Java 程序设计

第5章 继承

田英鑫 tyx@hit.edu.cn
哈尔滨工业大学软件学院

第5章 继承

& 本章导读

- 5.1 类、超类和子类
  - 类的继承、继承层次、多态和动态绑定
  - final关键字、抽象类、受保护访问
- 5.2 Object：所有类的超类
- 5.3 范型数组列表
- 5.4 对象包装器与自动打包
- 5.5 参数数量可变的方法
- 5.6 枚举类
- 5.7 继承设计技巧

第5章 继承

& 本章重点

- 5.1 类、超类和子类
- 5.3 范型数组列表
- 5.4 对象包装器与自动打包

& 本章难点

- 5.1 类、超类和子类
  - 多态和动态绑定、抽象类、受保护访问
- 5.3 范型数组列表

5.1.1 类的继承

- 在现有类的基础上构建新类
  - 当一个新类继承自一个现有类时，新类重用（继承）了现有类的方法和字段，同时还可以向新类中增添新的方法和字段以满足新的需求
  - 现有类称为
    - 超类（super class）
    - 基类（base class）
    - 父类（parent class）
  - 新类称为
    - 子类（subclass child class）
    - 派生类（derived class）

5.1.1 类的继承

- 为子类增加新的字段和方法
  - 例如：Manager类派生自Employee类
    - 为Manager类新增了字段用于存放奖金，并为之增加了一个新方法，用来设置奖金的值

```
class Manager extends Employee
{
    ...
    public void setBonus(double b)
    {
       bonus = b;
    }
    private double bonus;
}
```

5.1.1 类的继承

- 方法覆盖（override）
  - 设计类时，应该将通用的方法放在超类中，而将具有特殊用途的方法放在子类中
  - 有时超类中的方法对子类并不合适，可以在子类中覆盖超类中的方法

```
class Manager extends Employee
{
    ...
    public double getSalary()//覆盖超类中的getSalary方法
    {
        ...
    }
    ...
}
```

5.1.1 类的继承

- 方法覆盖（override）
  - 子类覆盖父类的方法时，子类中方法的访问权限不能低于父类中该方法的访问权限

```
class A {
    public int fun(int a,int b) {
        return a + b;
    }
}
class B extends A {
    //fun函数定义产生编译错误，该函数的访问权限不能低于public
    int fun(int a,int b) {
        return a * b;
    }
}
```

5.1.1 类的继承

- super 关键字
  - 在子类中调用父类的方法，使用super关键字
    - 例如，按如下方式实现Manager类的方法getSalary

```
public double getSalary() {
    return salary + bonus;//错误，不能访问父类中的私有字段
}
public double getSalary() {
    double baseSalary = getSalary();//错误，产生死循环
    return baseSalary + bonus;
}
public double getSalary() {
    double baseSalary = super.getSalary();//正确
    return baseSalary + bonus;
}
```

5.1.1 类的继承

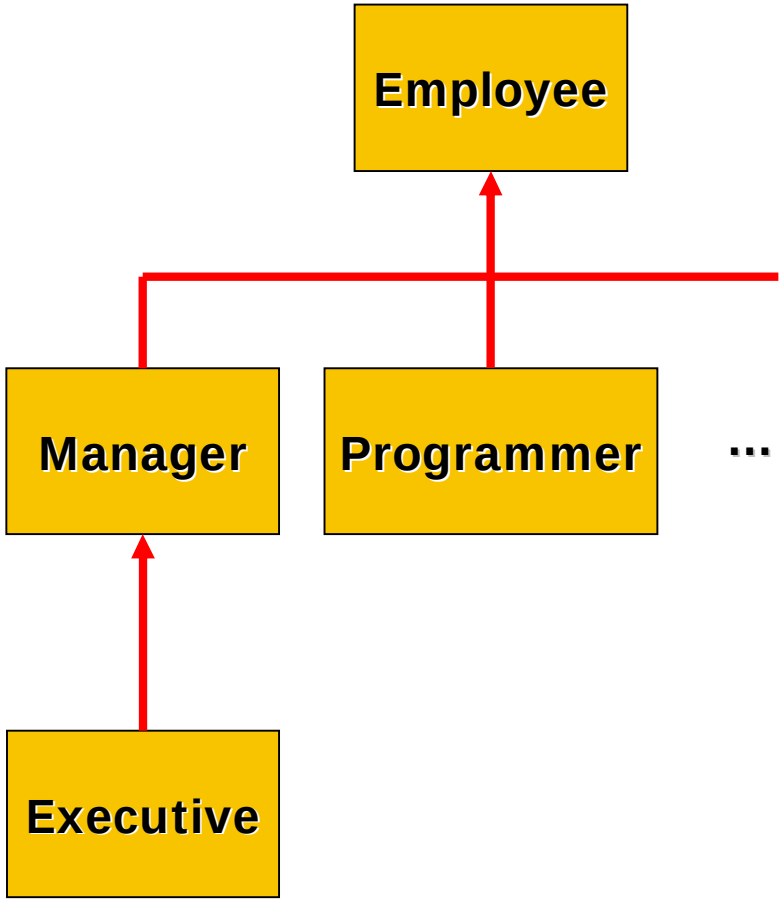
- super 关键字
  - super关键字的另一个作用是在子类的构造器中调用父类的构造器来初始化父类中的私有字段
    - 在子类的构造器中无法访问父类的私有字段，所以不能在子类的构造器中直接初始化父类的私有字段
  - 如果没有在子类构造器中显示调用父类的某个构造器，则编译器会自动调用父类无参构造器

```
public Manager(String n,double s,int year,
    int month,int day,double b) {
    //父类的字段调用父类构造器初始化，该语句必须作为第一条语句
    super(n,s,year,month,day);
    //子类的字段在子类的构造器中初始化
    bound = b;
}
```

参见例5-1:ManagerTest.java

5.1.2 继承层次

- 单继承
  - Java只支持单继承，不允许多重继承
    - 即一个类只能有一个超类
  - 继承未必只是一层
    - 还可以从Manager类派生出Executive类
  - 一个祖先类可以同时有多个子孙继承链



5.1.3 多态和动态绑定

- 置换法则
  - 无论何时，只要程序需要一个超类对象，那么就可以用一个子类对象来替代它
    - 例如：每个经理都是员工，可以把经理对象（子类对象）赋给员工变量（超类变量）

```
//可以把子类的对象赋给父类变量
Employee e;
e = new Employee(...);//变量e指向Employee类型对象
Manager boss;
boss = new Manager(...);
e = boss;//变量e指向Manager类型对象
boss.setBonus(5000);//正确
//下面的调用错误，e为Employee类型，没有setBonus方法
e.setBonus(5000);
```

5.1.3 多态和动态绑定

- 多态性
  - 一个对象变量可以指向多种实际类型对象的现象被称为“多态”（polymorphism）
- 动态绑定
  - 在运行时自动选择正确的方法进行调用的现象被称为“动态绑定”（dynamic binding）
    - Java中如果方法是用private、static或final修饰的，则采用静态绑定，否则都采用动态绑定
  - 多态性是靠“动态绑定”技术来实现的
  - 通过动态绑定无需对现有代码进行修改，就可以对程序进行扩展

参见: Polymorphism.java

5.1.4 final 关键字

- final 类和 final 方法
  - 类中用final关键字修饰的方法，在其子类中不能被覆盖
  - 用final关键字修饰的类不能被继承

```
class Employee {
    ...
    public final String getName() {
        return name;
    }//不能在Employee的子类中覆盖getName方法
    ...
}
final class Executive extends Manager {
    ...
}//不能从Executive派生子类
```

5.1.4 final 关键字

- 把方法或类设为 final 的原因
  - 效率
    - 调用final方法的行为属于静态绑定，减少了动态绑定带来的开销
    - 编译器可以对小的final方法调用使用内联代码替换
  - 安全
    - 动态绑定的灵活性可能带来一些不明确的情况
    - 一般可以把类或类中的方法设置final的来避免不明确情况的发生，即防止多态性的发生
      - 例如，Math类和System类都是final类

5.1.5 强制类型转换

- 类型转换
  - 像把浮点数转换成整数一样，有时也需要把某个类的对象引用转换为另一个类的对象引用
  - 对于类类型的转换只能在继承链上的类之间进行
    - 子类对象转换成父类对象可以隐式转换
    - 父类对象转换成子类对象需要强制类型转换

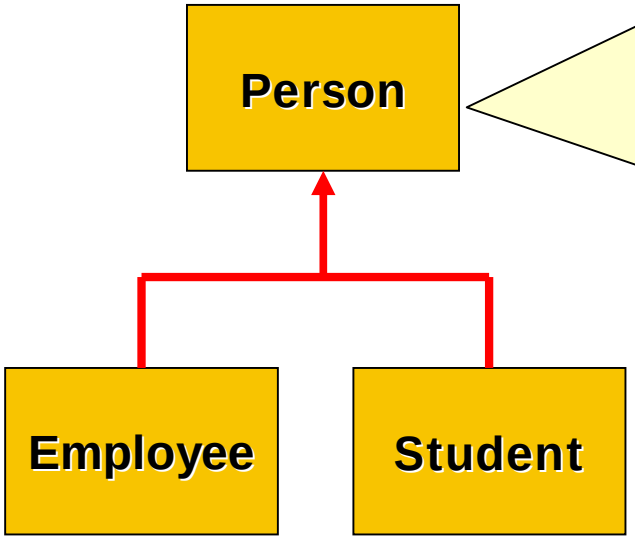
```
Manager boss = new Manager(...);
Employee e = new Employee(...);
e = boss;//正确，可以隐式转换
boss = e;//错误，需要强制类型转换
boss = (Manager)e;//正确

Date c = (Date)boss;//错误，c和boss的类型不具有继承关系
```

5.1.5 强制类型转换

- 什么时候需要强制类型转换？
  - 当要使用父类类型的变量调用子类独有的方法时需要使用强制类型转换
    - 如：希望通过Employee类型的变量e调用Manager类中的方法（如setBonus）时
    - 一般出现这种需求的时候很可能是类设计的不合理，应该把这样的方法设计在超类Employee中
  - 在使用一些通用集合类时经常需要进行强制类型转换
    - 如：从ArrayList中获取的对象永远是Object类型，实际使用时需要强制转换成需要的类型
  - 在强制类型转换之前应使用instanceof操作符进行判断

5.1.6 抽象类

- 抽象类的由来
  - 逆着继承层次关系由下而上，类逐渐变得更为通用，也更为抽象




```
public String getName()
{
   return name;
}
public String getDescription()
{
   return "";
}//无法为Person返回确定的描述信息

//把getDescription声明为抽象方法
public abstract String getDescription() ;
```

5.1.6 抽象类

- 抽象方法
  - 没有方法体的方法称为抽象方法，即抽象方法只需声明，不需实现
  - 用关键字abstract修饰抽象方法
- 抽象类
  - 一般称含有抽象方法的类称为抽象类
  - 抽象类是不能被实例化的类，用abstract修饰

5.1.6 抽象类

- 抽象类的注意事项
  - 抽象类不能被实例化，即不能用new关键字去产生一个抽象类的对象，但可以用抽象类声明变量去指向该抽象类的子类对象
  - 含有抽象方法的类一定是抽象类
  - 抽象类可以作为基类派生出子类
  - 抽象类的子类如果是一个非抽象类，那么该子类必须覆盖父类中所有的抽象方法，否则这个子类还是抽象类
  - 抽象类中仍然可以有非抽象方法

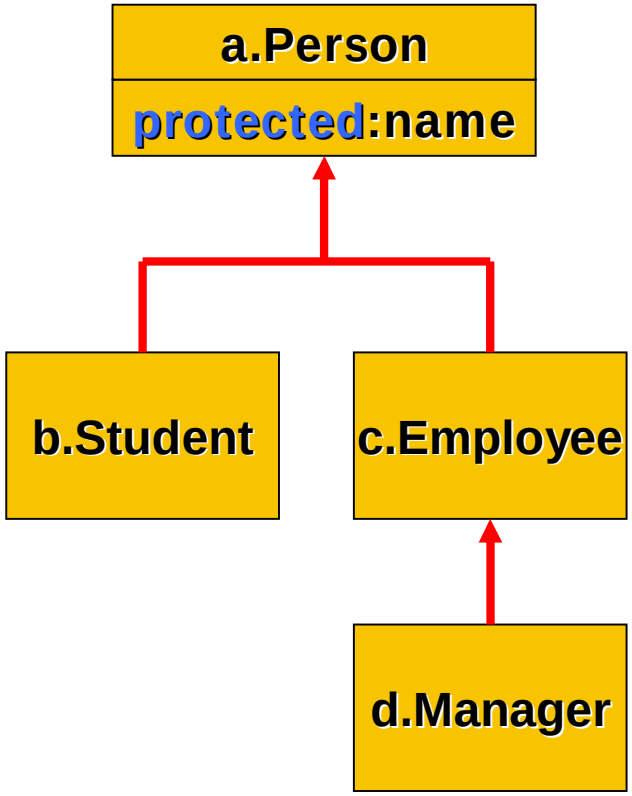
参见教材例5-2:PersonTest.java

5.1.7 受保护访问

- protected 关键字
  - 有时需要在子类中访问父类的成员（方法或字段），而不允许其它类访问该成员，这时需要把成员定义为protected（受保护的）
    - 例如，如果超类Employee把hireDay字段声明为受保护的，则可以在Manager类中访问该字段
    - 父类的protected成员只能在子类中直接访问，而不能在子类中访问其它父类对象的成员
  - 受保护成员除了可以被子类访问，还可以被同一包中的其他类访问

5.1.7 受保护访问

- 实例




```
//Manager.java
package d;
import a.Person;
import b.Student;
import c.Employee;
public class Manager extends Employee {
    public void displayName() {
        System.out.println(name);//正确
        //下面三条语句都有错误
        System.out.println(new Person().name);
        System.out.println(new Student().name);
        System.out.println(new Employee().name);
        //下面的语句正确
        System.out.println(new Manager().name);
    }
}
```

5.1.7 受保护访问

- 访问修饰符总结
  - 类的访问修饰符
    - public：任何包中的类都可以访问该类
    - 默认值：同一个包中的类可以访问该类
  - 成员的访问修饰符
    - public：对一切类可见
    - protected：对所有子类和同一包中的类可见
    - 默认值：对同一包中的类可见
    - private：只对本身类可见

注：访问成员的前提是首先能访问成员所属的类

5.2 Object：所有类的根类

- Object 类
  - Object类是Java中所有类的最终祖先，每一个类都直接或间接由Object类扩展而来

    class Employee {...} 等价于

    class Employee extends Object {...}
  - 可以用Object类型变量指向任意类型的对象

    Object obj = new Employee(...);
  - Object类型的变量只能执行各种通用的操作，如果要对它们进行专门的操作，需要强制类型转换

    Employee e = (Employee)obj;
  - Object类中封装了一些通用的方法，一般这些方法需要在其子类中覆盖

5.2 Object：所有类的根类

- equals方法
  - Object类中的equals方法用于测试某个对象是否同另一个对象相等
    - Object类中该方法的实现是判断两个对象是否指向同一个对象，与“==”的作用相同，一般需要覆盖该方法

```
class Employee {
  public boolean equals(Object otherObject) {
    if(this == otherObject) return true;
    if(otherObject == null) return false;
    if(getClass() != otherObject.getClass()) return false;
    Employee other = (Employee)otherObject;
    return name.equals(other.name)
       && salary == other.salary
       && hireDay.equals(other.hireDay);
   }
}
```

5.2 Object：所有类的根类

- equals 方法
  - Java语言规范要求equals方法应具有如下特点
    - 自反性：对于任意非空引用x，x.equals(x)应返回true
    - 对称性：对于任意非空应用x、y，当且仅当y.equals(x)返回true时，x.equals(y)返回true
    - 传递性：对于任意应用x、y和z，如果x.equals(y)返回true且y.equals(z)也返回true，那么x.equals(z)应返回true
    - 一致性：如果x和y引用的对象没有改变，那么对x.equals(y)的重复调用应返回同一结果
    - 对任意非空引用x，x.equals(null)应返回false

5.2 Object：所有类的根类

- equals 方法
  - 编写equals方法的建议
    - 显式参数命名为otherObject
    - 测试this同otherObject是否是同一个对象，如果是返回true
    - 测试otherObject是否为null，如果为null返回false
    - 测试this和otherObject是否属于同一个类，如果不属于同一个类返回false
    - 把otherObject的类型转换为你的类类型
    - 最后，比较所有的字段，如果所有字段匹配返回true，否则返回false
      - 使用“==”比较基本类型字段
      - 使用equals方法比较对象字段

5.2 Object：所有类的根类

- equals 方法
  - 定义子类的equals方法时，首先调用超类的equals方法
    - 如果调用超类的equals方法返回值为false，则子类的equals方法返回false
    - 如果调用超类的equals方法返回值为true，则继续比较子类的实例字段

```
class Manager extends Employee {
    ...
    public boolean equals(Object otherObject) {
       if(!super.equals(otherObject)) return false;
       Manager other = (Manager)otherObject;
       return bonus == other.bouns;
    }
}
```

5.2 Object：所有类的根类

- 常见的实现 equals 方法的错误

```
public class Employee {
    ...
    public boolean equals(Employee other) {
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
}
//该方法没有覆盖Object类中的equals方法，而是声明了一个新的方法
//在JDK5.0中可以使用@Override对覆盖超类的方法进行标记

@Override public boolean equals(Employee other)
```

5.2 Object：所有类的根类

- toString 方法
  - Object类的toString方法返回一个代表该对象值的字符串
  - 几乎每个类都会覆盖该方法，用来返回对该对象当前状态的表示，例如Point类的toString方法返回如下字符串
    - java.awt.Point[x=10,y=20]
    - Employee类的toString方法实现如下：

```
public String toString() {
   return getClass().getName()
      + "[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay + "]";
}
```

5.2 Object：所有类的根类

- toString 方法
  - 定义子类的toString方法时，在超类toString方法的基础上增加子类字段
    - Menager类的toString方法实现如下：

```
class Manager extends Employee {
    ...
    public String toString() {
        return super.toString()
            + "[bonus=" + bonus
            + "]";
    }
}
```

Manager对象的toString方法返回结果为：

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

5.2 Object：所有类的根类

- toString 方法
  - 无论何时对象同字符串相连接，那么就可以使用“+”操作符，这时Java编译器会自动调用对象的toString方法获得对象的字符串表示。例如：

```
Point p = new Point(10,20);
String message = "The current position is" + p;
//自动调用p.toString()
//可以使用"" + x来代替x.toString()
```

  - 应该在自己编写的每一个类中都添加toString方法，以方便在使用该类时进行调试

参见教材例5-3:EqualsTest.java

5.2 Object：所有类的根类

- 通用编程
  - 任何类（基本类型除外）的对象都可以用Object类型的变量来保存

```
Object obj = "Hello"; //正确
obj = 5; //错误
obj = false; //错误
```

  - 所有的数组都是从Object类派生出的类类型

```
Employee[] staff = new Employee[10];
Object arr = staff; //正确
arr = new int[10]; //正确
```

5.2 Object：所有类的根类

- 通用编程
  - 类类型的数组可以转换为超类类型的数组
    - 例如，一个Employee[]数组可以传递给一个参数为Object[]数组的方法，这种转换对于通用编程是非常有用的
    - 例如下面的例子，假设要在数组中找到某个元素的下标的方法如下，该方法使用Object[]作为参数，可以对任何类类型数组进行查找，以达到代码通用的目的

```
static int find(Object[] a,Object key) {
    int i;
    for(i=0;i<a.length;i++)
        if(a[i].equals(key) return i;
    return -1;
}
```

5.2 Object：所有类的根类

- 通用编程
  - 只能把对象数组转换为Object[]数组，而不能把int[]等基本类型数组转换成Object[]数组
  - 当把一个对象数组转换成Object[]数组，那么通用数组在运行时仍会记得对象的原始类型，不能把一个其他类型的对象存储进数组

```
Employee[] staff = new Employee[10];
... //填充Employee对象
Object[] arr = staff;
arr[0] = new Date(); //产生运行时错误，抛出异常
for(i=0;i<n;i++)
   staff[i].raiseSalary(3); //产生运行时错误，抛出异常
```

5.3 范型数组列表

- 数组列表
  - 可以存放任意类型对象的“自动伸缩”的数组
  - 构造一个保存Employee对象的数组列表

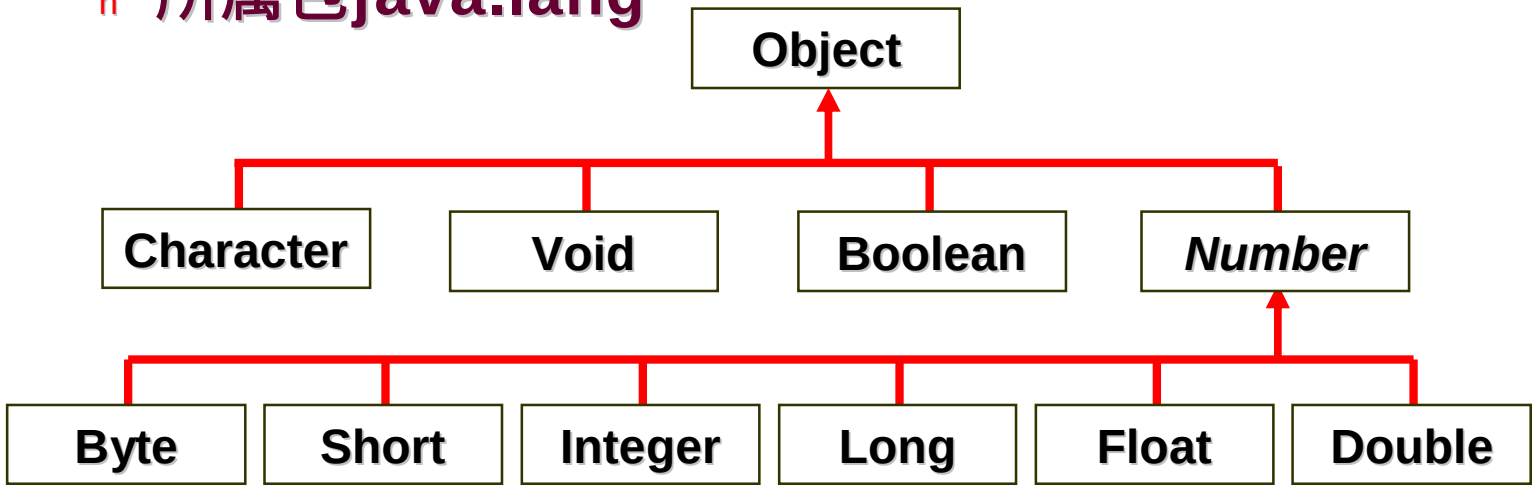
```
ArrayList<Employee> staff =
    new ArrayList<Employee>();
```

- 数组列表的常用方法
  - add、set、get、remove
  - size、trimToSize
- JDK 5.0 之前的数组列表
  - 是一个存放Object类型元素的“自动伸缩”的数组

参见教材例5-4:ArrayListTest.java

5.4 对象包装器与自动打包

- 有时需要把基本类型转换成引用类型
  - 例如，希望把基本类型存入数组列表
    - ArrayList<int> list = new ArrayList<int>(); //错误
    - ArrayList<Integer> list = new ArrayList<Integer>();
- 每种基本类型都对应一个包装器类
  - 所属包java.lang



5.4 对象包装器与自动打包

- 使用构造器构造对象包装器
  - 包装器可以使用其所对应的基本类型值或从数值字符串来构造
    - Double d = new Double(3.14);
    - Double d = new Double("3.14");
- 使用valueOf工厂方法构造对象包装器
  - 数值包装类中的静态方法valueOf用来创建一个数值包装类对象，并将初始化值表示为指定字符串表示的值
    - Double doubleObject = Double.valueOf("12.4");
    - Integer integerObject = Integer.valueOf("12");

5.4 对象包装器与自动打包

- Number类及其子类
  - 数值型包装类的通用方法都封装在Number抽象类中
  - Number类定义了抽象方法用来将对象表示的数值转换为基本数值类型
    - public byte byteValue()
    - public short shortValue()
    - public int intValue()
    - public long longValue()
    - public float floatValue()
    - public double doubleValue()

5.4 对象包装器与自动打包

- 数值包装器中的常量
  - MAX_VALUE
    - 表示对应基本类型的最大值
  - MIN_VALUE
    - 对应整型表示最小值，对于浮点型表示最小正浮点数
- 字符包装器中的方法
  - public static boolean isDigit(char ch);
  - public static boolean isLowerCase(char ch);
  - public static boolean isSpaceChar(char ch);
  - public static char toLowerCase(char ch);

5.4 对象包装器与自动打包

- 将数值型字符串转换为数值
  - 每个数值包装类中有一个静态方法，用来将数值型字符串转换为该数值包装类代表的数值
    - Double类的方法为parseDouble(String s)
    - Float类的方法为parseFloat(String s)
    - Long类的方法为parseLong(String s)
    - Integer类的方法为parseInt(String s)
    - Short类的方法为parseShort(String s)
    - Byte类的方法为parseByte(String s)

5.4 对象包装器与自动打包

- 自动打包
  - Java 编译器可以自动的将基本类型变量转换成其对应的包装类对象，例如：

    list.add(3); 将自动被编译器转换成如下语句：

    list.add(new Integer(3));
- 自动拆包
  - 当将一个基本类型对应的包装类类型变量赋值给基本类型时，编译器会自动进行拆包，例如：

    int n = list.get(i); 将被转换成

    int n = list.get(i).invValue();

5.5 参数数量可变的方法

- 方法的参数数目可以改变
  - 在JDK 5.0中，可以用可变的参数数目调用的方法，称为“可变参”方法
  - 例如System.out.printf方法的参数数目是可变的
- “可变参”方法的定义

```
public class PrintStream
{
   public PrintStream printf
      (String fmt,Object...args)
   {
      return format(fmt,args);
   }
}
```

5.6 枚举类

- 枚举类的声明

  enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE }

- 枚举类的常用方法
  - toString
    - 返回枚举常量名，Size.SMALL.toString();
  - valueOf
    - 构造枚举对象
    - Size s = (Size)Enum.valueOf(Size.class,"SMALL");
  - value
    - 返回全部枚举值的数组
    - Size[] values = Size.value();

参见教材例5-9:EnumTest.java

5.7 继承设计技巧

- 将公共操作和字段放到超类中
- 尽量不要使用受保护字段
  - 但可以使用protected方法
- 使用继承来模型化“is-a”关系
- 除非所有继承的方法都有意义，否则不要使用继承
- 在覆盖方法的时候，不要改变预期的行为
- 使用多态，而非类型信息
- 不要过多地使用反射

Any Question?