

# 《面向对象设计与构造》课程

## Lec1-对象与对象化编程(上)

2019

OO课程组

北京航空航天大学

# 内容提要

- 课程介绍
- 过程式程序回顾
- 为什么引入对象
- 对象化程序的构成
- 对象是什么
- 作业

# 课程介绍



- 三个关键词
  - 设计(design)与构造(build)==》工程化开发
  - 面向对象==》系统化的思维方式
- 讨论：软件与程序的区别
- 讨论：如何说明你所写程序有多好？
- 课程目标：掌握以工程化方法来开发高质量复杂软件系统的能力
  - 工程化方法：综合<sup>红</sup>分析软件功能和性能约束，综合<sup>红</sup>考虑相应约束进行设计和实现，并能使用测试和逻辑分析等手段进行综合<sup>红</sup>验证和优化
  - 高质量：能够使用技术手段来表明/论证所开发的软件质量是否满足要求

数据结构设计  
与应用能力

基于约束的需  
求分析能力

基于需求的规  
格化设计能力

线程安全  
设计能力

关注性能的实  
现与测试能力

基于规格的程序  
正确性论证能力

算法与工程  
优化能力

微型团队  
协同能力

# 体系化的课程

“昆仑课程”  
(二下春季)



“补给站课程” (二下暑期)

# “昆仑课程” 知识点设置

对象特性

OO构造机制

层次化设计

对象运行机制

线程交互机制

线程安全设计

过程规格设计

类规格设计

设计原则

自动化测试

基于规格的程序  
正确性证明

程序的模型化  
表示

# “昆仑课程” 核心规则

- **32(授课)+16(实验)+16(研讨)**学时，3学分，必修课
- 内容分4个模块，每个模块包括4次授课、2次实验和2次研讨
  - 3次/周介绍新内容
    - 每次一个程序作业、每次一个测试作业
  - 1次课程作业问题分析
    - 对各自的程序问题和测试问题进行总结分析，撰写技术博客
  - 2次实验围绕单元教学内容进行实践训练和分析
    - 每次实验当堂完成实验和在系统中完成相应报告
  - 2次围绕作业和课程内容的研讨
    - 组织同学们交流心得体会，邀请企业界大咖介绍相关经验
- 平衡与综合的测试
  - 公共测试
  - 竞争性互测

# “昆仑课程” 核心规则

- 采用Java语言
  - OO概念支持、类型安全、（编译器）强大的静态检查能力、跨平台
  - 尽快上手学习Java！！！！
- 成绩评定：综合排序
  - 作业成绩：65%
    - 作业完成质量和测试
  - 实验成绩：25%
    - 完成度和质量
  - 研讨与博客成绩：10%
    - 参与度、贡献度和质量

# 高质量之道

- 精心设计
  - 把复杂问题简单化，尽可能减少理解代价
  - 遵循业已形成的设计原则和规范
- 严格测试
  - 抄袭检测
  - 代码风格检测
  - 公测
  - 互测



# 台阶式的公测为你指路

- 中测
  - 基础测试：关注基本功能的正确性
    - 有效性检查的依据：100%
  - 进阶测试：功能覆盖和鲁棒性检查
    - 进入互测的依据：100%
- 强测
  - 目的：深度的组合式测试，关注功能和性能
  - 进入互测的依据：至少通过一个强测用例
- 在开发期间，可以按照一定配额使用中测服务，利用系统反馈来提高代码质量。配额用完后，仍然分配一定的抵扣性配额，每使用一次都会导致扣一些最终的测试分。

# 互测不再是两个人的游戏

- 互测分为三个等级
  - 每个等级下设多个互测ROOM
  - 每个同学按照其公测成绩等确定其参与的互测等级，并随机分配到相应的ROOM中
  - 每个同学在互测期间不知道自己在哪个ROOM中，所有的ROOM都采用统一的编码
- 互测期间可随意查看同处一个ROOM中的代码，提交测试数据，如果发现了bug，则算作有效测试，并被测试系统采纳
- 测试系统对所有有效互测测试输入，对一个ROOM中的所有程序进行测试
  - 测试得分：Hack了一个ROOM中的多少个程序
  - 被测失分：被Hack了多少个bug
  - 被测失分可以通过后续的bug修复找补回来

# 鼓励修复bug

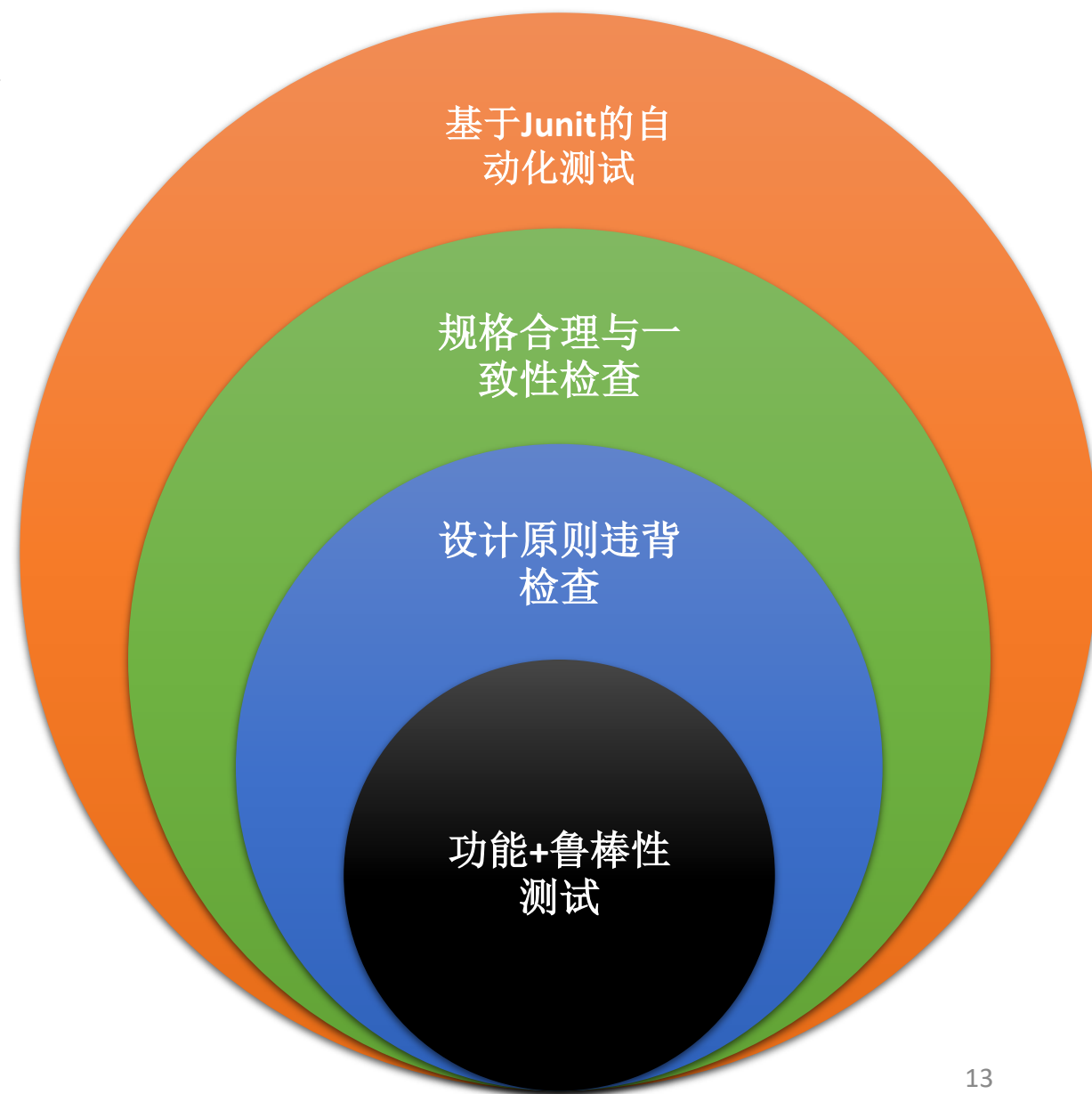
- 测试的目标是发现bug，但质量提升不能止于测试
- 在一次作业完成后的一周内，每个同学都可以积极去修复bug，从而找回测试阶段被扣掉的分数
  - 消除多个测试用例发现相同bug导致的重复扣分影响
- 一旦提交bug修复，系统会做严格检测
  - 使用所有的强测用例和互测用例
  - 不能引入新的bug
  - 成功修复所声明要修复的bug
- 具体得分规则见课程规则文件

# “昆仑课程” 核心规则

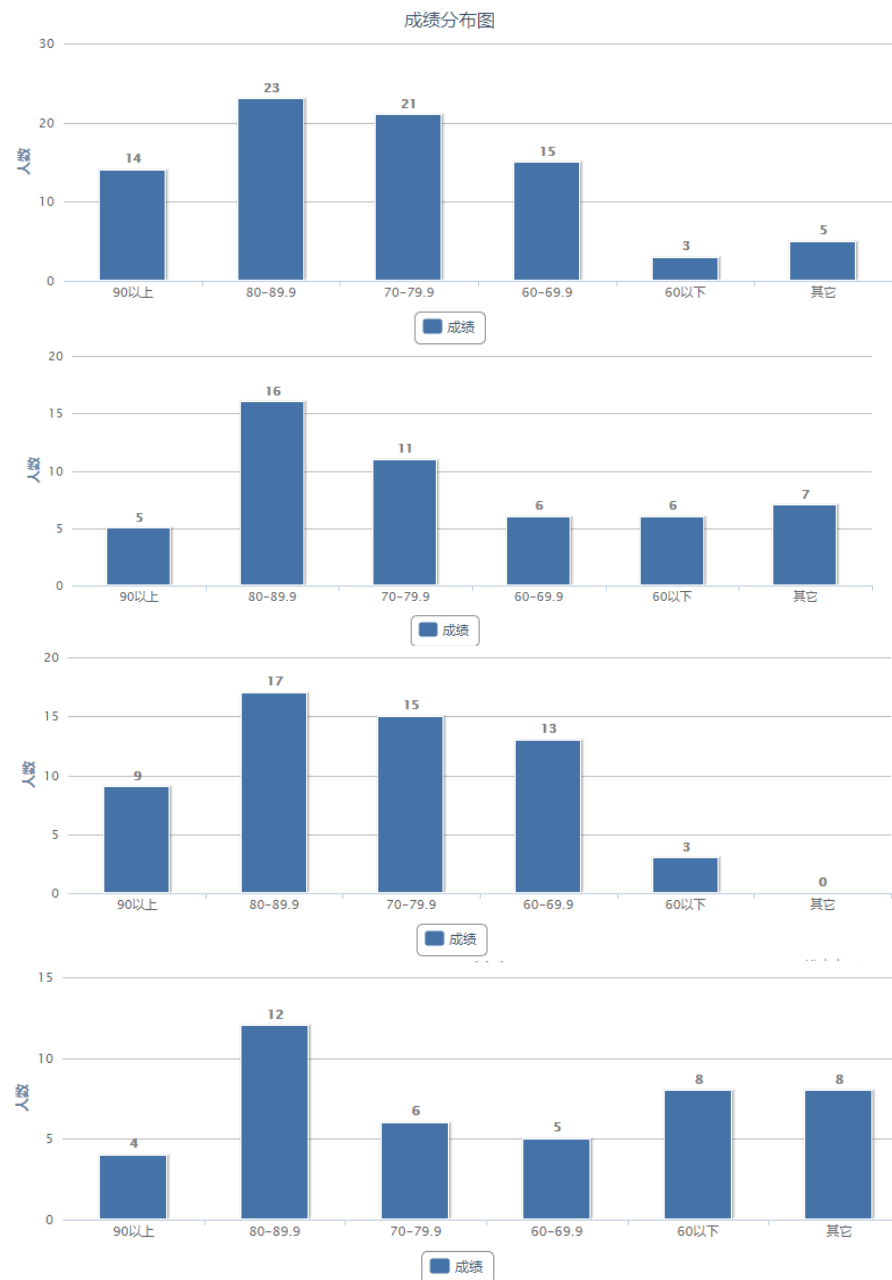
- 作业抄袭？
  - 抄袭检测是多年来的必备良药
  - 发现一次抄袭，取消作业成绩
- 胡乱交一个？
  - 无法通过中测的基础测试会被判为无效作业
  - 累积五次无效作业，取消作业成绩！
- 通过率？
  - “补给站”已准备好欢迎有困难的同学！
  - 补给站也是有进入条件的（无效次数在[6,10)之间）！！！！
- 采用改进了的自动化测评系统
  - 作业提交自动从你的git库中提取代码，不需要你手动上传
  - 确保每次代码更新都要提交到git库
  - 全程作业进行测试

# 强调测试的课程作业

- **Common Testing**
  - 针对作业的功能和性能要求，精心设计的测试用例
- **Double Blind Testing**
  - 你不知道谁将测试你的程序
  - 你不知道你测试的是谁的程序
- **Comprehensive Scoring**
  - 作业提交情况(+)
  - 通过的公共测试用例数(+)
  - 被发现的bug数(-)
  - 发现的bug数(+)
  - 修复的bug数(+)



# “昆仑课程”的能力目标



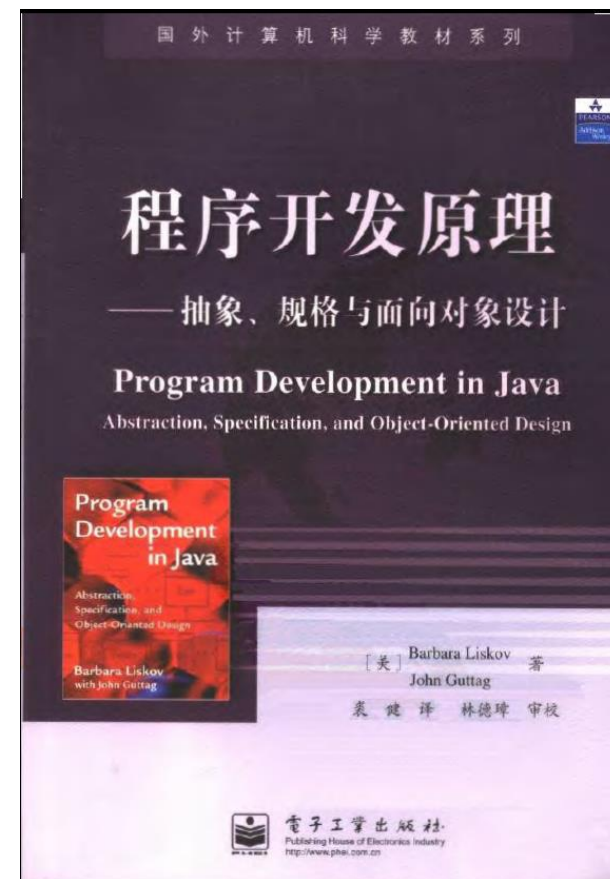
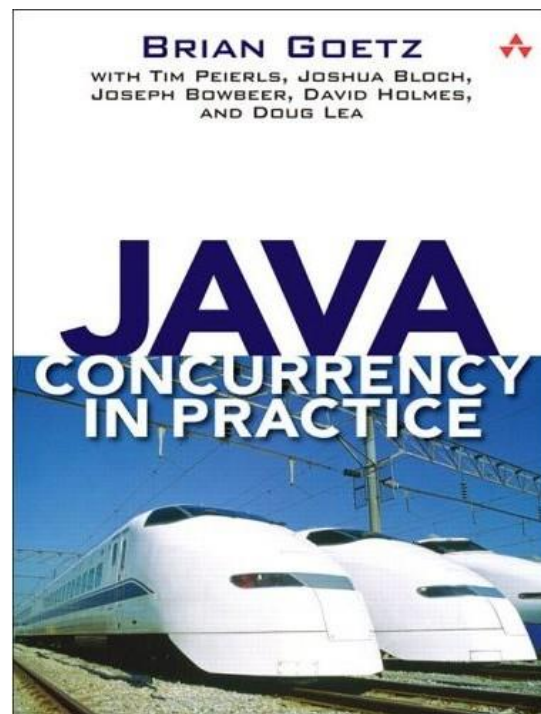
# “昆仑课程”参考材料

- 教材

- 程序开发原理—抽象、规格与面向对象设计  
(Barbara Liskov, John Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*)
- Java Concurrency in Practice

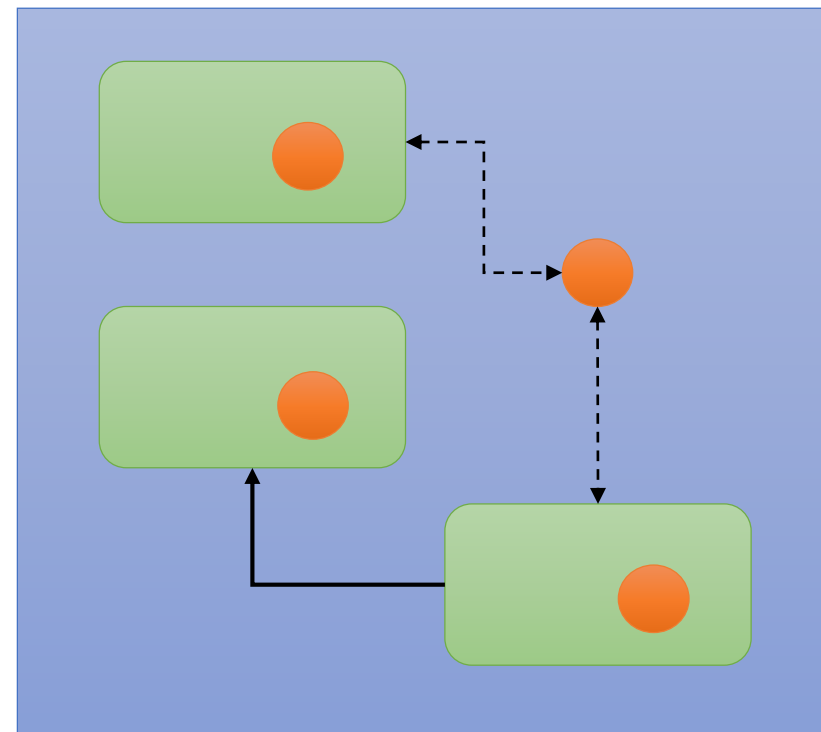
- 互联网

- 百科
- Jdk guideline
- Stackoverflow.com
- 技术博客



# 过程式程序回顾

- 结构化
  - 功能结构：模块、函数
  - 数据结构：类型、变量(全局、局部)
  - 组合结构（交互机制）：函数调用、变量共享
- 面向过程(procedure)
  - 是一种自然的思维方式：按照“自然过程/业务流程”来设计程序
  - 过程分解/业务分解
  - 提取公共过程





# 过程式程序回顾

- 模块表现形式
  - 物理意义上的模块：exe、lib、dll文件等
  - 逻辑意义上的模块：多个相关函数的集合体（.c文件+.h文件）
- 函数
  - 具有一定计算能力、相对独立的编程单位
  - 公共功能函数：围绕数据结构实施所需的计算和处理，如字符串处理、栈和队列处理函数等
  - 特定功能函数：直接源自于软件功能分解得到的函数，如学生注册、输入/输出函数等

# 过程式程序回顾

- 公共功能函数 vs 特定功能函数
  - 功能实现
    - 一般化 vs 特定程序功能
  - 调用场景
    - 不确定 vs 确定
  - 易变性
    - 不随程序功能变化而变化 vs 随程序功能变化而变化
  - 重用性
    - 高 vs 低
  - 测试难度
    - 高 vs 低

# 过程式程序回顾

- 函数调用是一种重要的程序组合机制
  - 调用者功能场景 vs 被调用者预期的功能场景
  - 形参与实参的匹配
  - 返回值的处理
- 变量
  - 全局变量：多个函数要使用 and 处理的变量，如电梯系统的电梯状态变量。
  - 局部变量：一个函数内部要处理的数据表示。
  - 临时变量：便于代码编写的一些临时变量，如循环变量、中间计算结果存储等

三者之间有什么关系？

# 过程式程序回顾

- 丰富的类型
  - 原子类型
  - 结构类型(struct)
  - 联合类型(union)
- 丰富的数据组织与使用方式
  - 数组、列表
  - 树与图
  - 指针

# 为什么引入对象

- 编码视角
  - 为什么多个函数需要共享访问数据（变量）？
    - 这些函数之间具有逻辑“聚合”的特性
  - 如何让一个函数使用之前运行所产生的中间数据？
    - 增加全局变量
    - 或者，使用外部存储
  - 如何管理逻辑相关的函数+变量？
    - 聚合在一个文件中

# 为什么引入对象

- 程序设计视角
  - 需要一种手段来封装逻辑相关的函数和数据
  - 需要一种手段，通过数据类型来绑定和使用其相应的处理
  - 可以不使用全局变量来进行模块组合
- 程序思维视角
  - 按照数据处理流程来设计模块
  - 按照数据及其状态变化来进行管理
  - 按照数据的层次化来建立抽象

# 面向对象程序的构成

- 类
  - 属性(数据)、操作及其实现
  - 作用域
- 接口
  - 抽象操作
- 关系
  - 继承：类型层次+重用
  - 关联：数据聚合+调用
  - 实现：为多种数据抽象提供统一接口
- 入口类
  - 提供入口函数main（静态函数）

包

模块层次  
作用域

# 面向对象程序的构成

- 过程式程序与面向对象程序的特点对比

## 过程式

- 强调函数分解
- 程序由函数组成
- 运行时由函数和数据表示
- 函数之间共享全局数据
- 函数之间传递数据

## 面向对象式

- 强调数据抽象
- 程序由类组成
- 运行时由对象表示
- 数据得到隐藏和保护
- 对象之间通过消息交互



# 面向对象程序的构成

- 以类作为基本的编程单位
- 类封装了数据和函数
- 类之间协作完成程序的功能
  - 有哪些协作方式？

```
public class Num {  
    // class providing useful numeric routines  
  
    public static int gcd (int n, int d) {  
        // REQUIRES: n and d to be greater than zero  
        // the gcd is computed by repeated subtraction  
        while (n != d)  
            if (n > d) n = n - d; else d = d - n;  
        return n;  
    }  
  
    public static boolean isPrime(int p) {  
        // implementation goes here  
    }  
}
```

# 对象与类

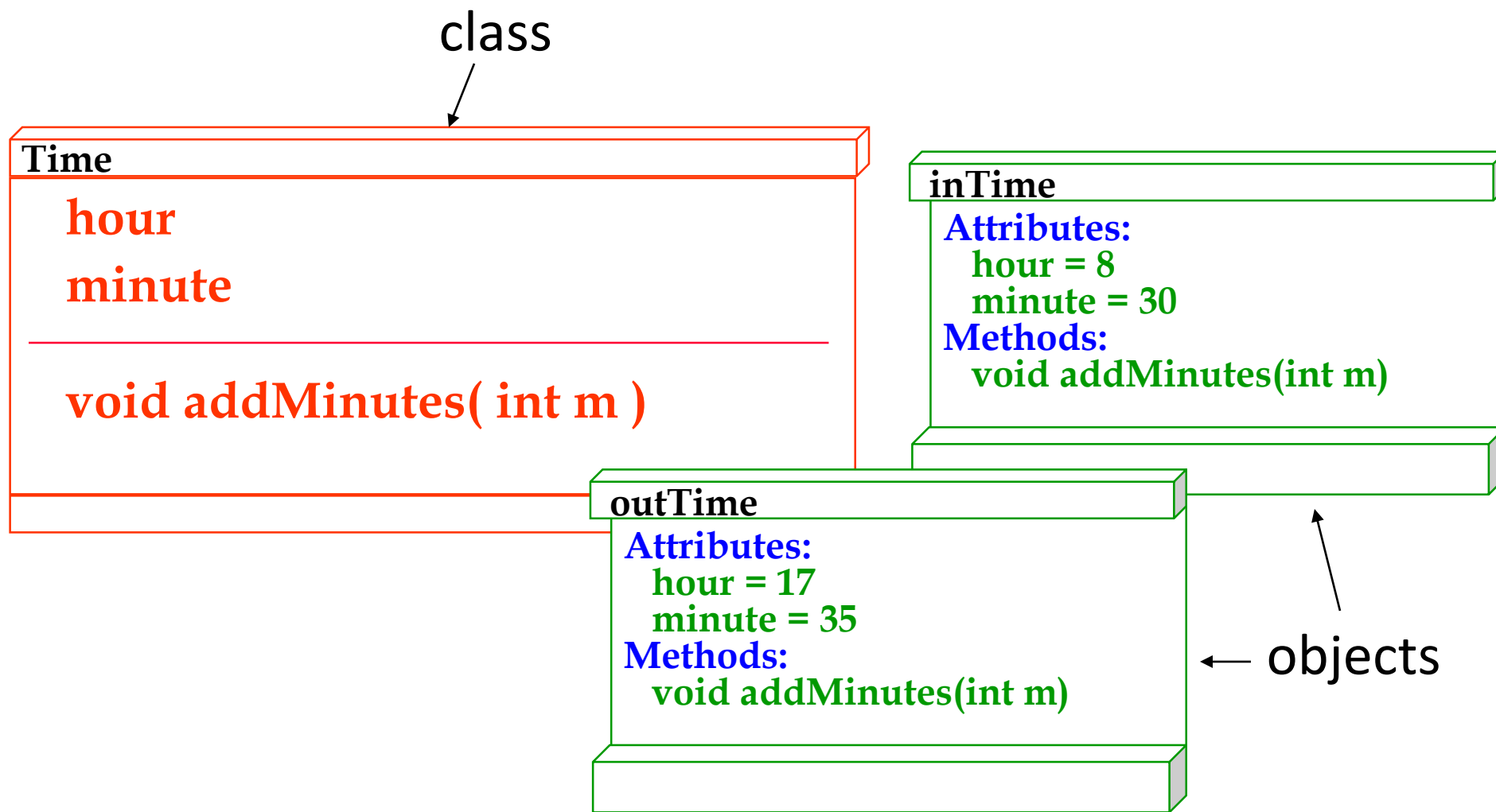
- 在面向对象程序中，我们称对象是类的**实例化**结果
  - 对象是**运行时**概念
  - 类是**规格**概念
- 类是通过关键词class定义的一个程序单位
  - `public class A {...}`
- 对象是方法中定义的变量（类型为某个类）
  - `A a = new A(...);`
- 一个对象可以通过多个变量来引用
  - `A b = a;`



# 对象与类

```
class Time {  
    private int hour, minute;  
  
    public Time (int h, int m) {  
        hour = h;  
        minute = m;  
    }  
  
    public void addMinutes (int m) {  
        int totalMinutes =  
            ((60*hour) + minute + m) % (24*60);  
        if (totalMinutes<0)  
            totalMinutes = totalMinutes + (24*60);  
        hour = totalMinutes / 60;  
        minute = totalMinutes % 60;  
    }  
}
```

# 对象与类




# 对象与类

- 类提供了构造对象的模板
  - 规定了对象拥有的数据及其类型
  - 规定了对象能够执行的动作
  - 规定了对象状态的变化空间
- 如果在程序中定义了一个类A，就意味着可以在类B中构造A的变量以实现B的方法/功能
  - B可以是A本身
- 每个类都应该提供相应的构造器，用来在构造对象时初始化和设置对象的初始状态

# 对象与类---构造器

```
class Time {  
    private int hour, minute;  
  
    public Time (int h, int m) {  
        hour = h;  
        minute = m;  
    }  
  
    public void addMinutes (int m) {  
        int totalMinutes =  
            ((60*hour) + minute + m) % (24*60);  
        if (totalMinutes<0)  
            totalMinutes = totalMinutes + (24*60);  
        hour = totalMinutes / 60;  
        minute = totalMinutes % 60;  
    }  
}
```

*constructor* for Time



# 对象与类

- 对象是一个具有计算能力的实体
  - 封装(*Encapsulate*) 其状态，对外部屏蔽细节
    - 状态由对象所有属性变量的取值联合确定
    - 例如Time类中的hour和minute属性，(22,10)表示晚上时间状态，(11,30)则表示白天时间状态
  - 能够在相应状态上执行动作即方法 (*method*)
    - 在不同状态下执行方法的效果可能会不同
  - 通过消息传递机制与其他对象交互(*message passing*)
    - 消息: *object.method(p1,p2,...,pm)*
    - 和函数调用存在本质上的不同(后面会解释)

# 对象与类---构造对象

```
class Time {  
    private int hour, minute;  
  
    public Time (int h, int m) {  
        hour = h;  
        minute = m;  
    }  
    ...  
}
```

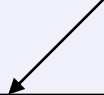
```
Time inToWork = new Time(8, 30);  
Time outFromWork = new Time(17, 35);
```



# 对象与类---执行方法

```
class Time {  
    private int hour, minute;  
    public Time (int h, int m) {  
        hour = h;  
        minute = m;  
    }
```

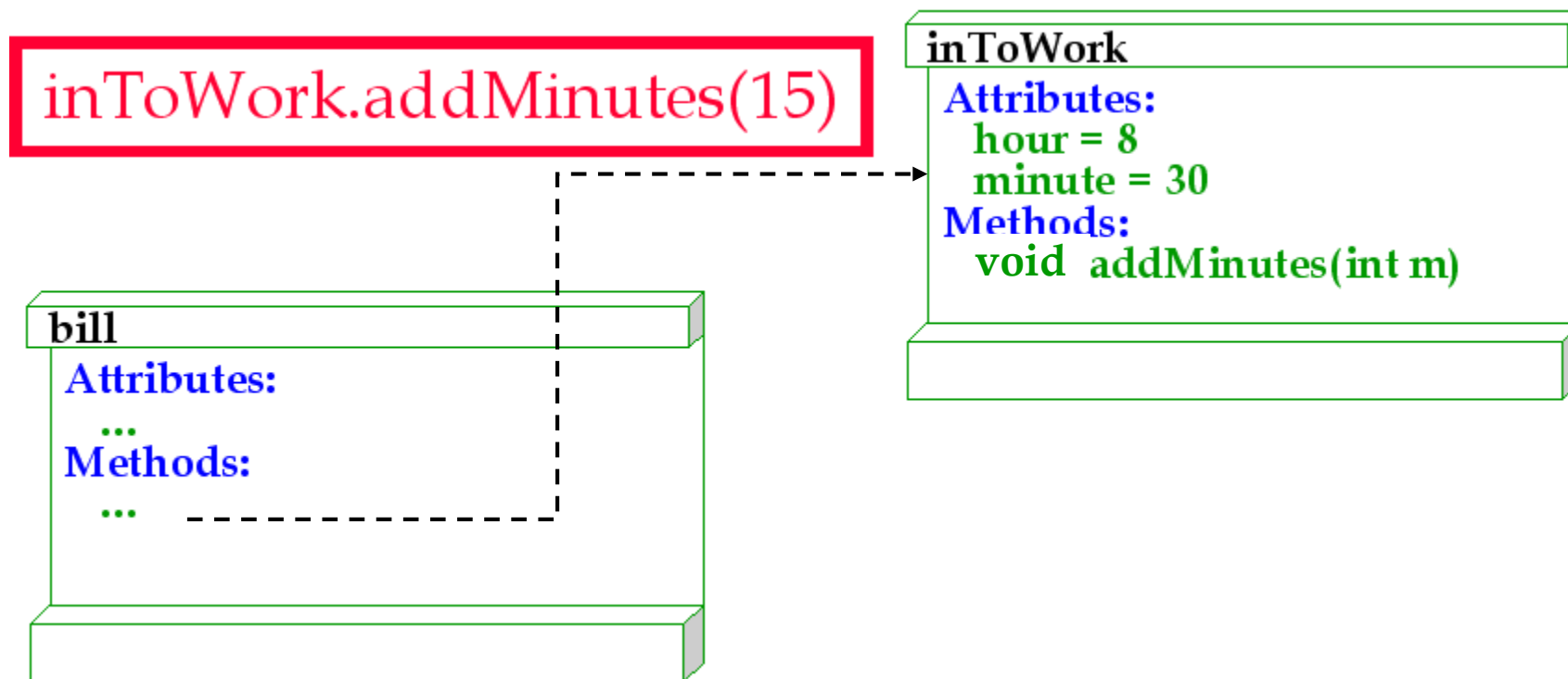
该方法能够根据对象状态来进行相应的计算



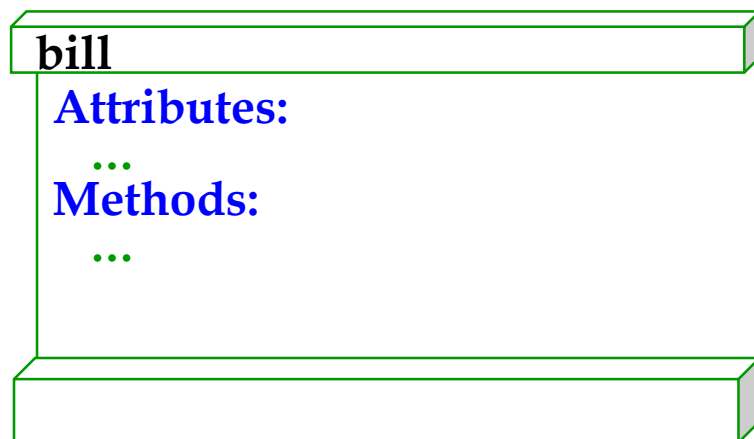
```
        public void addMinutes (int m) {  
            int totalMinutes =  
                ((60*hour) + minute + m) % (24*60);  
            if (totalMinutes<0)  
                totalMinutes = totalMinutes + (24*60);  
            hour = totalMinutes / 60;  
            minute = totalMinutes % 60;  
        }
```

```
}
```

# 对象与类---对象交互



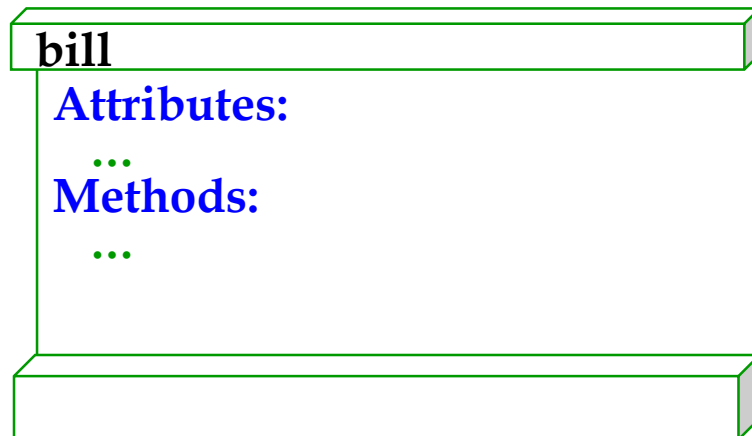
# 对象与类---对象交互



# 对象与类---对象交互

假设在**bill**的某个方法中有下面的代码:

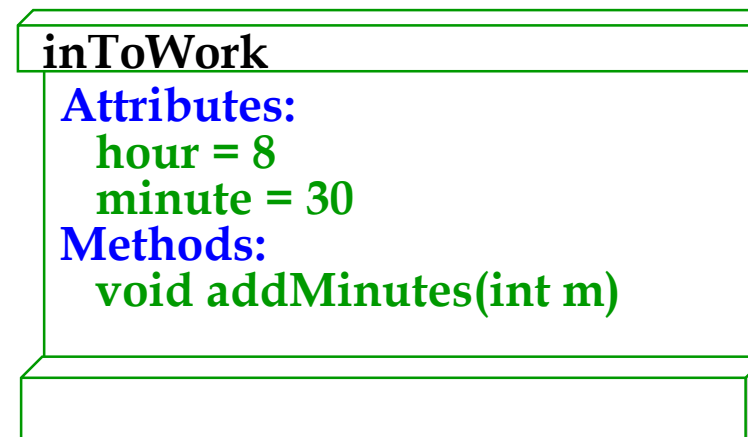
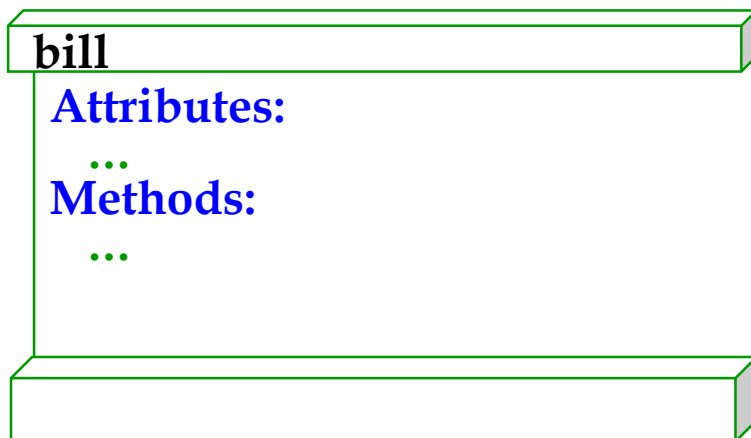
```
Time inToWork = new Time(8, 30);  
inToWork.addMinutes(15);
```



# 对象与类---对象交互

假设在bill的某个方法中有下面的代码:

```
➡ Time inToWork = new Time(8, 30);  
   inToWork.addMinutes(15);
```

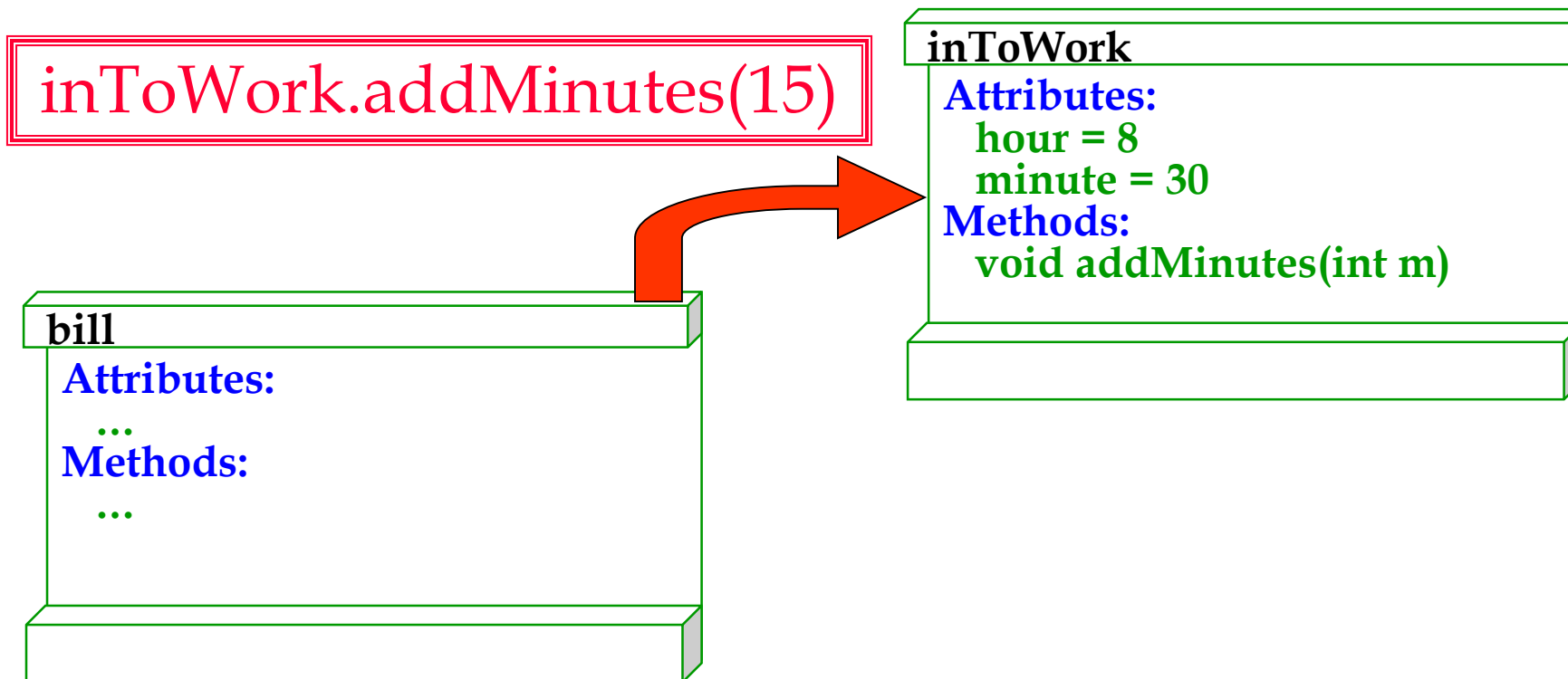


# 对象与类---对象交互

假设在bill的某个方法中有下面的代码:

```
Time inToWork = new Time(8, 30);
```

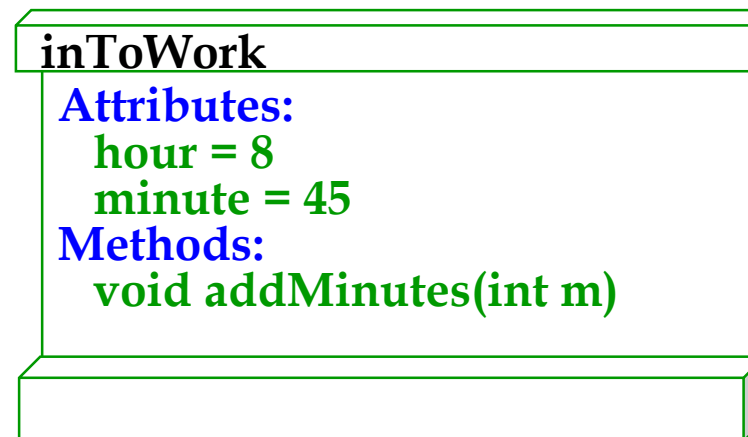
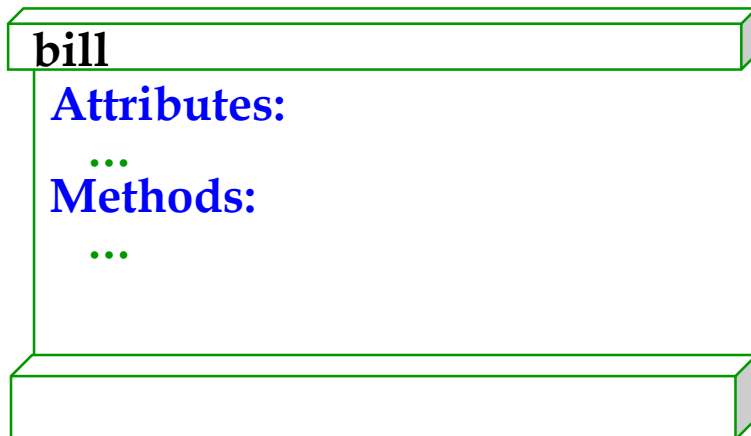
```
➡ inToWork.addMinutes(15);
```



# 对象与类---对象交互

假设在bill的某个方法中有下面的代码:

```
Time inToWork = new Time(8, 30);  
inToWork.addMinutes(15);
```



# 对象与类---类的结构

```
class name [extends ***][implements ***] {
```

*declarations*

← 属性和枚举常量

*constructor definition(s)*

← 对象构造和初始化手段

*method definitions*

← 对象状态查询与控制手段

```
}
```

这三部分之间没有次序规定



# 对象与类

- 从程序语法上看，似乎面向对象程序与过程式程序差别并不大
  - 都有数据结构
  - 都有过程式函数
  - 都有变量
  - 都有唯一的入口点main
- 差别在于
  - 过程式程序通常按照流程分解来设计开发
  - 面向对象程序按照数据抽象与处理来设计开发

# 过程式程序与面向对象程序的比较分析

- 假设要设计实现一个多项式加减运算程序
  - 多项式:  $c_0+c_1x+c_2x^2+...+c_mx^m$
  - $c_2x^2$ 为一个项, 其中2为该项的幂/阶(degree),  $c_2$ 为系数
  - 多项式的阶为其所有项中的最高阶
  - 多项式加减运算对应为阶数相同的项的系数加减
- 我们分别按照过程式程序设计和面向对象程序设计来实现

# 过程式程序设计实现

- (1)首先定义数据结构来表示多项式

```
struct Poly{  
    int coeff [];  
    int degree [];  
}
```

```
struct Poly{  
    Term pTerm [];  
}  
struct Term{  
    int coeff;  
    int degree;  
}
```

- (2)然后设计实现多项式加法和减法运算函数

```
void PolyAdd(Poly *p1, Poly *p2)  
//p1+p2 → p1  
{  
    ...  
}
```

```
void PolySub(Poly *p1, Poly *p2)  
//p1-p2 → p1  
{  
    ...  
}
```

# 过程式程序设计实现

- (3) 设计主函数main
  - (3.1) 读取多项式运算式，并构造多项式变量

```
//suppose user inputs poly in (c1,n1),(c2,n2)...  
while (...){  
    scanf("(%d,%d)", &c, &n);  
    Poly p1 = malloc (sizeof(Poly));  
    ...  
    //为p1中的coeff和degree申请内存，但是不知道输入的多项式的阶是多少？  
}  
//suppose here we have two polys: p1 and p2
```

- (3.2) 调用加减法函数进行预算
- (3.3) 输出计算结果

# 对象式程序设计实现

- (1)数据抽象

- 多项式如何表示？如何构造？外部关心它哪些状态？该软件需要对它进行什么处理？

```
public class Poly{  
    private int[] terms;  
    private int deg;  
    public Poly(int deg) {...}  
    public Poly(int c, int n){...}  
  
    public int degree(){return deg;}  
    public int coeff(int d){...}  
  
    public Poly add(Poly q){...}  
    public Poly sub(Poly q){...}  
}
```

# 对象式程序设计实现

- (2)设计主类和实现入口函数main
  - (2.1)主类管理多项式对象
  - (2.2)主类main必须是public static void main
  - (2.3)在main中构造主类对象来管理相关对象
  - (2.4)主类提供读取多项式操作，通过Poly构造多项式对象，并提供多项式计算操作

```
public class ComputePoly{
    private Poly polyList[];
    private Operator opList[];
    private int num;
    enum Operator{ADD, SUB};
    public ComputePoly() {...}
    private void parsePoly(String s){...}
    private void compute(){Poly p = polyList[0]; Poly p1,p2;
        for(int i=1;i<num; i++){p2 = polyList[i];Operator op=opList[i-1];
            if(op==ADD)p1=p.add(p2); if(op==SUB) p1=p.sub(p2); p=p1;
        }
    }
    private void parseOperator(String s){...}
    public static void main(String args[]){
        //从console获取用户输入的多项式计算表达式: String
        ComputePoly cp = new ComputePoly();
        cp.parsePoly(s); cp.parseOperator(s);cp.compute();
    }
}
```

# 对比分析

- 过程式程序
  - 每个函数都必须了解数据结构的全部细节。一旦Poly数据结构修改怎么办？
  - main函数的工作量非常大
  - main函数中的变量非常多，难以管理
- 对象式程序
  - 每个类管理着它应该管理的数据，外部无法访问
  - 每个类的操作只处理该类所管理的数据
  - 每个类对外提供状态查询操作
  - Poly对象一旦构造之后不允许改变它

# 对比分析

- 过程式程序
  - `main`函数控制整个流程，按照计算步骤来初始化相关变量、调用相关函数来处理相关变量等
  - 一旦相关函数和数据发生变化，`main`函数必须进行调整
- 对象式程序
  - `PolyCompute`类管理`polyList`, `opList`, `num`, 不关心`Poly`是什么
  - `PolyCompute`负责与用户交互构造和管理相应的`polyList`, `opList`, 并维护好状态`num` (在`Poly`对象的协助下)
  - `PolyCompute`负责按照`opList`来对`polyList`进行计算（在`Poly`对象协助下）
  - `Poly`对象负责...



# 需要掌握的结构

- 模块结构
  - 模块的基本单位是类
  - 按照数据及其处理来识别类
  - 按照数据之间的关系来构造模块间的关系
- 数据管理结构
  - 一个类如何管理具体的数据

# 模块结构

- 一个程序真正要处理的核心数据有哪些？
  - 按照数据特征构造相应的类，来管理数据和提供相应的数据处理行为
- 数据从哪里来，到哪里去？
  - 程序需要一个或多个类来专门处理数据输入和输出
- 程序必须要有一个控制框架
  - 提供入口方法
  - 管理程序执行过程中实际构造的对象
- 建立这三种不同角色的类之间关系

# 数据管理结构

- 单体数据
  - 单体变量
- 复合数据
  - 建立数据之间的关系
  - 建立类之间的关系
- 一组数据的管理
  - 使用数据**容器**
  - 规模是否静态可知？
  - 数据是否在运行中动态获得？

# 数据管理结构

• 手

• A

Modifier and Type	Method and Description
boolean	<code>add(E e)</code> Ensures that this collection contains the specified element (optional operation).
boolean	<code>addAll(Collection&lt;? extends E&gt; c)</code> Adds all of the elements in the specified collection to this collection (optional operation).
void	<code>clear()</code> Removes all of the elements from this collection (optional operation).
boolean	<code>contains(Object o)</code> Returns <code>true</code> if this collection contains the specified element.
boolean	<code>containsAll(Collection&lt;?&gt; c)</code> Returns <code>true</code> if this collection contains all of the elements in the specified collection.
boolean	<code>equals(Object o)</code> Compares the specified object with this collection for equality.
int	<code>hashCode()</code> Returns the hash code value for this collection.
boolean	<code>isEmpty()</code> Returns <code>true</code> if this collection contains no elements.
<code>Iterator&lt;E&gt;</code>	<code>iterator()</code> Returns an iterator over the elements in this collection.
boolean	<code>remove(Object o)</code> Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	<code>removeAll(Collection&lt;?&gt; c)</code> Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	<code>retainAll(Collection&lt;?&gt; c)</code> Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	<code>size()</code> Returns the number of elements in this collection.
<code>Object[]</code>	<code>toArray()</code> Returns an array containing all of the elements in this collection.
<code>&lt;T&gt; T[]</code>	<code>toArray(T[] a)</code> Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

很多

# 数据管理结构

- HashMap

- Map类型，存储key-value对
- Key用来获取Map中所存储元素的Value
- 可以遍历Entry集合（key-value）、Key集合和Value集合
- 不能通过下标来遍历访问，只能通过Iterator（迭代器）

- HashSet

- Set类型，使用HashMap实现的
- 不能通过下标来遍历访问，只能通过Iterator（迭代器）

```
Iterator iter = map.entrySet().iterator();  
while(iter.hasNext()) {  
    Map.Entry entry = (Map.Entry)iter.next();  
  
    key = (String)entry.getKey();  
    integ = (Integer)entry.getValue();  
}
```

```
// 假设set是HashSet对象  
for(Iterator iterator = set.iterator();  
    iterator.hasNext(); ) {  
    iterator.next();  
}
```

# 对象与类：小结

- 类是从数据视角的抽象结果
  - 不仅仅是数据结构定义
  - 从程序功能角度，把与特定数据相关的操作封装在一起
  - 一个类只管理和这个类职责密切相关的数据
  - 类之间形成层次和协作结构
  - 类的内部对外部不可见，只要确保相关方法规格不发生变化，类的内部细节变化就不会导致使用者跟着变化
- 对象是运行时数据抽象
  - 谁创建对象
  - 谁管理对象

# 高质量来自哪里？

- 什么叫质量
  - 外部质量：关注软件对需求的满足程度
  - 内部质量：关注软件内部结构和代码的易理解性、模块独立性等
- 本课程强调两方面的质量
  - 通过公测强调外部质量
  - 通过代码风格检查强调内部质量
  - 通过互测把内部质量与外部质量关联
- 高质量来自哪里？
  - 外部质量来自于内部质量
  - 内部质量来自于精细化设计

# 再次讨论课程目标

- 逻辑清晰的程序
  - 结构清晰
  - 命名清晰
- 严密的编程思维
  - 防御
  - 破坏
- 规范的编程思维
  - 规格
- 系统性的工程思维
  - 目标与代价的平衡
  - 严守deadline



# 作业分析

- 多项式求导
  - 多项式由项组成，每个项都是一个幂函数
- 准备工作
  - 熟悉求导操作，注意我们是符号上的求导操作，不是具体求值。
- 求导的本质是什么？
  - 按照一定的**规则**对**多项式**进行变换
  - 项的**结构类型**不同，**适用规则**不同
  - 有哪些规则？
- 输入输出操作
  - 要特别注意符号和不定长的整数
- 多项式的存储管理
  - 不知道会有多少项