

【软件构造】第三章第四节 面向对象编程OOP

第三章第四节 面向对象编程OOP

本节讲学习ADT的具体实现技术：OOP

Outline

- OOP的基本概念
 - 对象
 - 类
 - 接口
 - 抽象类
- OOP的不同特征
 - 封装
 - 继承与重写(override)
 - 多态与重载(overload)
 - 重写与重载的区别
 - 泛型
- 设计好的类

Notes

OOP的基本概念

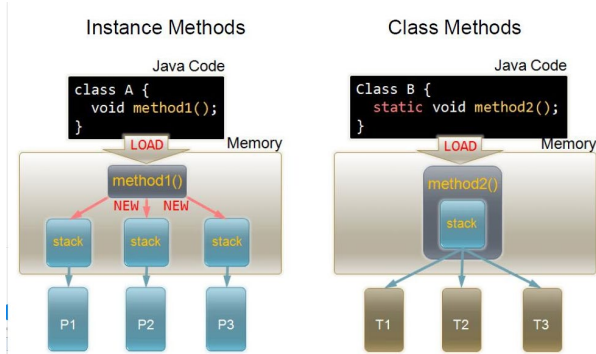
【对象】

- 对象是类的一个实例，有状态和行为。
- 例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- 概念：一个对象是一堆状态和行为的集合。
 - 状态是包含在对象中的数据，在Java中，它们是对象的fields。
 - 行为是对象支持的操作，在Java中，它们称为methods。

【类】

- 类是一个模板，它描述一类对象的行为和状态。
- 每个对象都有一个类
- 类定义了属性类型（type）和行为实现（implementation）
- 简单地说，类的方法是它的应用程序编程接口（API）。
- 类成员变量(class variable)又叫静态变量；类方法(class method)又叫静态方法：
- 实例变量(instance variable)和实例方法(instance method)是不用static形容的实例和方法；
- 二者有以下的区别：
 - 类方法是属于整个类，而不属于某个对象。

- 类方法只能访问类成员变量（方法），不能访问实例变量（方法），而实例方法可以访问类成员变量（方法）和实例变量（方法）。
- 类方法的调用可以通过类名.类方法和对象.类方法，而实例方法只能通过对象.实例方法访问。
- 类方法不能被覆盖，实例方法可以被覆盖。
- 当类的字节码文件被加载到内存时，类的实例方法不会被分配入口地址 当该类创建对象后，类中的实例方法才分配入口地址，从而实例方法可以被类创建的任何对象调用执行。
- 类方法在该类被加载到内存时，就分配了相应的入口地址。从而类方法不仅可以被类创建的任何对象调用执行，也可以直接通过类名调用。类方法的入口地址直到程序退出时才被取消。
- 注意：
 - 当我们创建第一个对象时，类中的实例方法就分配了入口地址，当再创建对象时，不再分配入口地址。
 - 也就是说，方法的入口地址被所有的对象共享，当所有的对象都不存在时，方法的入口地址才被取消。
- 总结：
 - 类变量和类方法与类相关联，并且每个类都会出现一次。使用它们不需要创建对象。
 - 实例方法和变量会在每个类的实例中出现一次。
- 举例：



【接口】

- 概念：接口在JAVA编程语言中是一个抽象类型，用于设计和表达ADT的语言机制，其是抽象方法的集合，接口通常以interface来声明。
 - 一个类通过继承接口的方式，从而来继承接口的抽象方法。
 - 接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。
 - 一个接口可以扩展其他接口，一个类可以实现多个接口；一个接口也可以有多重实现
 - 除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。
 - 接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。
- 另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

一个接口的实例：



```
/** MyString represents an immutable sequence of characters. */
public interface MyString {

    // We'll skip this creator operation for now
    // /** @param b a boolean value
    // * @return string representation of b, either "true" or "false" */
```

```
// public static MyString valueOf(boolean b) { ... }


/** @return number of characters in this string */
public int length();

/** @param i character position (requires 0 <= i < string length)
 *  @return character at position i */
public char charAt(int i);

/** Get the substring between start (inclusive) and end (exclusive).
 *  @param start starting index
 *  @param end ending index. Requires 0 <= start <= end <= string length.
 *  @return string consisting of charAt(start)...charAt(end-1) */
public MyString substring(int start, int end);
}
```



一种实现:

```

1 public class FastMyString implements MyString {
2
3     private char[] a;
4     private int start;
5     private int end;
6
7     /** Create a string representation of b, either "true" or "false".
8      *  @param b a boolean value */
9     public FastMyString(boolean b) {
10         a = b ? new char[] { 't', 'r', 'u', 'e' }
11             : new char[] { 'f', 'a', 'l', 's', 'e' };
12         start = 0;
13         end = a.length;
14     }
15
16     /** private constructor, used internally by producer operations.
17     private FastMyString(char[] a, int start, int end) {
18         this.a = a;
19         this.start = start;
20         this.end = end;
21     }
22
23     @Override public int length() { return end - start; }
24
25     @Override public char charAt(int i) { return a[start + i]; }
26
27     @Override public MyString substring(int start, int end) {
28         return new FastMyString(this.a, this.start + start, this.end + end);
29     }
}
```

```
30 }
```



客户端如何使用此ADT？这是一个例子：

```
MyString s = new FastMyString(true);
System.out.println("The first character is: " + s.charAt(0));
```

但其中有问题，这么实现接口打破了抽象边界，接口定义中没有包含**constructor**，也无法保证所有实现类中都包含了同样名字的**constructor**。故而，客户端需要知道该接口的某个具体实现类的名字。因为Java中的接口不能包含构造函数，所以它们必须直接调用其中一个具体类的构造函数。该构造函数的规范不会出现在接口的任何地方，所以没有任何静态的保证，即不同的实现甚至会提供相同的构造函数。

在Java 8中，我们可以用**valueOf**的静态工厂方法 代替构造器。



```
public interface MyString {
    /** @param b a boolean value
     * @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) {
        return new FastMyString(true);
    }
    // ...
}
```



此时，客户端使用ADT就不会破坏抽象边界：

```
MyString s = MyString.valueOf(true);
System.out.println("The first character is: " + s.charAt(0));
```

总结：接口的好处

- Safe from bugs

ADT是由其操作定义的，接口就是这样做的。

当客户端使用接口类型时，静态检查确保他们只使用由接口定义的方法。

如果实现类公开其他方法，或者更糟糕的是，具有可见的表示，客户端不会意外地看到或依赖它们。

当我们有一个数据类型的多个实现时，接口提供方法签名的静态检查。

- Easy to understand

客户和维护人员确切知道在哪里查找ADT的规约。

由于接口不包含实例字段或实例方法的实现，因此更容易将实现的细节保留在规范之外。

- Ready for change

通过添加实现接口的类，我们可以轻松地添加新类型的实现。

如果我们避免使用静态工厂方法的构造函数，客户端将只能看到该接口。

这意味着我们可以切换客户端正在使用的实现类，而无需更改其代码。

【抽象类】

- 抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

- 由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用。
- 父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。
- 在Java中抽象类表示的是一种继承关系，一个类只能继承一个抽象类，而一个类却可以实现多个接口。
- 如果一个类包含抽象方法，那么该类必须是抽象类。
- 任何子类必须重写父类的抽象方法，或者声明自身为抽象类。
- 构造方法，类方法（用static修饰的方法）不能声明为抽象方法。

OOP的不同特征

【封装】

- 封装（英语：**Encapsulation**）是指一种将抽象性函式接口的实现细节部份包装、隐藏起来的方法。
- 设计良好的代码隐藏了所有的实现细节
 - 干净地将API与实施分开
 - 模块只能通过API进行通信
 - 对彼此的内在运作不了解
- 信息封装的好处
 - 将构成系统的类分开，减少耦合
 - 加快系统开发速度
 - 减轻了维护的负担
 - 启用有效的性能调整
 - 增加软件复用
- 信息隐藏接口
 - 使用接口类型声明变量
 - 客户端仅使用接口中定义的方法
 - 客户端代码无法直接访问属性
- 实现封装的方法
 1. 修改属性的可见性来限制对属性的访问（一般限制为private），例如

```
public class Person {  
    private String name;  
    private int age;  
}
```

2. 对每个值属性提供对外的公共方法访问，也就是创建一对赋取值方法，用于对私有属性的访问，例如：



```
1 public class Person{  
2     private String name;  
3     private int age;  
4  
5     public int getAge() {  
6         return age;  
7     }  
8  
9     public String getName() {  
10        return name;  
11    }  
}
```

```

12
13     public void setAge(int age) {
14         this.age = age;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20 }

```



采用 **this** 关键字是为了解决实例变量（`private String name`）和局部变量（`setName(String name)`中的`name`变量）之间发生的同名的冲突。

【继承与重写】

- 继承概念：继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。
- 重写概念：重写是子类对父类的允许访问的方法的实现过程进行重新编写，返回值和形参都不能改变。即外壳不变，核心重写！
- 重写的好处在于子类可以根据需要，定义特定于自己的行为。也就是说子类能够根据需要实现父类的方法。
- 实际执行时调用那种方法，在运行时决定
- 重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。
- 子类只能添加新方法，无法重写超类中的方法。
- 当子类包含一个覆盖超类方法的方法时，它也可以使用关键字`super`调用超类方法。例子如下：



```

1 class Animal{
2     public void move() {
3         System.out.println("动物可以移动");
4     }
5 }
6
7 class Dog extends Animal{
8     public void move() {
9         super.move(); // 应用super类的方法
10        System.out.println("狗可以跑和走");
11    }
12 }
13
14 public class TestDog{
15     public static void main(String args[]){
16
17         Animal b = new Dog(); // Dog 对象
18         b.move(); //执行 Dog类的方法
19
20     }
21 }

```



- 方法重写的规则

- 参数列表必须完全与被重写方法的相同；
- 返回类型必须完全与被重写方法的返回类型相同；
- 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为`public`，那么在子类中重写该方法就不能声明为`protected`。
- 父类的成员方法只能被它的子类重写。
- 声明为`final`的方法不能被重写。
- 声明为`static`的方法不能被重写，但是能够被再次声明。
- 子类 and 父类在同一个包中，那么子类可以重写父类所有方法，除了声明为`private`和`final`的方法。
- 子类 and 父类不在同一个包中，那么子类只能重写父类的声明为`public`和`protected`的非`final`方法。
- 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
- 构造方法不能被重写。
- 如果不能继承一个方法，则不能重写这个方法。

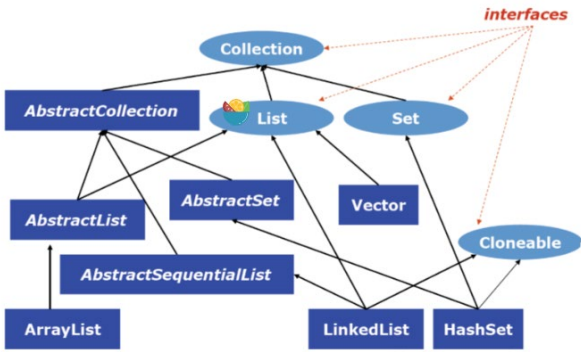
【多态与重载】

- 多态是同一行为具有多种不同表现形式或形态的能力
- 三种类型的多态
 - Ad hoc polymorphism (特殊多态): 功能重载，一个函数可以有多个同名的实现。
 - Parametric polymorphism (参数多态): 泛型或泛型编程，一个类型名字可以代表多个类型
 - Subtyping (also called subtype polymorphism or inclusion polymorphism 子类型多态、包含多态): 当一个名称表示许多不同的类与一些常见的超类相关的实例。
- 重载(overloading) 是在一个类里面，方法名字相同，而参数不同，返回类型可以相同也可以不同。
- 每个重载的方法（或构造函数）都必须有一个独一无二的参数类型列表。
- 价值：方便client调用，client可用不同的参数列表，调用同样的函数。
- 重载是静态多态，根据参数列表进行最佳匹配。在编译阶段时决定要具体执行哪个方法 (static type checking)，与之相反，重构方法则是在run-time进行dynamic checking！
- 重载规则
 - 被重载的方法必须改变参数列表(参数个数或类型不一样)；
 - 被重载的方法可以改变返回类型；
 - 被重载的方法可以改变访问修饰符；
 - 被重载的方法可以声明新的或更广的检查异常；
 - 方法能够在同一个类中或者在一个子类中被重载。
 - 无法以返回值类型作为重载函数的区分标准。

⊕ 特殊多态 (Ad hoc)

⊕ 参数多态性和泛型编程

子类型多态



子类型的规约不能弱化超类型的规约。
子类型多态：不同类型的对象可以统一的处理而无需区分，从而隔离了“变化”。

【重写与重载的区别】

区别点	重载方法	
参数列表	必须修改	
返回类型	可以修改	
异常	可以修改	可以减少或删除，
访问	可以修改	一定不能做更
调用情况	引用类型决定选择哪个重载版本（基于声明的参数类型）。在编译时发生。对象类型（换句话说，堆上实际	

方法的重写(Overriding)和重载(Overloading)是java多态性的不同表现，重写是父类与子类之间多态性的一种表现，重载可以理解成多态的具体表现形式。

- 方法重载是一个类中定义了多个方法名相同,而他们的参数的数量不同或数量相同而类型和次序不同,则称为方法的重载(Overloading)。
- 方法重写是在子类存在方法与父类的方法的名字相同,而且参数的个数与类型一样,返回值也一样的方法,就称为重写(Overriding)。
- 方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现。

Overriding 重写

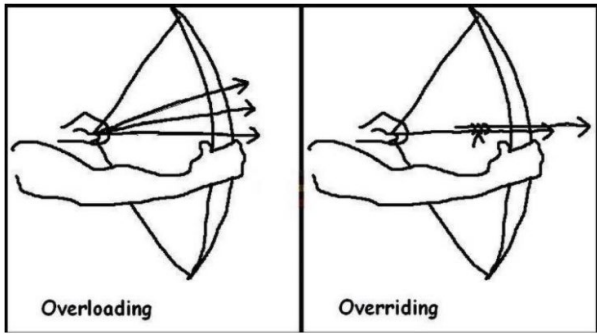
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }
    public void bark(){
        System.out.println("bowl");
    }
}
```

方法名与参数都一样

Overloading 重载

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++){
            System.out.println("woof ");
        }
    }
}
```

方法名相同，参数不同



【泛型】（参数多态）

- 泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。
- 可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。
- 下面是定义泛型方法的规则：
 - 所有泛型方法声明都有一个类型参数声明部分（由尖括号分隔），该类型参数声明部分在方法返回类型之前（在下面例子中的<E>）。
 - 每一个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。
 - 类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。
 - 泛型方法体的声明和其他方法一样。注意类型参数只能代表引用型类型，不能是原始类型（像int,double,char的等）。

⊕ 泛型接口，非泛型的实现类

⊕ 泛型接口，泛型的实现类

- 一些细节：
 - 可以有多个类型参数：例如Map<E, F>, Map<String, Integer>
 - 通配符，只在使用泛型的时候出现，不能在定义中出现，例：List<?> list = new ArrayList<String>();
 - 泛型类型信息被删除
 - Cannot use instanceof() to check generic type 运行时泛型消失了！
 - 无法创建通用数组

```
Pair<String>[] foo = new Pair<String>[42]; // won't compile
```

设计好的类

- 好的类具有的特点
 - 简单
 - 本质上是线程安全的
 - 可以自由分享
 - 不需要防御式拷贝
 - 优秀的building blocks
- 如何编写一个不可变的类
 - 使所有的fields有final修饰
 - 使所有的fields有private修饰
 - 不要提供任何mutators
 - 确保任何可变组件的安全性（避免表示泄露）
 - 确保没有方法可能被覆盖
 - 实现toString () , hashCode () , clone () , equals () 等。