAnnotation {

	public static void main(String[] args) throws ClassNotFoundException {

		Class c = Class.forName("g.Test");
		Method[] methods = c.getDeclaredMethods();

		for (Method m : methods) {
			if (m.isAnnotationPresent(MyComment.class)) {
				MyComment myComment = m.getAnnotation(MyComment.class);
				System.out.println(m.getName() + ":" + myComment.id() + ","
						+ myComment.info());
			}
		}

	}
}
```

内置标注

- @Override——用于修饰此方法覆盖了父类的方法，而非重载。
- @Deprecated——用于修饰已经过时的方法。
- @SuppressWarnings——用于通知Java编译器禁止特定的警告。

利用反射实现动态代理

- 动态代理的主要目的是当调用某一个类的方法时对这个方法进行拦截，以便追加一些额外操作或者改变方法的操作
- 动态代理只能针对于接口类型，被代理的类必须实现某一个XX接口

- 代理功能举例：例如调用某个类的所有方法都提前打印一句”welcome”

代理举例：HelloWorld接口

```
public interface HelloWorld {
    public abstract void print();
    public abstract void say();
}
```

代理举例：HelloWorldImpl实现类

```
public class HelloWorldImpl implements HelloWorld{
    public void print() {
        System.out.println("hello,world!");
    }

    public void say() {
        System.out.println("say hello");
    }
}
```

代理举例：DynamicProxy代理类

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class DynamicProxy implements InvocationHandler {

    private Object object;

    // 自定义方法,返回一个被代理类对象实例
    public Object bind(Object object) {
        this.object = object;
        return Proxy.newProxyInstance(object.getClass().getClassLoader(),
                                      object.getClass().getInterfaces(), this);
    }

    // 覆盖拦截方法，在方法被调用的时候进行拦截，追加其他操作
    public Object invoke(Object proxy, Method method, Object[] args)
                        throws Throwable {
        System.out.println("welcome");// 方法之外的额外操作
        Object result = method.invoke(object, args); //调用方法
        return result;// 返回值
    }
}
```

代理举例：Test测试类

```
public class Test {
    public static void main(String[] args) {
        HelloWorld hw1 = new HelloWorldImpl();
        HelloWorld hw2 = new HelloWorldImpl();

        DynamicProxy dp = new DynamicProxy();//动态代理类
        hw1 = (HelloWorld) dp.bind(hw1);//传入一个普通类对象，返
    一个被代理类对象

        hw1.print();//有输出welcome
        hw2.print();//无输出welcome
    }
}
```

第十章

文件操作和IO输入输出

文件操作File类

- File类既可以操作文件,也可以操作目录
- 创建File类对象

```
File f;
f = new File("Test.java");
f = new File("E:\\ex\\","Test.java");
```

- 在Java中，将目录也当作文件处理File类中提供了实现目录管理功能的方法

```
File path = new File("E:\\ex\\");
File f = new File(path, "Test.java");
```

File类方法介绍

- 关于文件/目录名操作
- String getName()
- String getPath()
- String getAbsolutePath()
- String getParent()
- boolean renameTo(File newName)

- File 测试操作
- boolean exists()
- boolean canWrite()
- boolean canRead()
- boolean isFile()
- boolean isDirectory()
- boolean isAbsolute();

- 获取常规文件信息操作
- long lastModified()
- long length()
- boolean delete()

- 目录操作
- boolean mkdir()
- boolean mkdirs()
- String[] list()
- File[] listFiles()

I/O数据流输入输出

- 在java.io包中提供了对数据流输入输入的类
- 可分为两组
  - 按字节为单位处理数据流的InputStream和OutputStream
  - 按字符为单位处理数据流的Reader和Writer
  - 它们都是抽象类，常用的子类有FileInputStream,ObjectInputStream,FileOutputStream,ObjectOutputStream,InputStreamReader,FileReader,BufferedReader,OutputStreamWriter,FileWriter,BufferedWriter

文件I/O有关类型

- 文件输入
  - 可使用FileReader类以字符为单位从文件中读入数据
  - 可使用BufferedReader类的readLine方法以行为单位读入一行字符
- 文件输出
  - 可使用FileWriter类以字符为单位向文件中写出数据
  - 使用PrintWriter类的print和println方法以行为单位写出数据

文件输入举例

```
import java.io.*;
public class Test9_4 {
    public static void main (String[] args) {
                String fname = "Test9_4.java";
                File f = new File(fname);
                try {
                        FileReader fr = new FileReader(f);
                        BufferedReader br = new BufferedReader(fr);
                        String s = br.readLine();
                        while ( s != null ) {
                                System.out.println("读入: " + s);
                                s = br.readLine();
                        }
                        br.close();// 关闭缓冲读入流及文件读入流的连接.
                } catch (FileNotFoundException e1) {
                        System.err.println("File not found: " + fname);
                } catch (IOException e2) {
                        e2.printStackTrace();
                }
        }
 }
```

文件输出举例

```
import java.io.*;
public class Test9_5 {
    public static void main (String[] args) {
        File file = new File("tt.txt");
        try {
            InputStreamReader  is = new InputStreamReader(System.in);
            BufferedReader in=new BufferedReader(is);
            FileWriter fw = new FileWriter(file)
            PrintWriter out  = new PrintWriter(fw);
            String s = in.readLine();
            while (!s.equals("")) {// 从键盘逐行读入数据输出到文件
                out.println(s);
                s = in.readLine();
             }
             in.close();  // 关闭BufferedReader输入流.
             out.close();  // 关闭连接文件的PrintWriter输出流.
         }catch (IOException e) {System.out.println(e);  }
    }
}
```

I/O控制台(Console I/O)

- System.out 提供向“标准输出”写出数据的功能
- System.out为 PrintStream类型.
- System.in 提供从“标准输入”读入数据的功能
- System.in 为InputStream类型.
- System.err提供向“标准错误输出”写出数据的功能
- System.err为 PrintStream类型.

向标准输出写出数据

- System.out/System.err的println/print方法
- println方法可将方法参数输出并换行
- print方法将方法参数输出但不换行
- print和println方法针对多数数据类型进行了重写(boolean, char, int, long, float, double以及char[], Object和 String).
- print(Object)和println(Object)方法中调用了参数的toString()方法，再将生成的字符串输出

从标准输入读取数据

```
import java.io.*;
public class Test9_3 {
        public static void main (String args[]) {
            String s;
            // 创建一个BufferedReader对象从键盘逐行读入数据
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            try {// 每读入一行后向显示器输出
                    s = br.readLine();
                    while (!s.equals("")) {
                            System.out.println("Read: " + s);
                            s = br.readLine();
                    }
                    br.close(); // 关闭输入流

            } catch (IOException e) { // 捕获可能的IOException.
                    e.printStackTrace();
            }
        }
}
```

第十一章 常用类

Math类

- Math类中定义了多个static方法提供常用数学运算功能
  - 截断操作(Truncation): ceil, floor, round
  - 取最大、最小及绝对值: max, min, abs
  - 三角函数: sin, cos, tan, asin, acos, atan, toDegrees,
  - 对数运算: log , exp
  - 其它: sqrt, pow, random
  - 常量: PI, E

String 类

- String 类对象保存不可修改的Unicode字符序列
- String类的下述方法能创建并返回一个新的String对象： concat, replace, substring, toLowerCase, toUpperCase, trim,String.
- 提供查找功能的有关方法：endsWith, startsWith, indexOf,, lastIndexOf.
- 提供比较功能的方法：equals, equalsIgnoreCase, compareTo.
- 其它方法：charAt ，length.

StringBuffer类

- StringBuffer类对象保存可修改的Unicode字符序列
- 构造方法
  - StringBuffer()
  - StringBuffer(int capacity)
  - StringBuffer(String initialString)
- 实现修改操作的方法:
  - append, insert, reverse, setCharAt, setLength

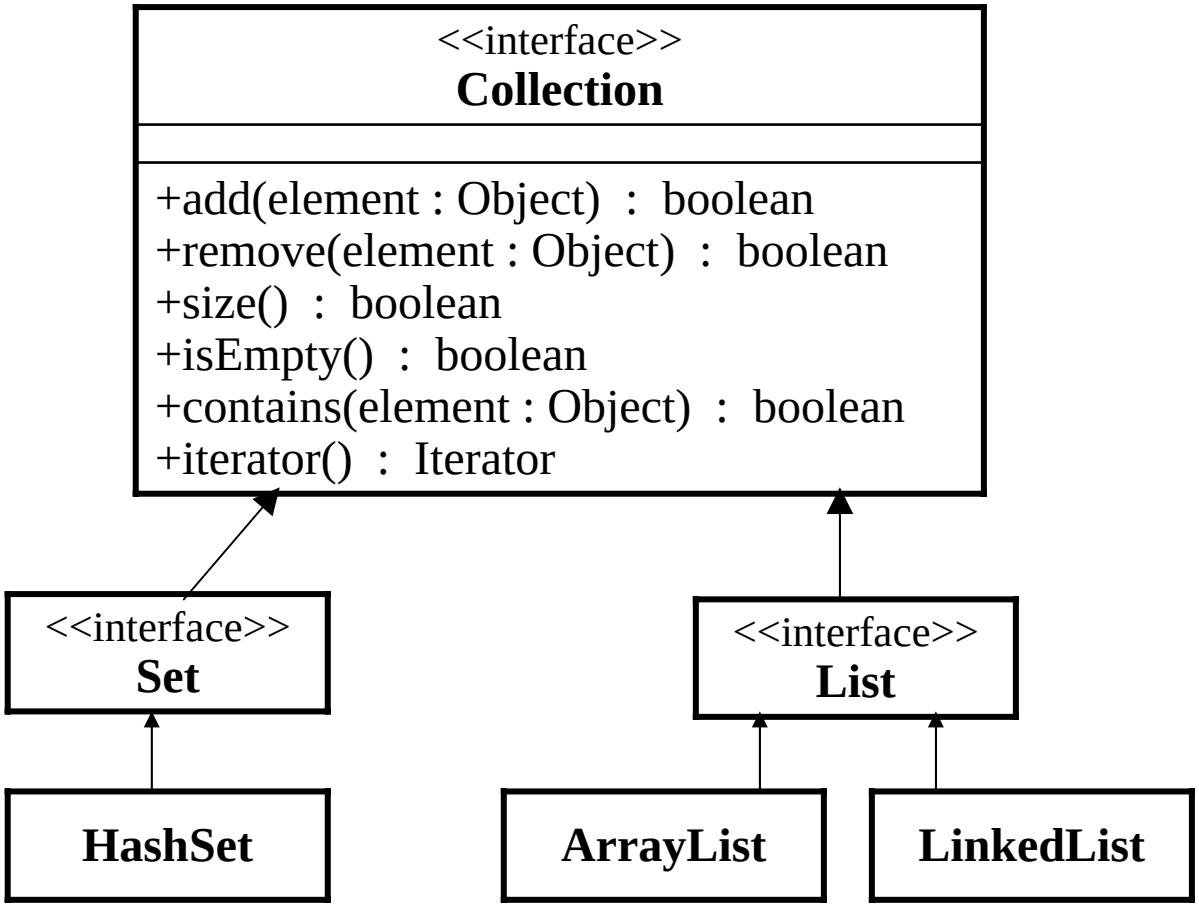
StringBuilder类

- StringBuilder类对象保存可修改的Unicode字符序列
- 功能与StringBuffer相同，但在多线程情况下是线程不安全的

Collection API

- Collection API提供“集合”的功能
- Collection API包含下述接口
  - *Colection*: 将一组对象以集合元素的形式组织到一起，在其子接口中分别实现不同的组织方式
  - *Set*: *Collection*的子接口，不记录元素的保存顺序，且不允许有重复元素
  - *List*: *Collection*的子接口，记录元素的保存顺序，且允许有重复元素

Collection API 层次结构



Set 接口用法举例

```
import java.util.*;
public class Test9_6 {
    public static void main(String[] args) {
          HashSet h = new HashSet();
          h.add("1st");
          h.add("2nd");
          h.add(new Integer(3));
          h.add(new Double(4.0));
          h.add("2nd");       // 重复元素, 未被加入
          h.add(new Integer(3)); // 重复元素, 未被加入
          m1(h);
    }
    public static void m1(Set s){
          System.out.println(s);
    }
}
//本应用程序输出结果如下:[1st, 3, 2nd, 4.0]
```

List 接口用法举例

```
import java.util.*;
public class Test9_7{
    public static void main(String[] args) {
          ArrayList h = new ArrayList();
          h.add("1st");
          h.add("2nd");
          h.add(new Integer(3));
          h.add(new Double(4.0));
          h.add("2nd");       // 重复元素, 加入
          h.add(new Integer(3)); // 重复元素,加入
          m1(h);
    }
    public static void m1(List s){
          System.out.println(s);
    }
}
//本应用程序输出结果如下:[1st, 2nd, 3, 4.0, 2nd, 3]
```

*Iterator*接口

- Iterator接口定义了对Collection类型对象中所含元素的遍历等增强处理功能
  - 可以通过Collection接口中定义的iterator()方法获得一个对应的Iterator(实现类)对象

Iterator接口用法举例

```
import java.util.*;
public class Test9_8 {
   public static void main(String[] args) {
        ArrayList h = new ArrayList();
        h.add("1st");
        h.add("2nd");
        h.add(new Integer(3));
        h.add(new Double(4.0));
        Iterator it = h.iterator();
        while ( it.hasNext() ) {
                System.out.println(it.next());
                }
   }
}
```

Map集合

- Map（映射）是一种把键对象和值对象进行映射的集合，每一个元素都包括一对键和值对象。
- Map本身是个接口，实现类：
  - HashMap（线程不安全）
  - Hashtable (线程安全)

Map举例

```
HashMap map = new HashMap();

//给集合添加元素
map.put("101", "tom");
map.put("102", "jack");
map.put("103", "rose");
map.put("101", "john"); //将会替换101对应的tom

//通过键对象返回值对象
System.out.println(map.get("101")); //打印john
```

系统属性(System Properties)

- 在Java中，系统属性起到替代环境变量的作用(环境变量是平台相关的)
- 可使用System.getProperties()方法获得一个Properties类的对象，其中包含了所有可用的系统属性信息
- 可使用System.getProperty(String name)方法获得特定系统属性的属性值
- 在命令行运行Java程序时可使用-D选项添加新的系统属性

Properties 类

- Properties类可实现属性名到属性值的映射，属性名和属性值均为String类型.
- Properties类的 propertyNames() 方法可以返回以Enumeration类型表示的所有可用系统属性属性名.
- Properties类的 getProperty(String key)方法获得特定系统属性的属性值.
- Properties类的load和save方法可以实现将系统属性信息写入文件和从文件中读取属性信息.

系统属性用法举例

```
import java.util.Properties;
import java.util.Enumeration;
public class Test9_2 {
       public static void main(String[] args) {
   Properties ps = System.getProperties();
   Enumeration pn = ps.propertyNames();
   while ( pn.hasMoreElements() ) {
           String pName = (String) pn.nextElement();
           String pValue = ps.getProperty(pName);
           System.out.println(pName + "----" + pValue);
   }
       }
}
// java -DmyProperty=MyValue Test9_2
```

第十二章 泛型（generic）

- 泛型的主要目标是类型参数化，后期绑定数据类型。在设计类和接口的时候可以定义一个泛型形参，然后在使用类和接口的时候再绑定具体的类型（实参）
- 代替了Object类型，减少类型转换的次数
- 泛型尚不支持基本类型
- 如果类在设计期间使用了泛型，而使用时未绑定任何类型，则默认为Object类型

泛型类

- 声明一个泛型类的语法
  - 修饰符 class 类名 <泛型形参1，泛型形参2，……>{}
- 例如：

```
public class UtilArray <T> {
    //返回数组的最后一个元素
    public T getLastItem(T[] a){
      return a[a.length-1];
    }
}
```

使用泛型类

```
Integer[] intArr = {1,2,3,4,5};
String[] strArr = {"tom","jack","rose"};

UtilArray<Integer> u1 = new UtilArray<Integer>();
System.out.println(u1.getLastItem(intArr));//打印5

UtilArray<String> u2 = new UtilArray<String>();
System.out.println(u2.getLastItem(strArr));//打印rose
```

泛型方法

- 类或接口本身没有声明泛型，而是方法本身声明了泛型
- 泛型方法中的泛型被绑定的类型是默认根据方法的实参来确定的
- 泛型方法的语法：

  修饰符 <泛型类型1，泛型类型2，……> 返回类型 方法名(参数列表){}
- 例如：

```
public class UtilArray2 {
    //返回数组的最后一个元素
    public static <T> T getLastItem(T[] a){
      return a[a.length-1];
    }
}
```

使用泛型方法

```
String[] ss = { "tom", "jack", "rose" };

//显式绑定泛型
String s1 = UtilArray2.<String>getLastItem(ss);

//不显式绑定泛型
String s2 = UtilArray2.getLastItem(ss);
```

声明泛型类的时候添加约束（有界类型）

- 语法形式：

  <T extends 类型A>

  或者

  <T extends 类型A & 类型B & 类型C ……>
- 其中类型A可以是一个类或接口，类型B，C必须是一个接口
- 使用类的时候，绑定的类型必须是已经继承了这些类或实现了这些接口

泛型约束举例

```
public class UtilArray <T extends Number> {

    //返回数组的最后一个元素
    public  T getLastItem(T[] a){
            return a[a.length-1];
    }

    public static void main(String[] args){

            UtilArray<Number> u1 = new UtilArray<Number>();//正确
            UtilArray<Integer> u2 = new UtilArray<Integer>();//正确
            UtilArray<Double> u3 = new UtilArray<Double>();//正确
            UtilArray<String> u4 = new UtilArray<String>();//错误
    }
}
```

使用泛型类的时候加约束

- 有以下几种情况
- 泛型类名<类型A> 表示仅接受绑定了A类型的泛型类对象
- 泛型类名<? extends 类型A> 接受绑定了A类型或A类型子类类型的泛型类对象
- 泛型类名<? super 类型A> 接受绑定了A类型或A类型父类类型的泛型类对象
- 泛型类名<?> 接受任何类型

泛型约束举例

```
ArrayList<Integer> list1 = new ArrayList<Integer>();
ArrayList<Number> list2 = new ArrayList<Number>();
ArrayList<String> list3 = new ArrayList<String>();
ArrayList<Object> list4 = new ArrayList<Object>();

ArrayList<Object> list5;
list5 = list4;//正确
list5 = list3;//错误

ArrayList<? extends Number> list6;
list6 = list1;//正确
list6 = list2;//正确
list6 = list3;//错误
list6 = list4;//错误
```