

复旦大学计算机科学技术学院



# 编程方法与技术

## 3.1. 上次课复习

周扬帆

2021-2022第一学期

# 关于JavaScript的var

## □ var定义的变量

- 链式作用域 – chain scope
- 函数一层层往外找，直到全局

```
var value = 'local';
var func = function() {
  if (false) {
    var value = 'func_local';
  }
  console.log(value);
}
func();
console.log(value);
```

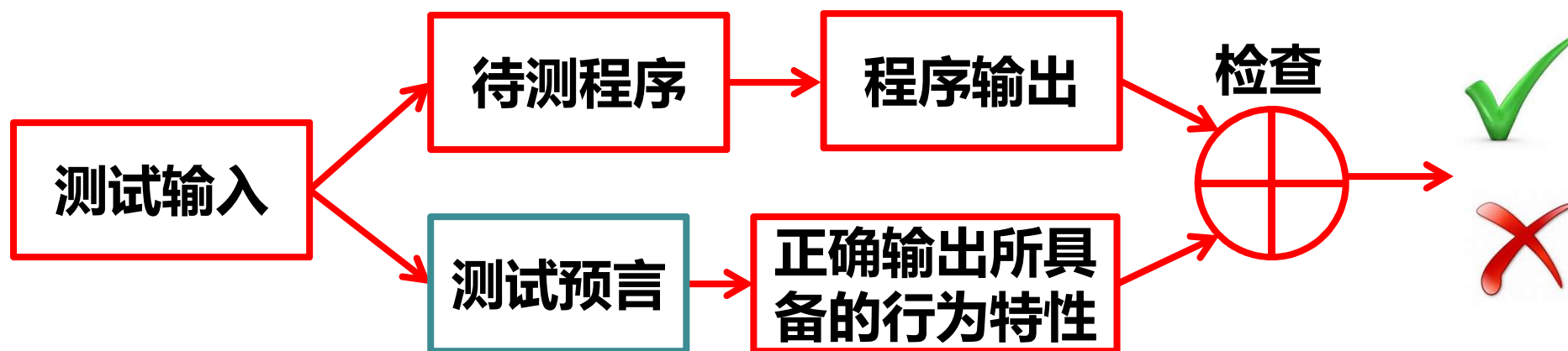
# 测试

## □ 怎么知道程序是对的

测试是一种常见的方法

- ◆ 给程序一组输入 (test inputs)
- ◆ 程序产生一组输出 (test results)
- ◆ 判断程序行为是否符合预期

功能性测试



# 排序程序的测试

1. **随机产生**的一个序列  $\text{nums}[0 \rightarrow n-1]$

2. 排序

3. **检查结果是否满足特性**

for each  $i = 0 \rightarrow n-2$

check if  $\text{num}[i] \leq \text{num}[i+1]$

# 代码风格

## □ 写出易于理解的代码

- 命名、排版、注释
- **先写注释，后写代码**

## □ Javadoc

- 通过注释自动生成文档

## □ 目的

- 协助别人理解
- 协助自己理解（遗忘）
- 因为大部分工业代码都需要维护（**升级、除错**）

# 方法（函数）

## □ C语言的函数定义

```
int add(int a, int b) {  
    return a + b;  
}
```

## □ Java语言的方法定义

- 类似

- 作用域 + 类型 + 返回值 + 函数名 (参数序列)

```
public static void main (String[] args)
```

## □ JavaScript语言的函数定义

```
function add(a, b) {  
    return a + b;  
}
```

# 引入方法（函数）的目的

## □ 提高代码可读性

- 尤其对于复杂的程序过程(procedure)
- 帮助分块理解

## □ 实现代码的可复用性

- 简洁、省内存空间
- 便于实现库 (library)

## □ 实现模块化设计

- 方便测试
- 方便修改
- 不同人/团队做不同的事情，松耦合

复旦大学计算机科学技术学院



# 编程方法与技术

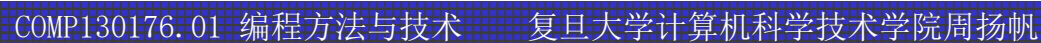
## 3.2. 面向对象入门

周扬帆

2021-2022第一学期



## ❑ 程序就是一个过程procedure



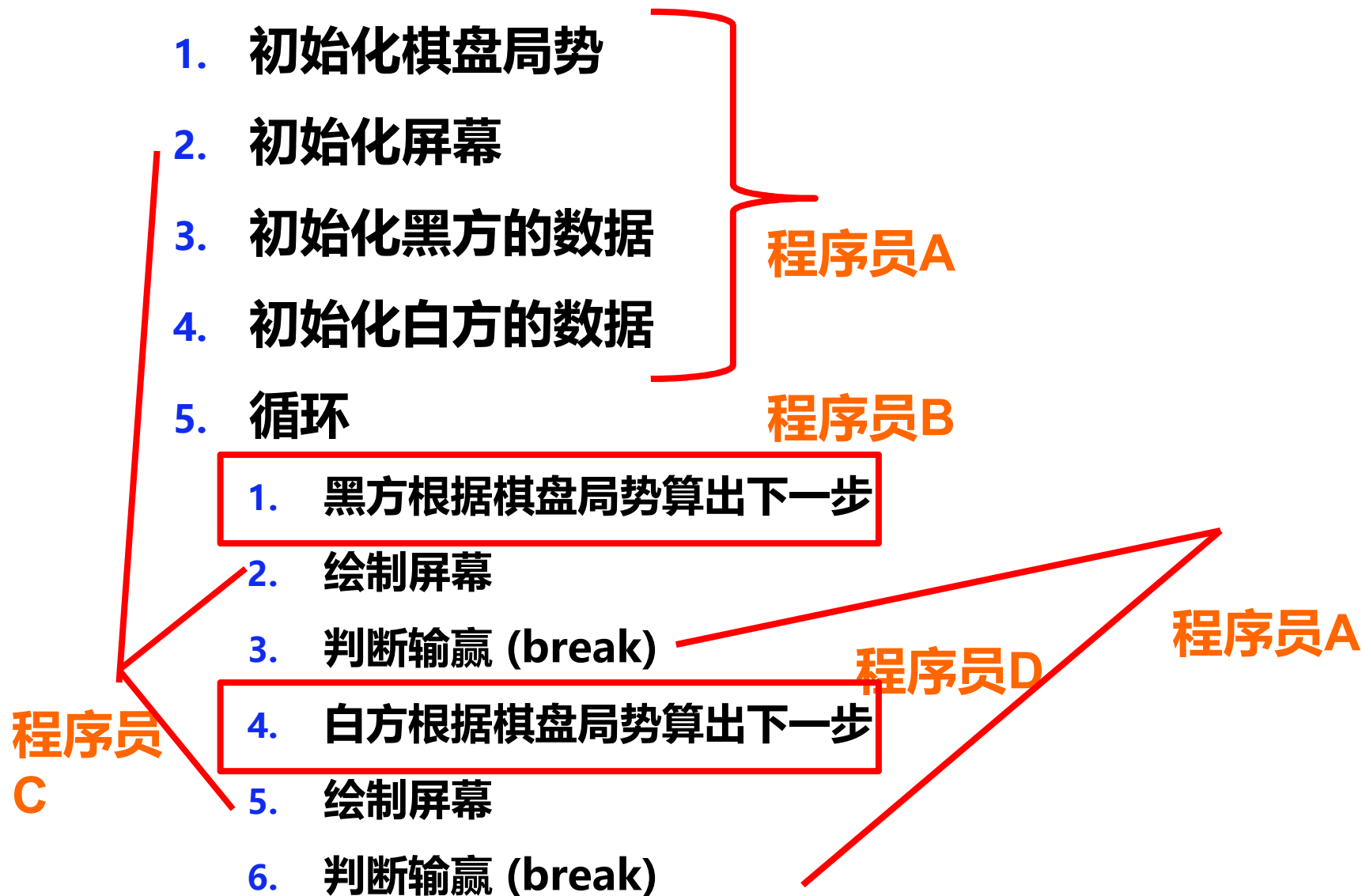
# 方法（函数）：面向过程的视角

- **程序就是一个过程procedure**
- **结构化程序**
  - 顺序执行
  - 选择分支执行
  - 循环执行
- **结构化程序可实现任何算法**
- **方法（函数）将过程变成子过程**

# 对弈：面向过程的做法

1. 初始化棋盘局势
2. 初始化屏幕
3. 初始化黑方的数据
4. 初始化白方的数据
5. 循环
  1. 黑方根据棋盘局势算出下一步
  2. 绘制屏幕
  3. 判断输赢 (break)
  4. 白方根据棋盘局势算出下一步
  5. 绘制屏幕
  6. 判断输赢 (break)

# 模块化的对弈



# 模块化的对弈

1. 初始化棋盘局势
2. 初始化屏幕
3. 初始化黑方的数据
4. 初始化白方的数据
5. 循环

程序员A

棋盘变了!!

可是我们写死了棋盘大小了

程序员B

1. 黑方根据棋盘局势算出下一步

2. 绘制屏幕

3. 判断输赢 (break)

程序员D

程序员A

4. 白方根据棋盘局势算出下一步

5. 绘制屏幕

6. 判断输赢 (break)

程序员C

# 模块化的对弈

1. 初始化棋盘局势
2. 初始化屏幕
3. 初始化黑方的数据
4. 初始化白方的数据
5. 循环

程序员A

程序员B

1. 黑方根据棋盘局势算出下一步

2. 绘制屏幕

3. 判断输赢 (break)

4. 白方根据棋盘局势算出下一步

5. 绘制屏幕

6. 判断输赢 (break)

输赢规则变了!

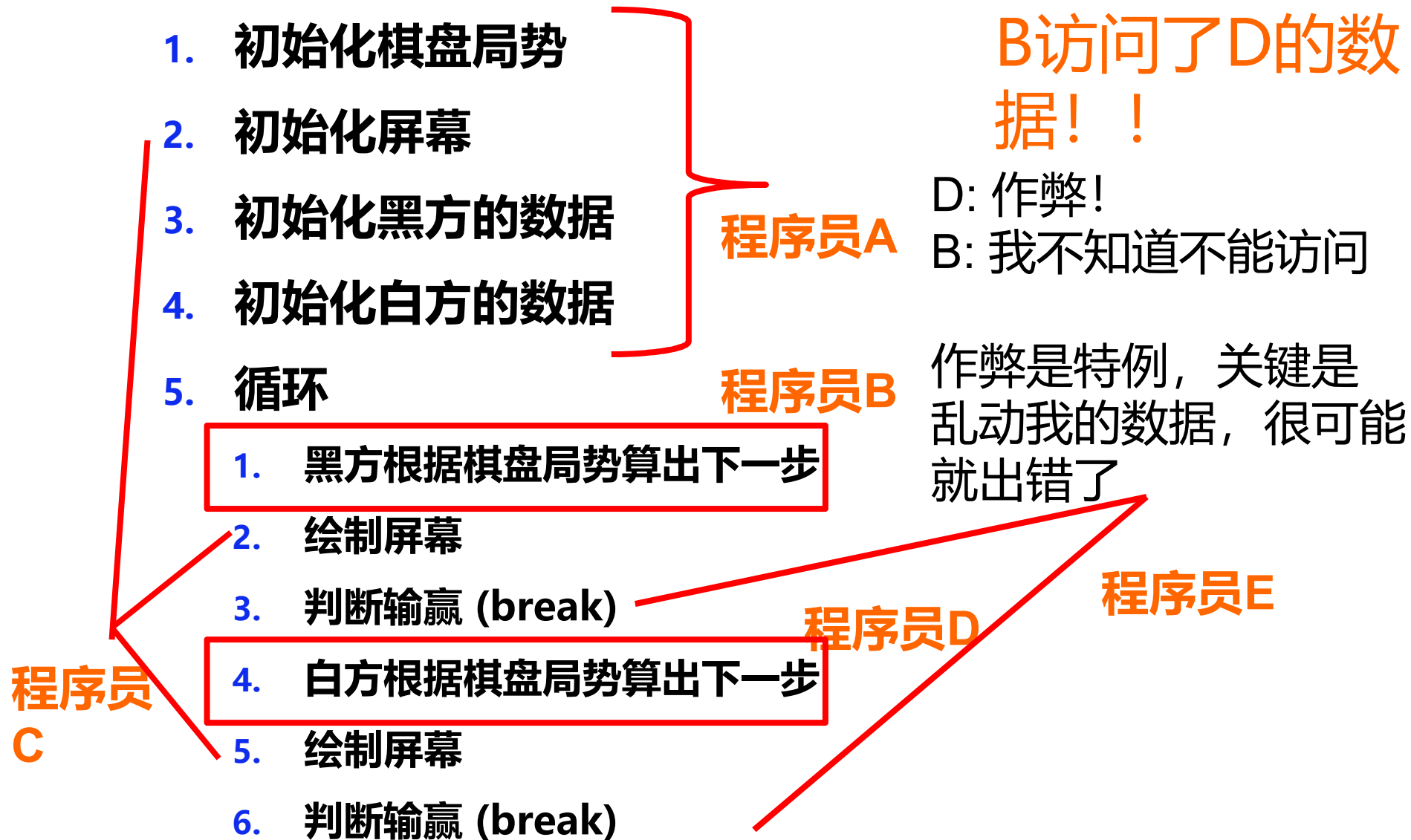
可是我们按规则写死程序了

程序员C

程序员D

程序员E

# 模块化的对弈

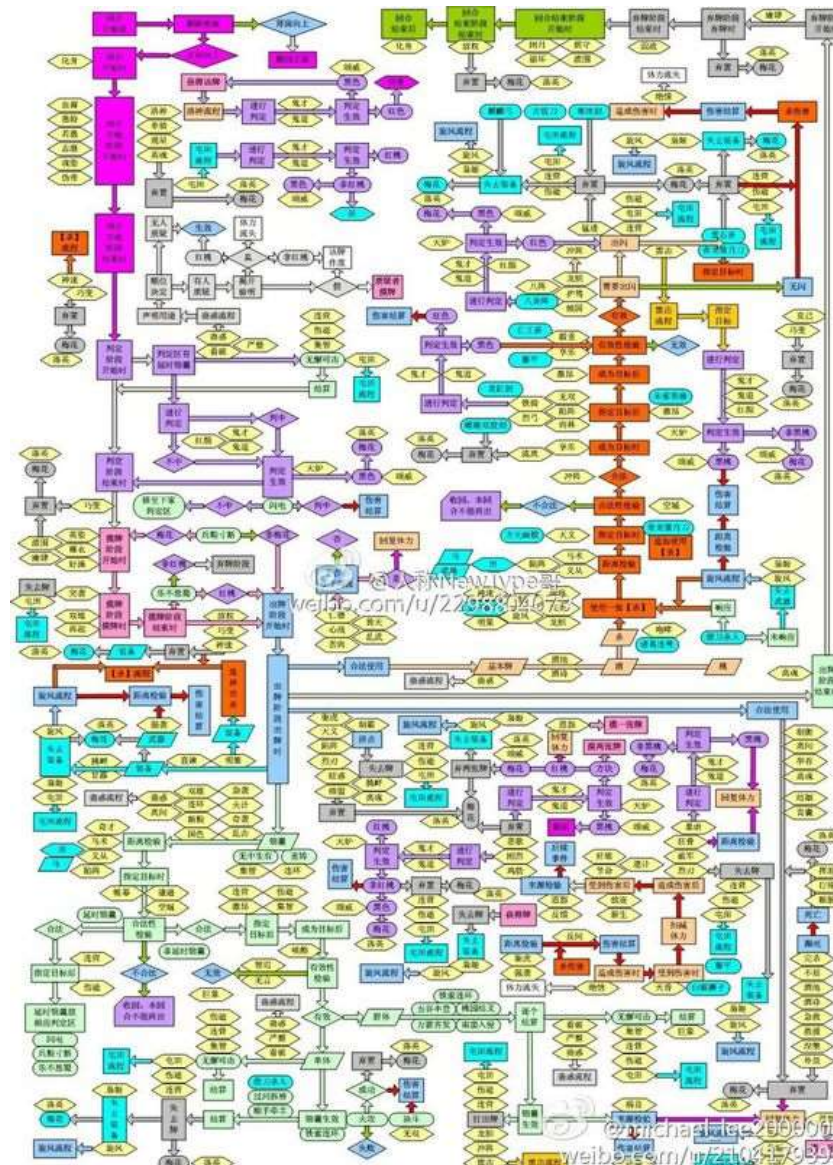


# 对弈：面向过程的做法

- ◆ 各种初始化
- ◆ 玩家循环
  - ◆ 玩家n摸牌
  - ◆ 玩家n根据局势和牌算出下一步
  - ◆ 根据规则决定
    - ◆ 有玩家胡牌吗？ (break)
      - ◆ 按规则决定顺序，调用玩家的决定函数
    - ◆ 有玩家碰吗？
      - ◆ 按规则决定顺序，调用玩家的决定函数
    - ◆ 下家吃吗？
      - ◆ . . .



# 对弈：面向过程的做法



# 面向对象

## □ 根据数据(对象)实现模块化

- 数据的操作**封装**在数据对象内部

- 子过程（方法）**针对数据各司其职**

- 可读性更好

- 改了数据，要改子过程更方便（不难找）

- 控制对象的数据访问权限

- 私有数据不能在对象外访问 → 更鲁棒

## □ 程序：对象的相互调用过程

## □ 程序设计：确定、定义要处理的数据，以此为程序设计的起点

# 简例：学生信息(排序)

- **Java语言定义一个Student数据结构**

```
class Student
{
    char name[] = new char[10];    //名字
    double gpa;                    //GPA
    int studentID;                  //学号
};
```

**看做类似于C的struct定义**

# 简例：学生信息(排序)

- 初始化包含n个Student元素的数组

```
Student students[] = new Student[n];
```

```
for(int i = 0; i < n; i++)
```

```
{
```

```
    students[i] = new Student();
```

```
    students[i].gpa = ra.nextDouble() * 4.0;
```

```
    students[i].studentID = n - i;
```

```
    String name = "student" + i;
```

```
    char [] charName = name.toCharArray();
```

```
    for(int j = 0; j < charName.length; j ++)
```

```
    {
```

```
        students[i].name[j] = charName[j];
```

```
    }
```

```
}
```

细节暂时不要管  
dirty也不要管

# 简例：学生信息(排序)

- **printStudents: 在屏幕显示记录**

- **实现**

- 按照GPA从小到大的排序函数

- ```
static void sortGPA(Student students[])
```

- 按照SID从小到大的排序函数

- ```
static void sortSID(Student students[])
```

提示:

1. 数组大小为students.length

2. 访问数据例子 `int id = students[i].studentID;`

# 排序的讨论

- 如果程序很大，怎么**分工**
  - 谁来实现sortGPA, sortSID
  - 谁来实现Student的定义
  - 谁来实现初始化
- 所有涉及student记录的操作由一个人（团队）负责
  - 改起来出错的概率小
  - 自己写的代码更容易复用

# Class是一种数据结构+

## □ 排序: 定义一个StudentRecord数据结构

```
class StudentRecord
{
    Student students[]; //存记录
    void initStudents() //初始化
    { ... }
    void sortSID () //根据SID排序
    { ... }
    void sortGPA () //根据GPA排序
    { ... }
    void printStudents () //显示记录
    { ... }
};
```

**方法针对数据各司其职**

# 对象的初始化和方法调用

## □ 排序: 程序流程

```
public static void main (String[] args) {  
    StudentRecord studentRecord = new StudentRecord();  
    studentRecord.initStudents();  
    studentRecord.printStudents();  
  
    System.out.println("GPA Sorting Result:");  
    studentRecord.sortGPA(); //根据gpa排序  
    studentRecord.printStudents();  
  
    System.out.println("SID Sorting Result:");  
    studentRecord.sortSID(); ; //根据sid排序  
    studentRecord.printStudents();  
}
```



# 对象的初始化和方法调用

## □ 排序的例子

```
public static void main (String[] args)
{
```

```
    StudentRecord studentRecord = new StudentRecord();
    studentRecord.initStudents();
    studentRecord.printStudents();
```

...

```
    StudentRecord studentRecord = new StudentRecord();
```

变量声明

对象初始化

studentRecord是类StudentRecord的对象

new 分配存储空间

```
    studentRecord.initStudents();
```

调用对象的initStudents方法

# 排序的讨论

## □ sortSID和sortGPA

```
void sortSID ()
{
    for(int i = 0; i < students.length; i ++)
    {
        for(int j = 0; j < students.length - i - 1; j ++)
        {
            if (students[j].studentID > students[j+1].studentID)
            {
                Student temp = students[j+1];
                students[j+1] = students[j];
                students[j] = temp;
            }
        }
    }
}

void sortGPA ()
{
    for(int i = 0; i < students.length; i ++)
    {
        for(int j = 0; j < students.length - i - 1; j ++)
        {
            if (students[j].gpa > students[j+1].gpa)
            {
                Student temp = students[j+1];
                students[j+1] = students[j];
                students[j] = temp;
            }
        }
    }
}
```

# 代码复用

## □ sortSID和sortGPA: 可复用

```
void sort (int type)
{
    for(int i = 0; i < students.length; i ++)
    {
        for(int j = 0; j < students.length - i - 1; j ++)
        {
            boolean flag = false;
            switch(type)
            {
                case 1:
                    flag = students[j].compareGPA(students[j+1]);
                    break;
                case 2:
                    flag = students[j].compareSID(students[j+1]);
                    break;
            }
            if (flag)
            {
                Student temp = students[j+1];
                students[j+1] = students[j];
                students[j] = temp;
            }
        }
    }
}
```

# 各司其职

## □ 比较，放到Student里

```
class Student
{
    char name[] = new char[10];
    double gpa;
    int studentID;
    void setName(char [] studentName)
    {
        for(int j = 0; j < name.length && j < studentName.length; j ++){
            //只取前10个
            name[j] = studentName [j];
        }
    }
    boolean compareSID(Student s2)
    {
        return studentID > s2.studentID;
    }
    boolean compareGPA(Student s2)
    {
        return gpa > s2.gpa;
    }
};
```

方法针对数据各司其职

# 数据的保护

## ■ 如果姓名需要输入

→ 输入大于10怎么办?

```
class Student
{
    char name[] = new char[10];    //名字
    double gpa;                    //GPA
    int studentID;                  //学号
};
```

```
Student student = new Student();
for(int j = 0; j < studentName.length; j++)
{
    student.name[j] = studentName [j];
}
```

# 数据的保护

## ■ 如果姓名需要输入

→ 输入大于10怎么办?

```
class Student
```

```
{
```

```
    char name[] = new char[10];    //名字
```

```
    void setName(char [] studentName)
```

```
    {
```

```
        for(int j = 0; j < name.length &&  
                j < studentName.length; j ++)
```

```
        { //最多只取前10个
```

```
            name[j] = studentName [j];
```

```
        }
```

```
    }
```

```
};
```

# 数据的保护

- ❑ 通过setName修改name可以保护数据
- ❑ 不够！别人不遵守用setName的约定

```
Student student = new Student();  
student.setName(name); //name是char[]数组
```

```
//也可以这么来  
student.name = name;
```

```
//保护就没用了
```

# 数据的保护

- 把数据分成四类

- public private protected default(不指定)

- 四类数据可访问范围不一样

- 本次课关注public和private

- 其他两类以后讲

- 例子

- public int studentID;

- private char[] name;



# 数据的保护

## ■ public数据

→ 在其他对象中都可直接访问该数据

→ 用对象名.变量名 访问

```
Student student = new Student();  
student.studentID = 1;
```

## ■ private数据

→ 其他对象不可访问该数据

→ 用对象名.变量名 访问 （报错）

# 数据的保护

- ❑ 将name定义为私有数据
- ❑ 通过setName修改name

```
Student student = new Student();  
student.setName(name); //name是char[]数组
```

```
//不可以这么来  
student.name = name;
```



```
//保护就有用了
```

# 数据的保护

## ■ 数据的保护

→ 强制让一些数据的修改只能通过方法调用实现

→ 目的?

- 方法调用可以进行sanity check

- 甚至可以方便记录日志

- 可控、可查

# 数据的保护

- **对象的方法可以访问对象所有的数据**
  - 太强大
  - 太危险
  - 某些方法调用可能只是子过程
    - 别的对象**没必要用**
    - 别的对象也**容易用错**

# 数据的保护

## ■ 某些对象的方法参数复杂

```
void sort (int type)
{
    for(int i = 0; i < students.length; i ++)
    {
        for(int j = 0; j < students.length - i - 1; j ++)
        {
            boolean flag = false;
            switch(type)
            {
                case 1:
                    flag = students[j].compareGPA(students[j+1]);
                    break;
                case 2:
                    flag = students[j].compareSID(students[j+1]);
                    break;
            }
            if (flag)
            {
                Student temp = students[j+1];
                students[j+1] = students[j];
                students[j] = temp;
            }
        }
    }
}
```

别的对象容易搞错  
参数  
type = ? 是什么意思

# 数据的保护

## ■ 某些对象的方法只在特定情况才执行

```
Student[] students[];  
void initStudents()  
{  
    Student s2 = new Student();  
    String name2 = "student";  
    char [] charName2 = name2.toCharArray();  
    s2.setName(charName2);  
    System.out.println(s2.name[0]);  
    Random ra = new Random();  
    int n = 5;  
    students = new Student[n];  
    for(int i = 0; i < n; i++)  
    {  
        students[i] = new Student();  
        students[i].gpa = ra.nextDouble() * 4.0;  
        students[i].studentID = n - i;  
        String name = "student" + i;  
        char [] charName = name.toCharArray();  
        for(int j = 0; j < charName.length; j++)  
        {  
            students[i].name[j] = charName[j];  
        }  
    }  
}
```

别的对象错误执行,  
将导致对象数据被  
篡改。

# 数据的保护

## ■ public方法

→ 在其他对象中可直接调用

→ 用对象名.方法名 调用

```
studentRecord.sortGPA ();
```

## ■ private方法

→ 其他对象不可调用该方法

→ 用对象名.方法名 调用 （报错）

# 对象的初始化

## ■ 对象的数据大部分情况需初始化

→ 设定初始值

→ 初始化存储空间

```
Student[] students[];  
void initStudents()  
{  
    Student s2 = new Student();  
    String name2 = "student";  
    char [] charName2 = name2.toCharArray();  
    s2.setName(charName2);  
    System.out.println(s2.name[0]);  
    Random ra = new Random();  
    int n = 5;  
    students = new Student[n];  
    for(int i = 0; i < n; i++)  
    {  
        students[i] = new Student();  
        students[i].gpa = ra.nextDouble() * 4.0;  
        students[i].studentID = n - i;  
        String name = "student" + i;  
        char [] charName = name.toCharArray();  
        for(int j = 0; j < charName.length; j ++)  
        {  
            students[i].name[j] = charName[j];  
        }  
    }  
}
```



# 对象的初始化

- 对象的数据大部分情况需初始化

- 设定初始值

- 初始化存储空间

- 万一别人用的时候忘了怎么办？

```
public static void main (String[] args)
{
    StudentRecord studentRecord = new StudentRecord();
    studentRecord.initStudents();
    ...
    studentRecord.sortGPA(); //根据gpa排序
}
```

# 对象的初始化

## ■ 对象的数据大部分情况需初始化

- 设计一种方法，该方法一定会在对象被创建时被调用
- 叫构造方法

```
class A
```

```
{
```

```
    A(参数表)
```

```
    { // 实现
```

```
    }
```

```
}
```

和类名同名，无返回值

# 对象的初始化

## ■ 构造方法

→ 和类名同名，无返回值

→ 可以有参数

```
class A
{
    A(int i)
    { //实现
    }
}
```

对象初始化 `A a = new A(10);`

# 对象的高级TOPIC

- **数据存在哪里**
- **方法的代码存在哪里**
- **对象的引用reference存在哪里**

# 思考

- ❑ 思考面向对象是否一定优于面向过程, 各自有什么适合场景
- ❑ 编写面向对象程序解决问题时, 思考如何分类, 如何划分类里的方法
- ❑ C++与Java的方法和类的限定符异同

复旦大学计算机科学技术学院



# 编程方法与技术

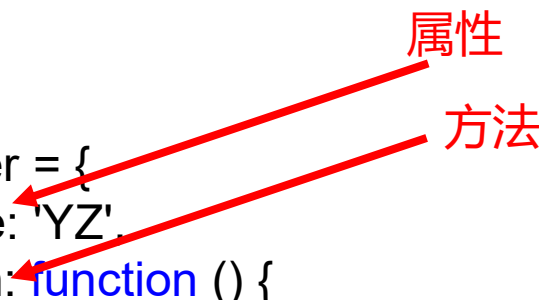
## 3.3. JavaScript对象

周扬帆

2021-2022第一学期

# JavaScript对象

- 对象由属性(properties)和方法(methods)两个基本元素构成



```
var lecturer = {  
  name: 'YZ',  
  teach: function () {  
    alert(this.name + ' teaches nothing useful');  
  }  
}  
  
lecturer.teach();
```

# JavaScript对象

## ■ 对象的属性和方法可以动态增删

```
var lecturer = {  
  name: 'YZ',  
  teach: function () {  
    alert(this.name + ' teaches nothing useful');  
  }  
}
```

```
lecturer.teach();  
lecturer.title = 'Dr. ';  
lecturer.quit = function () {  
  alert(this.title + this.name + ' quits.');
```

```
delete lecturer.quit;  
lecturer.quit();
```

Define = Use

等式左边define了一个属性\方法

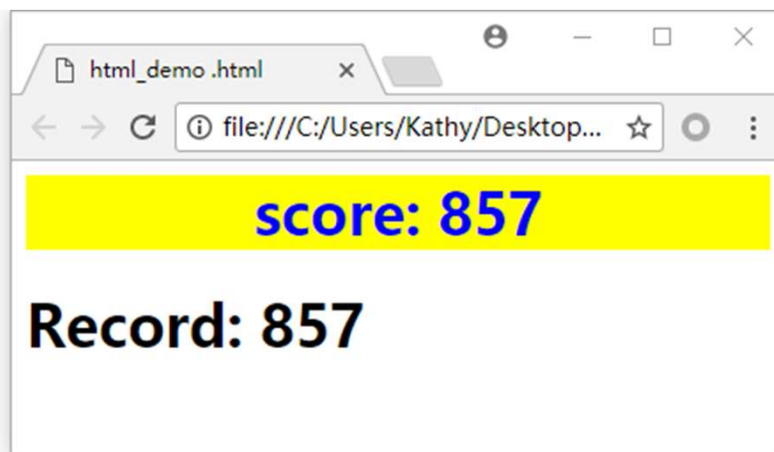
等式右边use了一个值



# 回顾课堂练习得到时间的代码

## □ 写一个小游戏网页

- 计算双击之间的时间差
- 必须小于1000ms
- 越接近1000ms分数越高



- 得到当前离盘古开天地的时间差（毫秒）

- myDate=new Date();
- Number(myDate.getTime());

new Date()发生了什么?

# JS的对象的创建

- **new func(...)**就是以func为构造函数，构造了一个对象，并返回

- 例子

```
function Lecturer (name) {  
    alert(name + ' teaches nothing useful');  
}  
var YZ = new Lecturer ('YZ');
```

YZ现在是一个没有自定义属性和方法的对象

```
alert(YZ.name);  
YZ.name = 'YZ';  
alert(YZ.name);
```

# JS的对象的创建

```
function Lecturer () {  
  this.getName = function () {  
define    return this.name;  
          use  
  }  
  this.setName = function (name) {  
define    this.name = name;  
          define  
  }  
}  
var Y = new Lecturer ();  
Y.setName('Y');  
alert(Y.getName());
```

- 函数的this指针，在new的时候绑定到新创建的数据空间

# JavaScript对象

- 对象由属性(properties)和方法(methods)两个基本元素构成
- 可动态增删
- 没有访问控制
- 任何函数可以作为构造函数，当然编程的时候要设计好构造函数，专门用于构造对象

```
function Lecturer () {  
    this.getName = function () {  
        return this.name;  
    }  
    this.setName = function (name) {  
        this.name = name;  
    }  
}
```

v.s.

```
function Lecturer () {  
    alert('...');  
}
```

复旦大学计算机科学技术学院



# 编程方法与技术

## 3.4. JavaScript闭包

周扬帆

2021-2022第一学期

# 面向对象的数据封装

## □ Java数据封装

- 面向对象设计：面向数据，组织代码
- 封装数据和数据的处理方法为类

```
public class Student {  
    private String name;  
    public String getName() {  
        return name;  
    }  
}
```

## □ JavaScript?

# 面向对象的数据封装

## □ JavaScript数据封装

- 面向对象设计：面向数据，组织代码
- 考虑一个计数器例子
  - 数据：计数
  - 方法：自增

可能解法1

```
var count = 0;  
var counter = function () {  
    ++count;  
    return count;  
}
```

```
console.log(counter());  
console.log(counter());
```

全局变量，没有访问控制

# 全局变量有什么不好

- ❑ 全局变量：不进行作用域封装
- ❑ 不能进行访问控制
  - 大家都能改，容易有bug
  - 统一修改函数 / 便于调试，log
- ❑ 不便于模块化设计
  - 代码复杂、不容易理解
- ❑ 全局变量太多，占用空间
- ❑ 全局变量太多，影响效率
  - 出现一个变量，要进行遍历查询



# 面向对象的数据封装

## □ JavaScript数据封装

- 面向对象设计：面向数据，组织代码

- 考虑一个计数器例子

- 数据：计数

- 方法：自增

可能解法1

```
var count = 0;
var counter = function () {
  ++count;
  return count;
}
```

```
console.log(counter());
console.log(counter());
```

全局变量，没有访问控制

Object() → JS基本对象的构造方法  
counter → 一个对象

可能解法2

```
var counter = new Object();
counter.count = 0;
counter.inc = function () {
  this.count ++;
  return this.count;
}
```

```
console.log(counter.inc());
console.log(counter.inc());
```

没有访问控制

# 数据的访问控制

## 可能解法3

```
var counter = function () {  
    var count = 0;  
    ++count;  
    return count;  
}
```

```
console.log(counter());  
console.log(counter());
```

## 可能解法4

```
function counter() {  
    var count = 0;  
    var innerCounter = function () {  
        ++count;  
        return count;  
    }  
    return innerCounter();  
}
```

```
console.log(counter());  
console.log(counter());
```

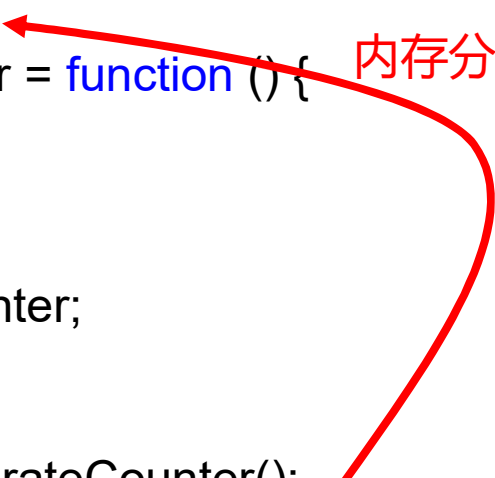


关键在于每次都调用counter, 会清零counter作用域的count数据

# 面向对象的数据封装

可能解法5

```
function generateCounter() {  
  var count = 0;  
  var innerCounter = function () {  
    ++count;  
    return count;  
  }  
  return innerCounter;  
}  
  
var counter = generateCounter();  
console.log(counter());  
console.log(counter());
```



内存分配

# 闭包

## □ 闭包

- 内部函数及其定义时的上下文
- 作用：外部作用域访问内部作用域中变量
  - 实现面向对象的数据封装

```
function generateCounter() {  
  var count = 0;  
  var innerCounter = function () {  
    ++count;  
    return count;  
  }  
  return innerCounter;  
}
```

```
var counter = generateCounter();  
console.log(counter());  
console.log(counter());
```

# 闭包

## □ 闭包

- 内部函数及其定义时的上下文
- 作用：外部作用域访问内部作用域中变量
  - 实现面向对象的数据封装

会不会相互影响？

```
function generateCounter() {  
    var count = 0;  
    var innerCounter = function () {  
        ++count;  
        return count;  
    }  
    return innerCounter;  
}  
  
var counter1 = generateCounter();  
console.log(counter1());  
var counter2 = generateCounter();  
console.log(counter2());
```

# 闭包

## □ 内存管理

```
function generateCounter() {  
  var count = 0;  
  var innerCounter = function () {  
    ++count;  
    return count;  
  }  
  return innerCounter;  
}  
var counter = generateCounter();  
console.log(counter());
```

内存不释放，直到没有变量存有innerCounter

内部函数chain scope的变量都不会释放

注意避免内存浪费/泄露

# 闭包用途

## □ 实现私有成员 – 进行面向对象数据封装

```
function generateCounter() {  
  var count = 0;  
  var innerCounter = function () {  
    ++count;  
    return count;  
  }  
  return innerCounter;  
}  
var counter = generateCounter();  
console.log(counter());
```

```
var Student = function () {  
  var name = 'default';  
  return {  
    getName: function () {  
      return name;  
    },  
    setName: function (newName) {  
      name = newName;  
    }  
  };  
};
```

```
var student = Student();  
console.log(student.getName());
```

封装  
getter/setter



复旦大学计算机科学技术学院



# 编程方法与技术

## 3.5. Java程序的组织

周扬帆

2021-2022第一学期



# Java程序

- 设想一个程序(项目)由很多开发者开发
- 之前我们说过的分工?
  - 模块化, 面向类(数据和数据的操作)分工
- 程序由许多类(classes)组成
- 问题: 能不能最后放在一个.java文件里?



LinkedListTest.java

```
class Element()  
{ ... }  
class LinkedList()  
{ ... }  
  
public class LinkedListTest {  
    public static void main(String argv[]) { ... }  
}
```

# Java程序源文件

## □ 放在一个文件的坏处

- 不方便模块化分工
- 不方便编译
- 不方便动态加载
- 不方便版本控制

...

## □ Java程序一般由多个源文件构成

# 多文件项目

## □ 问题

- 源文件怎么组织
- 源代码文件中怎么指定需要用到的别的源代码文件的类

# Java项目的源代码组织

## □ 最直接的做法：一个类对应一个源文件

- 好处1：方便程序员管理
- 好处2：方便JVM加载
- 坏处：琐碎、麻烦

后面会详细讲

## □ Java的折中

- 一个源文件可以包含多个类
- 但是每个文件只能包含一个public类

# Java项目的源代码组织

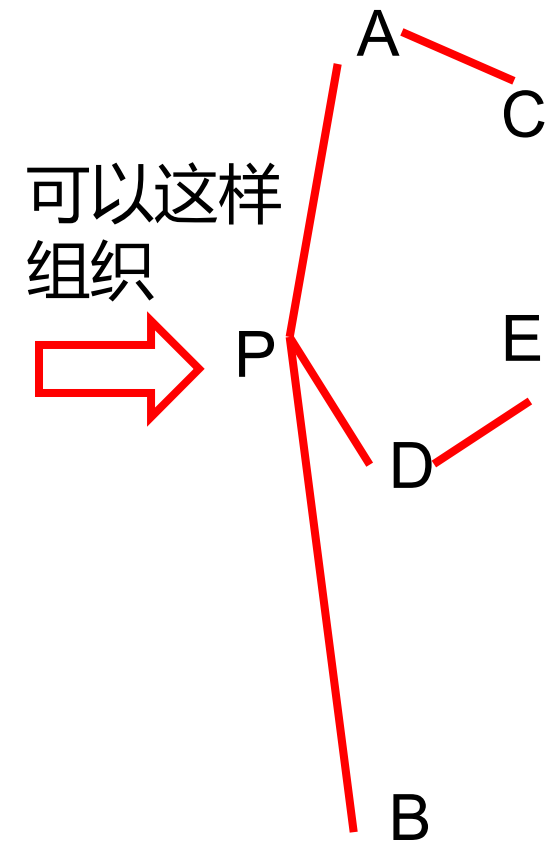
## □ 多个源文件如何组织其层次结构

- 扁平化处理？放在同一个文件夹中
- 缺点：太乱、不方便模块化管理等
- 根据逻辑关系，按模块分目录，树状管理

```
src/.../module1/module1.1/filename1.java
                        | filename2.java
                        | ...
                        | module1.2/filename3.java
                        | filename4.java
                        | ...
                        | ...
module2/filename5.java
...
...
```

方便理解、方便分工 ...

# 面向对象



# 命名空间

- **不同团队需要协调类的名字**
  - 不能重名，否则JVM出错
- **保证不重名太麻烦，甚至不可行**
- **需引入命名空间**
  - 在一个命名空间内不重名
  - 不同命名空间，可以重名
  - 方法：Java包(package)

# Java包(package)

- 包是一组类（源文件）的集合
- 一般一个功能模块放在一个包里
  - 如日志处理
  - 如网络功能
- 指定包名的语句
  - `package pkg1[.pkg2[.pkg3...]]`
  - 如 `package abc.de.fg`
  - 源文件的第一条语句



# Java包 (package)

- 指定了包名的源文件需**放在特定目录**
  - 如用package abc.de.fg指定了包名的文件
  - 放在 abc/de/fg/ 目录下
  - 方便JVM查找
- 不指定包名的文件，属于“无名包”
  - 只能放在源文件根目录下
- Java类库 (JDK) 都会指定在JDK定义的包中
  - 如之前用过的Random类：  
java.util.Random

# 多文件项目

## □ 问题

- 源文件怎么组织
- 源代码文件中怎么指定需要用到的别的源代码文件的类

# 类的访问

- 类似于类里变量和方法有访问控制
  - private public
- 源文件里的类也需要有访问权限
  - 有些数据结构（类）无须其他团队的访问/操作
  - 避免误用
- Java提供两种类别的访问权限
  - public: 整个程序都可以访问
  - default: 只有包内部可以访问

# 单向链表例子实现

LinkedListTest.java

```
class Element()
```

```
{
```

```
...
```

```
}
```

```
class LinkedList()
```

```
{
```

```
...
```

```
}
```

```
public class LinkedListTest
```

```
{
```

```
    public static void main(String argv[]) {
```

```
        Element e = new Element();
```

```
        LinkedList list = new LinkedList();
```

```
        e.setNum(1);
```

```
        list.add(e);
```

```
        ...
```

```
    }
```

```
}
```

# public类

- 一个源文件最多只有一个public类 (?)
- public类是可以被别人使用的
  - JVM是动态加载类的 (Week 1)
  - 运行时JVM需快速定位到实现某个类的文件
    - bytecodes
  - 最快的方法：文件名和类名一样，根据名字来定位
    - 找Student类，只需要找名为Student的bytecodes文件(Student.class)
- 因此一个文件只能有一个public类

# 使用别的源文件的类

## □ 使用包内部的类

- 直接用，就像定义在本文件里一样

## □ 使用其他包的public类

- 必须指定包名
- 原因：方便JVM查找，容许不同包的类重名

## □ 假如包p2的类B需要使用包p1里定义了类A

- `p1.A a = new p1.A();`
- 如果包名很长，书写非常繁琐

# 使用别的源文件的类

## □ 假如包p2的类B需要使用包p1里定义了类A

- `p1.A a = new p1.A();`

- 第二种方法: **import语句**

`import` pkg1[.pkg2[.pkg3...]].类名;

```
import p1.A; //置于package语句后
...
A a = new A();
```

`import` pkg1[.pkg2[.pkg3...]].\*; (使用所有的类)

```
import p1.*; //置于package语句后
...
A a = new A();
```

# 思考

- 了解Java的包机制，与各种你熟悉的语言如C++/C#/Python/Ruby的命名空间机制进行比较，找出其异同点
- 比较C++与Java之间的类访问权限的异同点
- JVM这种规定一个源文件只有一个public类的做法有无缺点



# 课堂练习1：数组

## ■ 实现一个IntArray类

- 管理一个存放整数的数据结构
- 数组大小的最大值在初始化时构造函数指定
- 数组的可供别的对象调用的方法

### ■ 数组一开始无元素

### ■ add(int n)将n顺序加入数组中，成功返回true

- 数组元素数目++，但不能超过初始化的最大数目

### ■ isExist(int n)如果数组里有n返回true

### ■ getnthNumber(int k), 得到第k大的数

# 课堂练习2：单向链表

## □ 实现单向链表LinkedList

- 链表元素命名为Element
  - 包含整形变量n
- 实现**delete(Element e)**: 删除链表里面的元素ele, 当且仅当ele包含的整数值和e包含的整数值一样, 没得删返回false (删除所有符合条件的ele)
- 实现**isExist(Element e)**: 查找链表是否存在元素ele, 而ele包含的整数值和e包含的整数值一样, 没有返回false
- 实现**add(Element e)**: 在链表的末端中添加元素ele, 使ele包含的整数值和e包含的整数值一样

链表： 在一个类中可大胆使用自己这个类来定义变量

# 单向链表例子实现

LinkedListTest.java

```
class Element
```

```
{
```

```
    private int n;
```

```
    private Element next;
```

内部数据保护

```
    public int getNum() ...
```

```
    public Element getNext() ...
```

```
    public void setNum(int num) ...
```

```
    public void setNext(Element e) ...
```

通过方法访问被保护的数据

```
}
```

```
class LinkedList
```

```
{
```

```
    private Element first;
```

内部数据保护

```
    public boolean isExist(Element e)
```

```
    ...
```

```
    public boolean delete(Element e)
```

通过方法访问被保护的数据

```
    ...
```

```
    public void add(Element e) ...
```

```
}
```

# 单向链表例子实现

LinkedListTest.java

**class** Element

{

...

}

**class** LinkedList

{

...

}

**public class** LinkedListTest

{

**public static void** main(String argv[]) {

        Element e = **new** Element();

        LinkedList list = **new** LinkedList();

        e.setNum(1);

        list.add(e);

        ...

    }

}