

# Java程序设计



## 第12章 怎样写好程序

# 内容提要




- 1. 写好程序的一些经验
- 2. 重构
- 3. 设计模式
- 4. 反射

# 1. 怎样写好程序 —经验谈





# 写好程序的经验谈

A horizontal line composed of white dots, spanning the width of the slide, positioned below the title.

# 写好程序



- 写好单词
- 写好语句
- 写好函数
- 写好类



- 变量

- ▣ 大小写：变量小写，方法名小写、类名大写
- ▣ 长度：单个字母的变量只能在三五行内结束
- ▣ 含义：用特定含义
  - 少用temp, it ,do等没有意义的词
  - 界面组件用匈牙利命名法（如btnSayHello）

```
void keepdog(String name_, List<Animal>
    System.out.println("I love pets and
    Dog dog = new Dog(name_);
    dog.born(familymember);
}
void happylife(){
    System.out.println("I live a happy
    System.out.println("My wife loves m
    System.out.println("Every day doggy
    Dog.bark();
}
```



# 例：使用中间变量

```
if (DTC[i].substring(0, 1).equals("0")) {
    DTC[i] = "P0" + DTC[i].substring(1);
} else if (DTC[i].substring(0, 1).equals("1")) {
    DTC[i] = "P1" + DTC[i].substring(1);
} else if (DTC[i].substring(0, 1).equals("2")) {
    DTC[i] = "P2" + DTC[i].substring(1);
} else if (DTC[i].substring(0, 1).equals("3")) {
    DTC[i] = "P3" + DTC[i].substring(1);
} else if (DTC[i].substring(0, 1).equals("4")) {
    DTC[i] = "C0" + DTC[i].substring(1);
}
```

```
char first = str.charAt(0);
String remain = str.substring(1);
String result = "";
```

```
if (first=='0') {
    result = "P0" + remain;
} else if (first=='1') {
    result = "P1" + remain;
} else if (first=='2') {
    result = "P2" + remain;
} else if (first=='3') {
    result = "P3" + remain;
}
```



# 例：使用中间变量

```
if((long)Math.pow(res,3.0)==sum  
    || (long)Math.pow(res+1,3.0)==sum){
```





# 写好“单词”（续）

- 常量

- 不要从天上掉下来一个数

- 常量用 final 或 enum
    - 字符串常量 如 BorderLayout.CENTER
    - 从配置中获取
    - 使用 java.util.Properties 类的 load() 及 loadXml()

```
if (mode == 3)
{
    timelimit += (int)(50*Math
```

# 写好“语句”



- 简单语句
  - 写好赋值语句
- 分支语句
  - 使用括号
- 循环语句
  - 变量局部化



# 例：使用卫语句(Guard Clauses)

```
public void Vlidate()
{
    if ((this.vlirate != null) && (this.vlirate != "")) {

        if (this.Session["VNum"].ToString() == this.vlirate

        UsersInfo info = this.um.UserLogin(this.userName, this.vlirate);
        if (info != null)
        {
            if (((info.UserRole == "管理员") || (info.UserRole == "普通用户")) {
                this.Session["User"] = info;
                this.Session["CompanyID"] = this.companyID;
                base.Response.Redirect("index.aspx");
            }
            else
            {
                base.Response.Write("<script>alert('您的登录权')");
            }
        }
    }
}
```

```
public void Vlidate()
{
    if ((this.vlirate == null) || (this.vlirate == "")) re

    if (this.Session["VNum"].ToString() == this.vlirate.To

        UsersInfo info = this.um.UserLogin(this.userName, this.vlirate);
        if (info != null)
        {
            if (((info.UserRole == "管理员") || (info.UserRole == "普通用户")) {
                this.Session["User"] = info;
                this.Session["CompanyID"] = this.companyID;
                base.Response.Redirect("index.aspx");
            }
            else
            {
                base.Response.Write("<script>alert('您的登录权')");
            }
        }
    }
}
```



- 写简单函数

- 使用卫语句降低层次

- 还有try-catch的正确使用、Lambda表达式的使用

- 语句不要太多

- 将一段语句提出来，形成新的函数

- 层次不要太多

- 将内部语句提出来，形成新的函数

- 改变算法

- 如果太复杂了，说明思路还不够清晰



```
for(int i=0;i<RC;++i){
    for(int j=0;j<RC-2;++j){
        int cnt = i*RC+j;
        int cur=num[cnt];
        if(cur!=0&&cur==num[cnt+1]&&cur==num[cnt+2])
        {
            int k,c;
            int tmpgrade=0;
            for(k=0;;++k){
                if((j+k)>=RC||cur!=num[cnt+k])break;
                num[cnt+k] = 0;
                int z=1;
                c=1;
                while(cnt-z*RC>0&&cur==num[cnt-z*RC]){
                    ++z;++c;
                }
            }
        }
    }
}
```





```
void addBtn(int r, int c) {  
    for (int i = 0; i < R; ++i) {  
        for (int j = 0; j < C; ++j) {  
            btn[i][j].addMouseListener(new MouseAdapter()  
            {  
                public void mouseClicked(MouseEvent e) {  
                    if (fail == false) {  
                        int r = 0, c = 0;  
                        boolean flag = false;  
                        for (r = 0; r < R; ++r) {  
                            for (c = 0; c < C; ++c) {  
                                if ((JButton) e.getSource()  
                                    flag = true;  
                                    break;  
                            }  
                        }  
                        if (flag == true)  
                            break;  
                    }  
                }  
            });  
        }  
    }  
}
```



# 写好“对象”



- 对象的功能是独立的
  - 高内聚、低耦合
  - 不要太多的成员
- 处理好对象之间的关系
  - 继承与实现接口
  - 关联：使用构造方法或普通方法
  - 更复杂的：使用设计模式 ( design patterns)

# “定律”



- 写简单程序
- 代码永远不要写两遍
  - 推论：永远不要copy代码





```
public void keyPressed(KeyEvent e){
    switch(e.getKeyCode()){
    case KeyEvent.VK_LEFT:
        for(int i=0; i<4; ++i)
            for(int j=0; j<4; ++j){
                if(iscur[i][j]){
                    if(current[i][j][1]==0)
                        return;
                    if(current[i][j][0]<0) continue;
                    if((j==0 || !iscur[i][j-1])
                        && (grids[current[i][j][0]
                        return;
                }
            }
        actions.ClearCur();
        ChangePosition(0,-1);
        actions.PaintCur();
        return;
    case KeyEvent.VK_RIGHT:
        for(int i=0; i<4; ++i)
```



```
void addBtn(int r, int c) {  
    for (int i = 0; i < R; ++i) {  
        for (int j = 0; j < C; ++j) {  
            btn[i][j].addMouseListener(new MouseAdapter()  
            {  
                public void mouseClicked(MouseEvent e) {  
                    if (fail == false) {  
                        int r = 0, c = 0;  
                        boolean flag = false;  
                        for (r = 0; r < R; ++r) {  
                            for (c = 0; c < C; ++c) {  
                                if ((JButton) e.getSource() == btn[r][c]) {  
                                    flag = true;  
                                    break;  
                                }  
                            }  
                        }  
                        if (flag == true)  
                            break;  
                    }  
                }  
            });  
        }  
    }  
}
```





# 重复的代码：提炼成函数

```
private void radioButton2_CheckedChanged(  
{  
    num1 = ran.Next(20);  
    operation = ran.Next(2);  
    num2 = ran.Next(20);  
    label11.Text = num1.ToString();  
    label13.Text = num2.ToString();  
    label12.Text = "+";  
}
```

```
private void radioButton3_CheckedChanged(  
{  
    num1 = ran.Next(20);  
    num2 = ran.Next(20);  
    label11.Text = num1.ToString();  
    label13.Text = num2.ToString();  
    label12.Text = "*";  
}
```



- Java语言编码规范(Java Code Conventions)

□ <http://www.oracle.com/technetwork/java/codeconv-138413.html>

## 2. 重构





# 重构





- 重构 ( Refactoring )
  - 通过调整程序代码改善软件的质量、性能，使其程序的设计模式和架构更趋合理，提高软件的扩展性和维护性

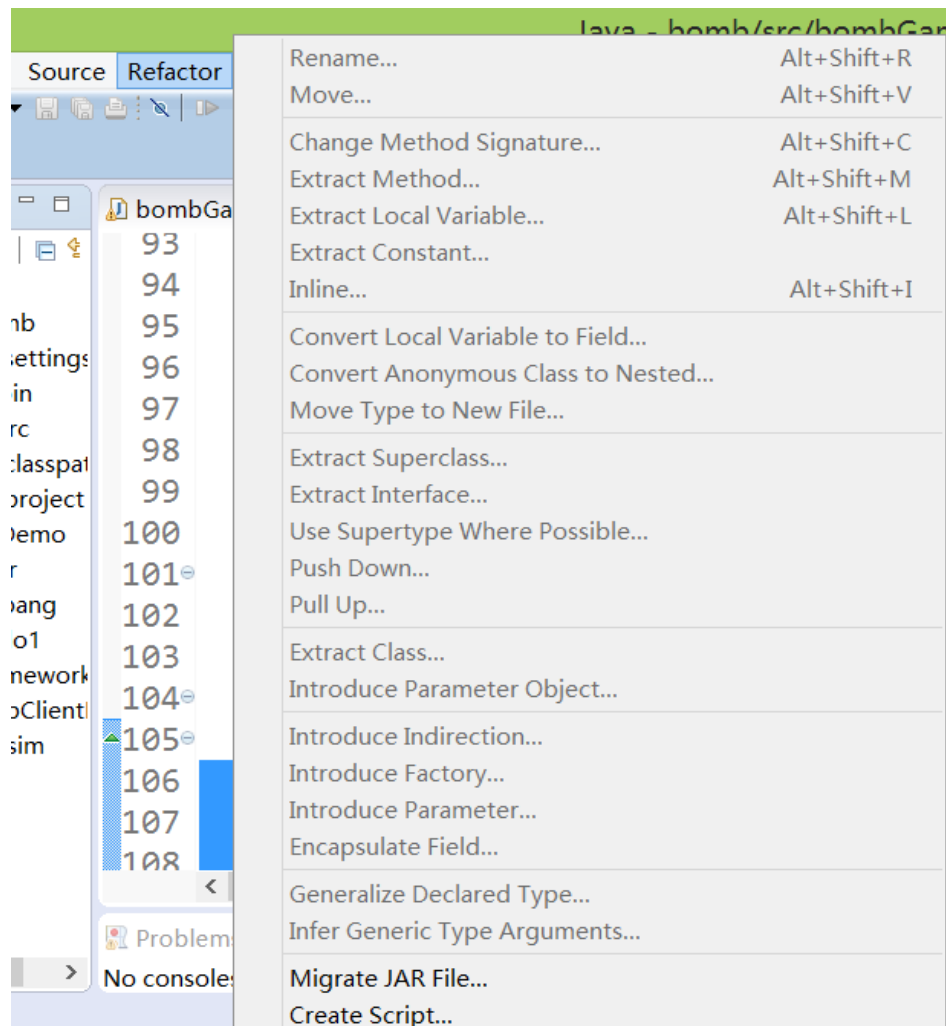




- Eclipse中的重构功能

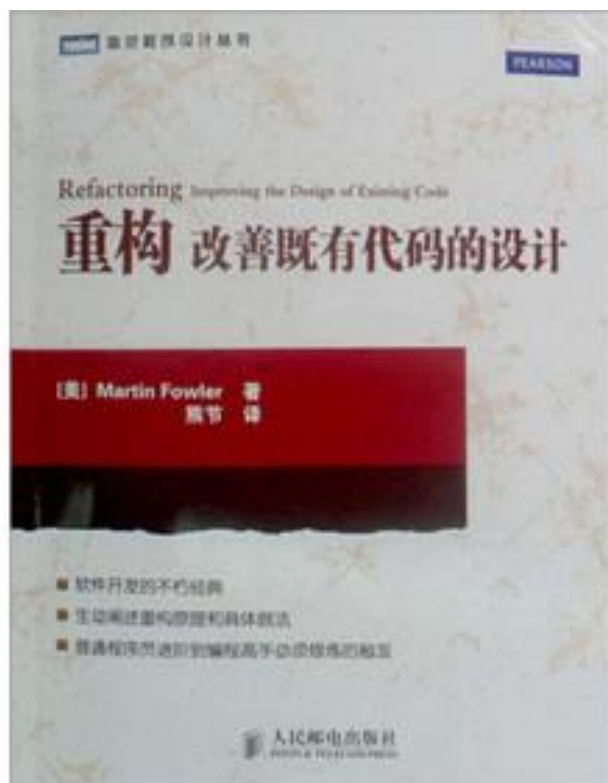
- 点右键，Refactor ( 重构 )

- Rename 重命名(写好单词)
    - Extract Method提取方法(写好函数)





- 《重构—改变既有代码的设计》
  - ▣ Refactoring: Improve the Design of Existing Code
  - ▣ Martin Fowler著





- 代码的坏味道(Bad Smell)意味着应该重构了
- Martin Fowler 提了20多条代码的坏味道及70多种重构的方法
  - 名字不清楚----取好名字
  - 代码重复----代码永远不要写第二遍
  - 代码过长----写简单代码：提炼函数
  - 代码层次过多----写简单代码：提炼函数、使用卫语句、使用Lambda表达式

# 坏味道列表



- 1、 Duplicated Code (重复代码)
- 2、 Long Method (过长函数)
- 3、 Large Class (过大类)
- 4、 Long Parameter List (过长参数列)
- 5、 Divergent Change (发散式变化)
- 6、 Shotgun Surgery (散弹式修改)
- 7、 Feature Envy (依恋情结)
- 8、 Data Clumps (数据泥团)
- 9、 Primitive Obsession (基本型别偏执)
- 10、 Switch Statements (Switch 惊悚现身)
- 11、 Parallel Inheritance Hierarchies (平行继承体系)



# 坏味道列表(续)

- 12、Lazy Class (冗赘类)
- 13、Speculative Generality (夸夸其谈未来性)
- 14、Temporary Field (令人迷惑的暂时值域)
- 15、Message Chains (过度耦合的消息链)
- 16、Middle Man (中间人)
- 17、Inappropriate Intimacy (狎昵关系)
- 18、Alternative Classes with Different Interfaces (异曲同工的类)
- 19、Incomplete Library Class (不完整的程序类库)
- 20、Data Class (单纯的数据类)
- 21、Refused Bequest (被拒绝的遗赠)
- 22、Comments (过多的注释)



# 如何保证代码正确性

- 使用JUnit测试
  - ▣ 重构以后仍然要保证测试是通过的
  - ▣ 在Eclipse中，new—JUnit Case
- 测试驱动开发
  - ▣ (TDD , Test-Driven Development)

# 3. 设计模式







# 设计模式



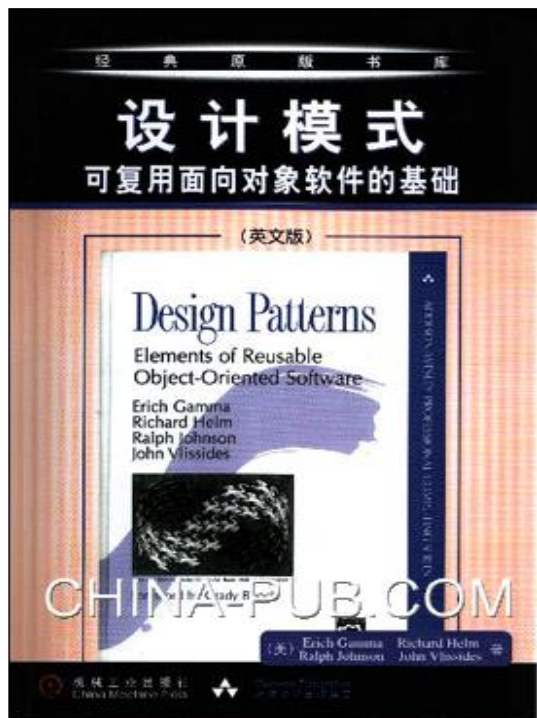


- 设计模式 ( Design pattern )
- 是一套可以反复使用的代码设计经验的总结
- 目的是为了 提高代码的可读性、可维护性
- 其中，最核心的思想就是 “适应变化”



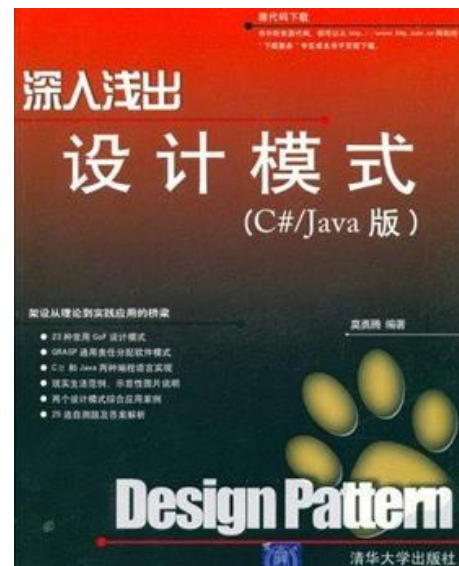
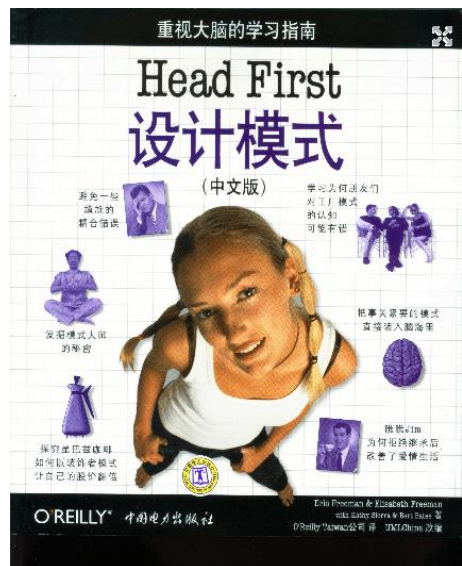
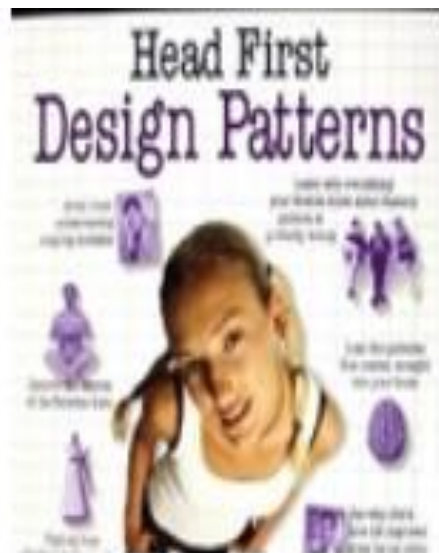
- Design Patterns

▣ GoF (Gang of Four)总结了23种设计模式



# 更易读的书

- 深入浅出 (Head First)设计模式
- 莫勇腾, 《深入浅出设计模式 (C#、Java版)》





# GoF设计模式的分类

- 对已有模式的整理、分类
- 为软件设计总结了宝贵的经验
  - ▣ 这些设计经验可以被重用，但不是简单的代码重用
- 分类：
  - ▣ Creational Patterns
  - ▣ Structural Patterns
  - ▣ Behavioral Patterns





# 经典的23种模式

	创建型 <b>Creational</b>	结构型 <b>Structural</b>	行为型 <b>Behavioral</b>
类	工厂方法(Factory Method)	适配器(Adapter)	解释器(Interpreter) 模板方法(Template Method)
对象	抽象工厂(Abstract Factory) 生成器(Builder) 原型(Prototype) 单态(Singleton)	适配器(Adapter) 桥接(Bridge) 组成(Composite) 装饰(Decorator) 外观(Facade) 享元(Flyweight) 代理(Proxy)	责任链(Chain of Responsibility) 命令(Command) 迭代器(Iterator) 中介者(Mediator) 备忘录(Memento) 观察者(Observer) 状态(State) 策略(Strategy) 访问者(Visitor)



# 设计模式的原则

- 1、单一职责原则。
  - 要把功能尽可能的细分，每一个类应该只负责一块内容或只执行一个任务。那么怎么样才算达到单一职责了呢，那就是当一个类仅有一个引起它变化的原因时。
- 2、开放封闭原则。
  - 尽量不要去修改原有的类，但却可以扩展现有的功能。
- 3、替换原则。
  - 子类必须能够替换它们的基类。
- 4、依赖倒置原则。
  - 高层模块不应该依赖于低层模块，二者都应该依赖于抽象；抽象不应依赖于实现细节，实现细节应该依赖于抽象。
- 5、接口隔离原则。
  - 客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

# JDK中的设计模式



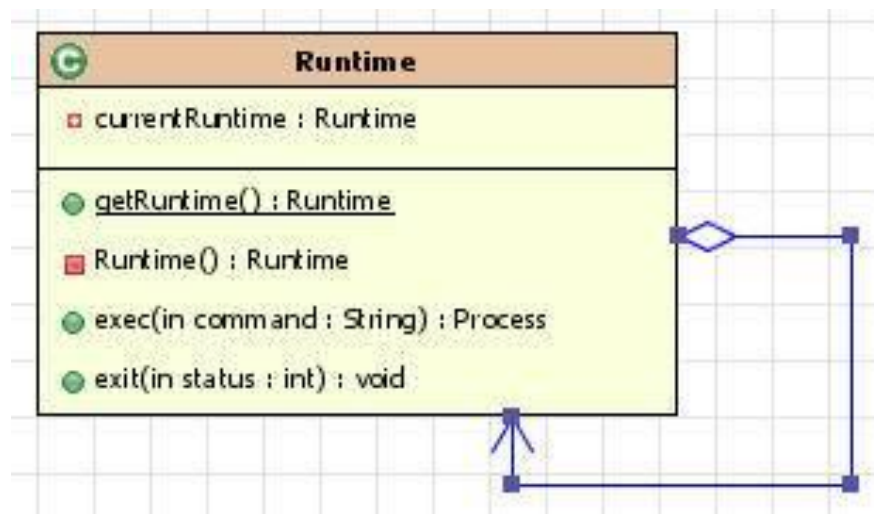
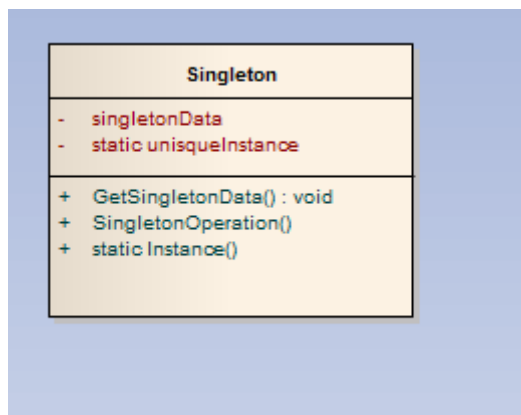
- JDK中使用了大量的设计模式
- 参见
  - ▣ <http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns>

# 创建模式 ( Creational )



## • Singleton:

- Singleton ( 单例 ) 作用：保证类只有一个实例；提供一个全局访问点
- 只允许一个实例。这比static量要好
- java.lang.Runtime#getRuntime()
- java.awt.Toolkit#getDefaultToolkit()
- java.awt.Desktop#getDesktop()





## • Factory:

### □ Factory ( 静态工厂 ) 作用 :

- ( 1 ) 代替构造函数创建对象 ( 2 ) 方法名比构造函数清晰

□ 简单来说 , 按照需求返回一个类型的实例。

□ `java.lang.Class#newInstance()`

□ `java.lang.Class#forName()`

□ `java.lang.reflect.Array#newInstance()`

□ `java.lang.reflect.Constructor#newInstance()`





- **Abstract factory:**

- Abstract Factory ( 抽象工厂 ) 作用 :
- 创建一组有关联的对象实例
- `java.sql.DriverManager#getConnection()`
- `java.sql.Connection#createStatement()`



- **Adapter:**

- **Adapter ( 适配器 )** 作用：使不兼容的接口相容
- 把一个接口或是类变成另外一种。
- `javax.swing.JTable(TableModel)`
- `javax.swing.JList(ListModel)`
- `java.io.InputStreamReader(InputStream)`
- `java.io.OutputStreamWriter(OutputStream)`



## • Composite

- Composite ( 组合 ) 作用：一致地对待组合对象和独立对象
- 让使用者把单独的对象和组合对象混用。
- `java.awt.Container#add(Component)`
- `javax.swing.JComponent#add(Component)`



## • Decorator:

□ Decorator (装饰器) 作用:

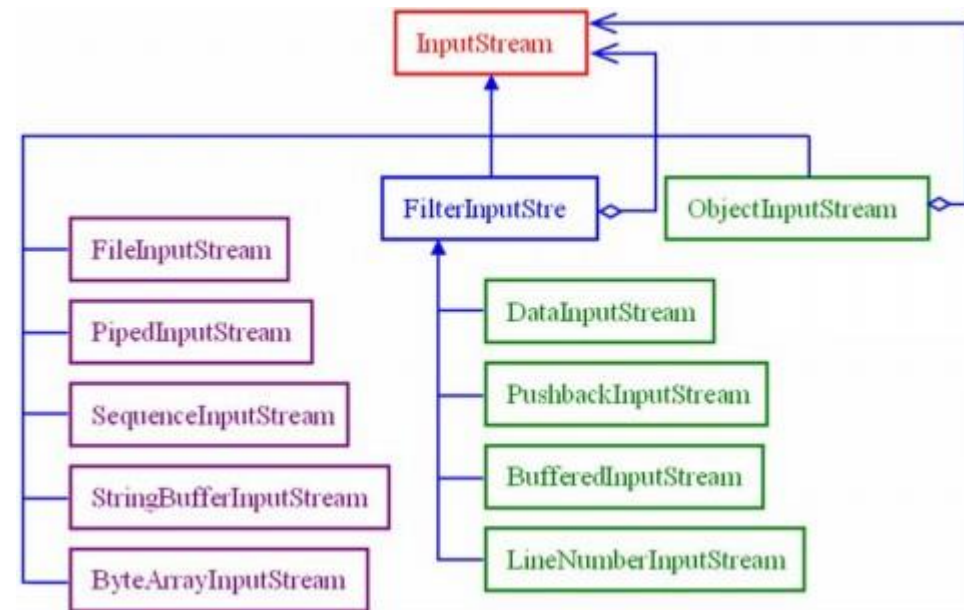
□ 为类添加新的功能; 防止类继承带来的爆炸式增长

□ 为一个对象动态的加上一系列的动作, 而不需要因为这些动作的不同而产生大量的继承类。

□ `java.io.BufferedInputStream(InputStream)`

□ `java.io.DataInputStream(InputStream)`

□ `java.io.BufferedOutputStream(OutputStream)`



# 行为型 ( Behavioral )



- **Observer:**

- **Observer ( 观察者 )** 作用：通知对象状态改变
- 允许一个对象向所有的侦听的对象广播自己的消息或事件。
- `java.util.EventListener`
- `btn.addActionListener(。。。)`



- GRASP ( General Responsibility Assignment Software Pattern ) 是通用职责软件分配模式。
  - 1 Information Expert ( 信息专家 )
  - 2 Creator ( 创造者 )
  - 3 Low Coupling ( 低耦合 )
  - 4 High Cohesion ( 高内聚 )
  - 5 Controller ( 控制器 )
  - 6 Polymorphism ( 多态 )
  - 7 Pure Fabrication ( 纯虚构 )
  - 8 Indirection ( 间接 )
  - 9 Protected Variations ( 受保护变化 )



# 4 反射





# 反射





- 反射 ( reflection)
  - ▣ 在运行状态中，对于任意一个**类**，都能够知道这个类的所有属性和方法；对于任意一个**对象**，都能够调用它的任意一个方法和属性。
- 在一些框架性的程序中，反射是相当重要的
  - ▣ 如 plugin



- `java.lang.reflect.*`
- 首先要得到类的Class
- 得到Class对象的三种方法
  - **类名.class**
    - `Class<?> cls = String.class ;`
  - **对象.getClass()**
    - `String str = "abc" ; Class<?> cls = str .getClass() ;`
  - **Class.forName(类的全名)**
    - `Class<?> cls = Class.forName( "java.lang.String" );`



# 得到字段及方法

- 由Class获得该类的信息
  - ▣ 得到成员（字段、方法）
  - ▣ 例：[reflect\ClassViewer.java](#)



# 动态创建对象

- 由Class来创建相关的实例、调用相关的方法
  - ▣ 例：reflect\PlayerMaster.java    ReflectionTest.java
- 应用示例
  - ▣ 加了反射功能的 reflect\HTTPServer.java





# 再谈Annotation

- Annotation：注记，有译为注释、注解、标记、元数据
- JDK内置的Annotation
  - @Override
    - 表示覆盖
  - @Deprecated
    - 表示过时
  - @SuppressWarnings({"unchecked","deprecation"})
    - 表示不产生警告信息



# 自定义注解

- 注解的定义

- 使用 `@interface` 来定义一个 **类型**，表示它是一个注解
- 使用 **方法名()** 表示它的一个 **属性**（**值或数组**）
  - （其中 `value()` 是默认属性）使用 `default` 表示其默认值
    - `// 定义一个注解`
    - `@Target(ElementType.METHOD)` `// 这个表明可以用于方法上`
    - `@Retention(RetentionPolicy.RUNTIME)` `// 这个表明可以用反射来读取`
    - `@Documented` `// 这个表明它会生成到javadoc中`
    - `@interface DebugTime{`
    - `boolean value() default true;`
    - `long timeout() default 100;`
    - `String msg();`
    - `int [] other() default {};`
    - `}`



□@笔记名

□(属性=值, 属性={值,值} )

- //使用笔记
- class MyClass
- {
- @DebugTime(value=true, timeout=10, msg="时间太长", other={1,2,3} )
- public double fib(int n){
- if(n==0||n==1) return 1; else return fib(n-1)+fib(n-2);
- }
- }



# 用反射来读取注记

- `method.getAnnotation(注记.class)`
- `method.getAnnotations()`
- 示例 `annotation\DebugTool.java`