
Effective Java

1. 创建和销毁对象

1.1 考虑用静态工厂方法代替构造器

- 静态工厂方法

区分设计模式的工厂方法。这里就是在原有的构造器上包装一层。例如：


```
public static StaticFactory newInstance() {  
    //TODO 做一些处理  
    return new StaticFactory();  
}
```

- 问题：静态工厂比构造器好在哪里？

1、有名称

2、不必在每次调用它们的时候都创建一个新对象

```
private volatile static StaticFactory staticFactory = null;  
private StaticFactory() {  
}  
public static StaticFactory newInstance() {  
    if (staticFactory == null) {  
        synchronized (StaticFactory.class) {  
            if (staticFactory == null) {  
                return new StaticFactory();  
            }  
        }  
    }  
    return staticFactory;  
}
```



3、可以返回原返回类型的任何子类型对象

向上转型。更精细化的处理构造函数。

4、使代码变得更加简洁

```
public static HashMap<K,V> newInstance() {  
    return new HashMap<K,V>();  
}
```

本来是 `new HashMap<K,V>`（或者里面嵌套更多层），现在变成 `newInstance()`

- 总结

考虑使用并不意味着非要使用。使用静态工厂的四条好处需要不同的实现（或集中实现）。根据使用场景来考虑。主要考虑点应该是第 2 条和第 3 条好处。

1.2 遇到多个构造器参数时要考虑用构建器

- 多个构造器参数

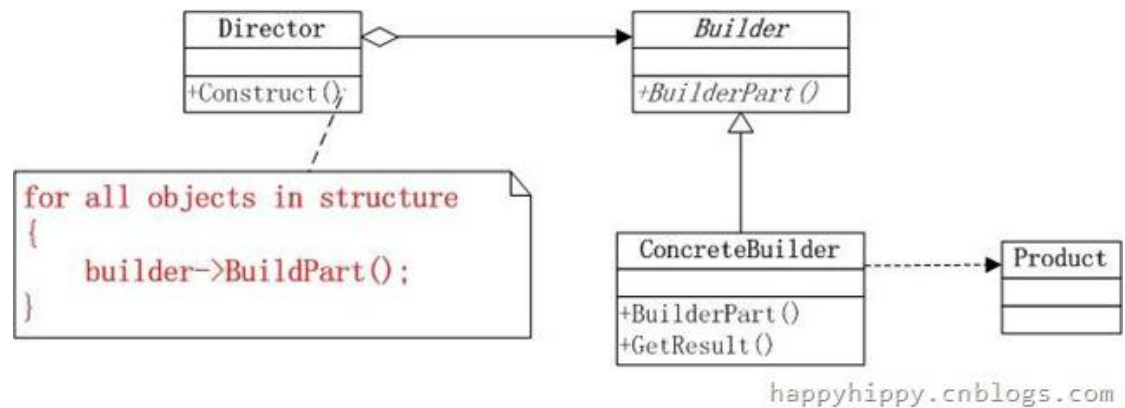
一般要四个及以上。

- 什么时候用构建器

如果类的构造器或者静态工厂中有多个参数，设计这种类时，Builder 模式就是种不错的选择，特别是大多数参数都是可选的时候。

- 构建器使用

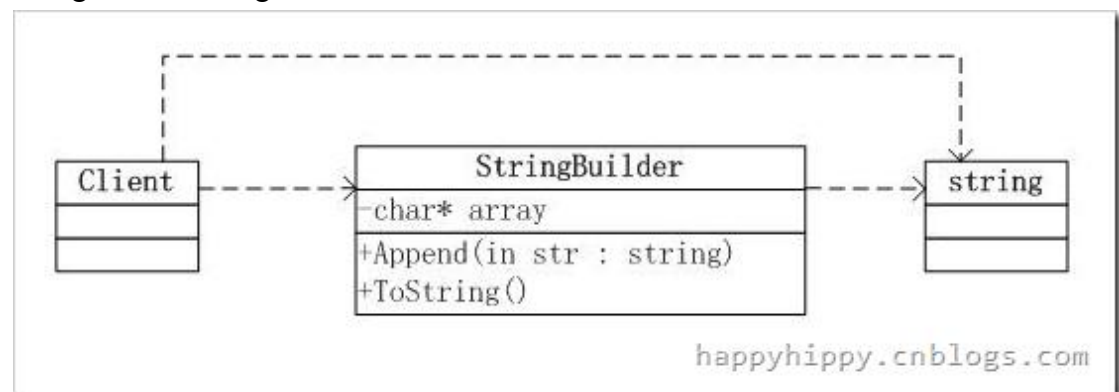
Builder 设计模式



值得注意的是红色字体部分的 for 循环，表示了循环调用函数。例如：

builder->BuildPart1->BuildPart2...

StringBuilder/StringBuffer



StringBuilder 类中的一个实现函数

```
public StringBuilder append(StringBuffer sb) {
    super.append(sb);
    return this;
}

sBuilder.append("first").append("second").toString();
```

例子：构建器构造

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // 必选参数
        private final int servingSize;
        private final int servings;
        // 可选参数
```



```

private int calories = 0;
private int fat = 0;
private int sodium = 0;
private int carbohydrate = 0;

public Builder(int servingSize, int servings) {
    this.servingSize = servingSize;
    this.servings = servings;
}

public Builder calories(int val) {
    calories = val;
    return this;
}

public Builder fat(int val) {
    fat = val;
    return this;
}

public Builder sodium(int val) {
    sodium = val;
    return this;
}

public Builder carbohydrate(int val) {
    carbohydrate = val;
    return this;
}

public NutritionFacts build() {
    return new NutritionFacts(this);
}

private NutritionFacts(Builder builder) {
    this.servingSize = builder.servingSize;
    this.servings = builder.servings;
    this.calories = builder.calories;
    this.fat = builder.fat;
    this.sodium = builder.sodium;
    this.carbohydrate = builder.carbohydrate;
}
}

```

例子：构建器使用

```

NutritionFacts cocaCola = new
NutritionFacts.Builder(240,8).calories(100).fat(10).build
();

```



● 总结

知识点：

- 1、静态内部类调用外部类的私有构造器

2、final 类型赋值的三种情况：1) 声明赋值 2) 构造器赋值 3) static{} 块中赋值

3、构建器通俗的理解就是把构造函数给分成了多个，不过拆分成的不是构造函数，而是与属性相对应的函数，返回值依然是本类对象。

构建器好处：对于多个变量，给参数赋值时更加明确了。用构造器时是根据编译器提示来对应赋值，构建器是显示约束条件，更加精准。

1.3 用私有构造器或者枚举类型强化 Singleton 属性

- 常见单例模式

1.1 好处 2 示例代码

- Joshua Bloch 推荐用法

好处：1.线程安全 2.不会因为序列化而产生新实例 3.防止反射攻击

枚举类型的单例声明

```
public enum Singleton{  
    singleton;  
}
```



这种怎么比较妥当的使用呢？

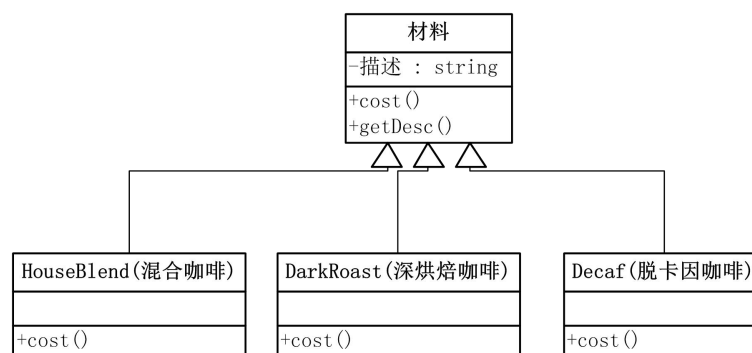
2. 类和接口

2.1 复合优先于继承

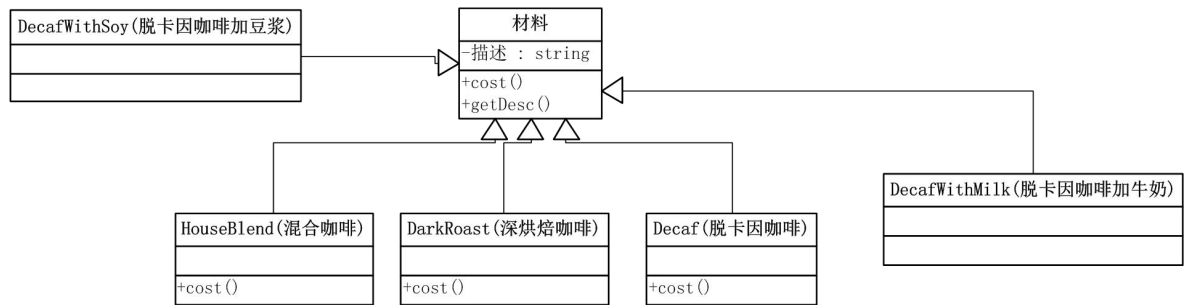
- 继承的缺陷

星巴克咖啡例子：（装饰者模式）

材料类为抽象类，子类通过继承材料类实现 cost() 方法来计算所需要费用。

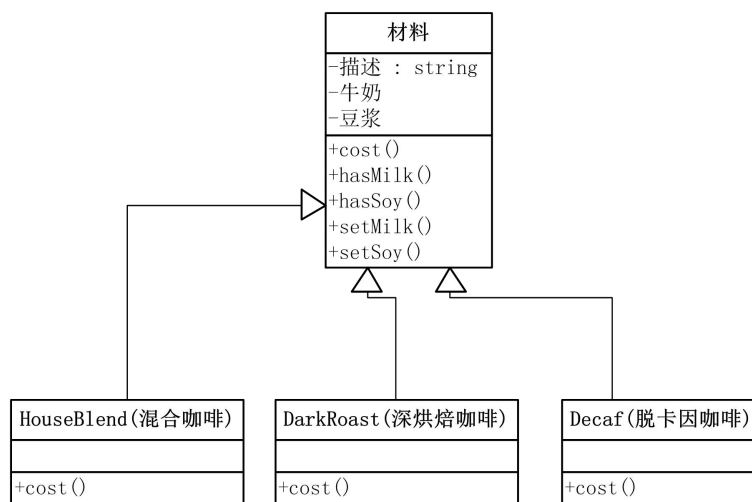


问题：购买咖啡时，也可以在其中加入各种调料，星巴克会根据所加入调料的不同收取不同的费用。这可能导致子类非常多...



继承改进版:

超类 `cost()` 计算所有的调料的价钱，而子类覆盖 `cost()`，把指定饮料类型的价钱加进来。



问题：以后可能会出现新饮料，（例如冰茶），某些调料可能不适合，但是继承了材料类之后就继承了这些调料（例如:奶泡，牛奶，,,）

- 使用组合

装饰者模式（I/O 类），通过调用时候“包裹”达到“动态拼凑”的效果。

```
BufferedReader br = new BufferedReader(new
InputStreamReader(new FileInputStream(new File("xxx"))));
```

- 总结：继承是编译时静态决定，组合是运行时动态扩展。

3. 泛型

3.1 请不要在新代码中使用原生态类型

- 问题：为什么泛型会比原生态类型要好？

```
List list = new ArrayList<String>(); 1
list.add(new Integer(10));
List<Object> listObject = new ArrayList<Integer>(); 2
```

Change 'new ArrayList<Integer>()' to 'new ArrayList<Object>()'

Change variable 'listObject' type to 'java.util.ArrayList<java.lang.Integer>'

Migrate 'listObject' type to 'java.util.ArrayList<java.lang.Integer>'

在 IDEA 编译器中，第一种是原生态类型的使用，第二种是泛型的使用。

如上图，①中使用原生态数据类型编译器在编译期间不能检查到类型错误（及时期望的是 String 传入的是 Integer,试验发现几乎可以传递任何类型而都不会报错）。

②中使用泛型声明一个泛型对象，此时，即使实例化的时候使用 Object 的子类 Integer,仍然不能通过检查。（在集合优先于数组 中也有提到）

- **结论：**出错之后应该尽快发现，最好是编译时能发现。从这个方面考虑，泛型确实要好于原生态类型，泛型严格控制输入类型与所期望的类型的一致性。而且在 add()时，泛型也能检查类型。此时可以添加泛型 E 的子类型。

```
List<Object>listObject = new ArrayList<Object>();
listObject.add(new Integer(1));
```

- 问题：泛型中的通配符在编译期间也不检查类型，那它和原生态类型有什么区别？

上面的代码也可以说明这个问题，泛型在编译期间在添加元素的时候也可以元素类型。例如下面的代码编译不通过。

```
List<?> listGeneric = new ArrayList<String>();
listGeneric.add(new Integer(1));
```

而原生态类型则没有这个检查。

4. 枚举和注解

4.1 用 enum 代替 int 常量

项目中的部分代码：

```
private final String RESPONSE_SUCCESS_CODE = "0000";
private final String UNION_RESP_CODE_SUCCESS = "00";
private final String UNION_ORIG_RESP_CODE_SUCCESS = "00";
```



改造代码：

```
public enum ResponseCode {
    RESPONSE_CODE_SUCCESS("0000"),
    UNION_RESP_CODE_SUCCESS("00"),
    UNION_ORIG_RESP_CODE_SUCCESS("00");
    private final String code;
    ResponseCode(String code){
        this.code = code;
    }
}
```



```

    }
    public String getCode() {
    return code;
    }
}

```

4.2 用实例域代替序数

- 什么是实例域，什么是序数

序数就是枚举类中的通过 `ordinal()` 方法得到的枚举对象的顺序值，从 0 开始。实例域就是在枚举类中声明的保存值域的成员变量。

- 序数的缺陷

不好维护，因为它不能很好的标识枚举对象的性质（相当于根据数组的下标关联数组中的值）。当枚举对象顺序变化时，容易出错。

项目中代码：

```

int orderSerial = OrderStatus.GrandFail.ordinal(); // 0:待确认 1: 已确认 2: 已发
放 3: 发放物品失败 4: 已完成
orderStatus = OrderStatus.GrantSuccess.ordinal();

```



改造代码：

```

public int getOrderStatus() {
return orderStatus;
}
public String getOrderStatusName() {
return orderStatusName;
}
orderStatus = OrderStatus.GrantSuccess.getOrderStatus();

```



- 总结：永远不要根据枚举的序数导出与它关联的值，而是要将它保存在一个实例域中。`ordinal()` 方法最好完全避免使用。

5. 异常

5.1 只针对异常的情况才使用异常

- 说明：异常应该只用于异常的情况下；它们永远不应该用于正常的控制流。通过捕获异常来故意中断程序的正常运行是不可取的。

```

try{
int[] fiveContent = new int[5];
int i= 0;
while(true){
i++;
fiveContent[i] = i;
}
} catch (ArrayIndexOutOfBoundsException e) {
}

```



6. 方法

6.1 检查参数的有效性

- 总结

在方法执行他的计算任务之前，应该先检查它的参数。在方法体的开头处检查参数。

6.2 谨慎设计方法签名

- 避免过长的参数列表（四个或者更少，具体根据公司要求而定）


缩短参数列表方法：

1、分解方法

2、创建辅助类（静态成员类）


例子：

```
/**
 * 统计余下扑克牌的信息
 * points: 扑克点数
 * color: 花色
 */
private void countRemainPlate(int points, int color, ...) { }
/**
 * 是否有同花色顺子
 * points: 扑克点数
 * color: 花色
 */
private void sameColorStraight(int points, int color, ...) { }
```



大部分函数都需要花色和扑克点数等基本信息，这时候可以考虑声明静态成员类。例如：

```
private void sameColorStraight(PokerProperty property) {}
private void countRemainPlate(PokerProperty property) {}
public static class PokerProperty {
    private int point;
    private int color;
}
```




3、采用 Builder 模式（参考 1.2）

6.3 返回零长度的数组或者集合，而不是 null

- 数组

返回 null 值方式：（比较失败的处理方式）

```
private List<Cheese> cheeseInStock = new ArrayList<Cheese>();
public Cheese[] getCheese() {
    if (cheeseInStock.size() == 0) {
        return null;
    }
}
```




```
...
}
```

返回零长度数组

```
private List<Cheese>cheeseInStock = new ArrayList<Cheese>();
private final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
public Cheese[] getCheese(){
    if(cheeseInStock.size() == 0){
        return cheeseInStock.toArray(EMPTY_CHEESE_ARRAY);
    }
    ...
}
```



● 集合

```
private List<Cheese>cheeseInStock = new ArrayList<Cheese>();
private final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
public List<Cheese> getCheese(){
    if(cheeseInStock.isEmpty()){
        return Collections.emptyList();
    }else{
        return new ArrayList<Cheese>(cheeseInStock);
    }
}
```



7. 通用程序设计

7.1 将局部变量的作用于最小化

● 局部变量

局部变量区别于成员变量。成员变量是在类加载完后初始化的。跟放置的位置没有关系，所以一般放在类开头。局部变量的作用最小化就是在第一次使用它的地方声明。什么时候使用什么时候声明，而不是放在函数开头的位置。

7.2 for-each 循环（增强 for）优先于传统的 for 循环

● 总结

- 1、优先考虑使用增强 for(for(A aa:Alist)表示对于 Alist 中的每个元素。默认排除 Alist 为空的情况)
- 2、当需要对 for 里面的局部指定元素进行过滤或者转换时，慎用 for

7.3 如果需要精确的答案，请避免使用 float 和 double

- 计算货币时避免使用 float 和 double 类型
- 使用精确的 BigDecimal

BigDecimal 用法：

add(BigDecimal)	BigDecimal 对象中的值相加，然后返回这个对象。
subtract(BigDecimal)	BigDecimal 对象中的值相减，然后返回这个对象。
multiply(BigDecimal)	BigDecimal 对象中的值相乘，然后返回这个对象。
divide(BigDecimal)	BigDecimal 对象中的值相除，然后返回这个对象。

static int	<u>ROUND_CEILING</u> 接近正无穷大的舍入模式。 (直接进位)
static int	<u>ROUND_DOWN</u> 接近零的舍入模式。 (直接舍弃)
static int	<u>ROUND_FLOOR</u> 接近负无穷大的舍入模式。
static int	<u>ROUND_HALF_DOWN</u> 向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则为上舍入的舍入模式。 (四舍五入，如果为 5，入)
static int	<u>ROUND_HALF_EVEN</u> 向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则向相邻的偶数舍入。
static int	<u>ROUND_HALF_UP</u> 向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则为向上舍入的舍入模式。 (四舍五入，如果为 5，舍)

代码：

```
BigDecimal originalMoney = new BigDecimal("1.04");
originalMoney = originalMoney.add(new BigDecimal("0.0001"));
System.out.println(originalMoney.setScale(2, BigDecimal.ROUND_CEILING));
结果：1.05
```

7.4 基本类型优先于装箱基本类型

- 基本类型和装箱基本类型的区别

1、基本类型只有值，装箱基本类型是引用类型，值相等的两个装箱基本类型对象引用地址是不同的。

2、基本类型只有值，装箱基本类型除了它对应的基本类型的功能值以外，还有个非功能值：**null**，在拆箱操作时可能会抛 **NullPointerException** 异常。

3、基本类型通常比装箱基本类型更加节省时间和空间。

- 使用时，基本类型优先于装箱基本类型，能用基本类型的就用基本类型
- 项目中问题

```
private Integer id;
@Column(value = "order_serial")
private String orderSerial;
@Column(value = "uid")
private Long uid;
@Column(value = "total_fee")
```



每次从数据库中查到数据时，一条数据中数据转换时，uid 和 id 都会创建装箱类型对象。

- 总结

当可以使用基本类型时，就使用基本类型。

7.5 如果其他类型更适合，则尽量避免使用字符串

- 总结

- 1、字符串不适合代替其它值类型（int,float,boolean...）
- 2、字符串不适合代替枚举类型
- 3、字符串不适合代替聚集类型（例如：拼凑字符串来表示一个实体，这时候通常采用静态成员类）

7.6 通过接口引用对象

例子：

静态代理模式的例子：

```
public class Proxy implements GiveGift {
private Pursuit pursuit;
public Proxy(SchoolGirl girl){
pursuit = new Pursuit(girl);
}
@Override
public void sendFlower() {
pursuit.sendFlower();
}
}
```

直接使用类来引用对象：

```
Proxy proxy = new Proxy(schoolGirl);
proxy.sendFlower();
```



使用接口来引用对象：

```
GiveGift p = new Proxy(schoolGirl);
p.sendFlower();
```



- 总结

通过接口（而不是类）来引用对象可以降低耦合度，实现更好的扩展。面向接口编程。

8. 其它规则

- 注解优先于命名模式（不知道神马命名模式，就用注解）
- 坚持使用 `Override` 注解（有时候 `Override` 注解报错，检查工程中使用的 jdk 版本）
- 大量字符拼接时选用 `StringBuilder` 而不是 “+”
- 对可恢复的情况使用受检异常，对编程错误使用运行时异常