

注: do loop 与 while loop 的区别

```
while (条件) {  
    循环主体  
}
```

while 先判断后执行

```
do  
{ 循环主体
```

```
} while (条件);
```

do 先执行一次 (至少), 再判断 \rightarrow 判断表达式.

布尔表达式规则

(注意逻辑是否等同)

$!(A \ \&\& \ B) == !A \ || \ !B$

$!(A \ || \ B) == !A \ \&\& \ !B$

$(\text{score} \geq 0 \ \&\& \ \text{score} \leq 100) == (A \ \&\& \ B)$

$(\text{score} < 0 \ || \ \text{score} > 100) == !(A \ \&\& \ B)$

二者本质相反

for each 结构.

```
for (type var: iterator_obj)  
<loop body>;
```

=

```
for (数据类型 数据变量 x; 遍历对象 obj)  
<用了 x 的 Java statement>;
```

```
int[] a = {1, 2, 3};
```

```
for (int x: a)
```

```
    system.out.println(i + ",");
```

$\Rightarrow 1, 2, 3$.

Switch Statement

最输出的是匹配成功的初值

\downarrow
`switch (int-expr)`

First to match decides where

execution within switch body BEGINS.

/ execution proceeds from π to End of block.

如果想要 each case 相互排斥,
让匹配成功就 break, 使其跳出.

```

char grade = 'C';

switch(grade)
{
    case 'A':
        System.out.println("优秀");
        break;
    case 'B':
    case 'C':
        System.out.println("良好");
        break;
    case 'D':
        System.out.println("及格");
        break;
    case 'F':
        System.out.println("你需要再努力努力");
        break;
    default:
        System.out.println("未知等级");
}
System.out.println("你的等级是 " + grade);

```

Switch 实例

Output: 良好

你的等级是 C

↓
初值被 case 后匹配

当程序过长,容易出错 => break .. into small segments

✧ Method (subprogram)

方法(定义&调用)

方法提供了 functional abstraction

— 我们要知道 $\left\{ \begin{array}{l} \text{arguments} \\ \text{effort of the method} \end{array} \right.$

```

// Make an array
int [] A = new int[100];
// Put some data into it
// We will see this soon
// Print out the data
for (int i = 0; i < A.length; i++)
    System.out.println(A[i]);
// Sort the data. We don't need to
// know how this works!
Arrays.sort(A);
// Print it out again (sorted)
for (int i = 0; i < A.length; i++)
    System.out.println(A[i]);

```

// In Class Arrays

```

public static void sort(int [] A)
{
    // Code to sort the data
    // This will execute but the
    // caller does not need to
    // know how it works. The
    // details are abstracted
    // out of the caller's view
}

```

Java的方法主要有两个用途:

- ① act as function (函数), 将结果返回给调用代码
— 这些方法用返回类型声明, 在表达式中被调用

例: $X = \text{inScan.nextDouble}();$
 $Y = (\text{Math.sqrt}(X))/2;$

- ② act as a subroutine (子例程) 或 procedure (过程); 代码执行但不返回结果.
— 这些方法被声明为 void, 被作为分离的独立语句调用

例: Arrays.sort(myData);

Java有很多预定义方法 (online API).

ClassName . methodName (param-list)

↙ ClassName — the class in which the method is defined

methodName — the name of method

param-list — 传递给该method的0或多个 variables or expressions 的列表

例 $Y = \text{Math.sqrt}(X);$

Math 类

方法名

variable 列表

(static methods or class methods.
与 class 相关联, 而不是对象)

另. ClassName . ObjectName . methodName (param-list)

ObjectName — 包含该方法的静态预定义的对象名称.

例. System.out.println("Hi");

预定义类

系统中预定义PrintStream
的对象

方法名

实例方法 [instance methods] ↗ 与对象相关联

Lecture 6

如果我们想使用没有被预定义的方法? — 那就自己写

句法样式: `public static void methodName(param_list)`
{ // method body -- procedure }

`public static retval methodName(param_list)`
{ // method body - function
 return <something>;
}

对于②: — retval is some java type
— 当 method X void, there
MUST be a return statement

例: [方法、方法调用及输出]

```
sayWacky();  
sayWacky();  
for (int i = 0; i < 4; i++)  
  sayWacky();
```

```
public static void sayWacky()  
{  
  System.out.println("Wacky");  
}
```

Output:
Wacky
Wacky
Wacky
Wacky
Wacky
Wacky

调用3次
其中一次作为 Loop Body

param_list

[参数列表] ⇒ 这关乎如何把 values 传递给 method
⇒ 使方法能在不同点处理不同信息
(有列表就有不同的 value)

— In the method definition

type identifier 对的列表, 由逗号分隔
被称为 parameter [形参] 虚拟

— In the method call 调用

与定义中形参 — 对应的变量或表达式的列表
被称为 argument [实参]

当方法被调用时, 实参被求值,
然后传递给形参

对于: 方法内的形参随即在需要时被使用
方法完成后, 形参消失 (与方法中其它部分一起)

如果在定义方法的同一个类中
调用了方法, 调用时无需
使用类名.

```
public static double area(double radius)  
{  
  double ans = Math.PI * radius * radius;  
  return ans;  
}
```

(无 void, 要 return)

```
... // code where method is called  
double rad = 2.0;  
double theArea = area(rad);
```

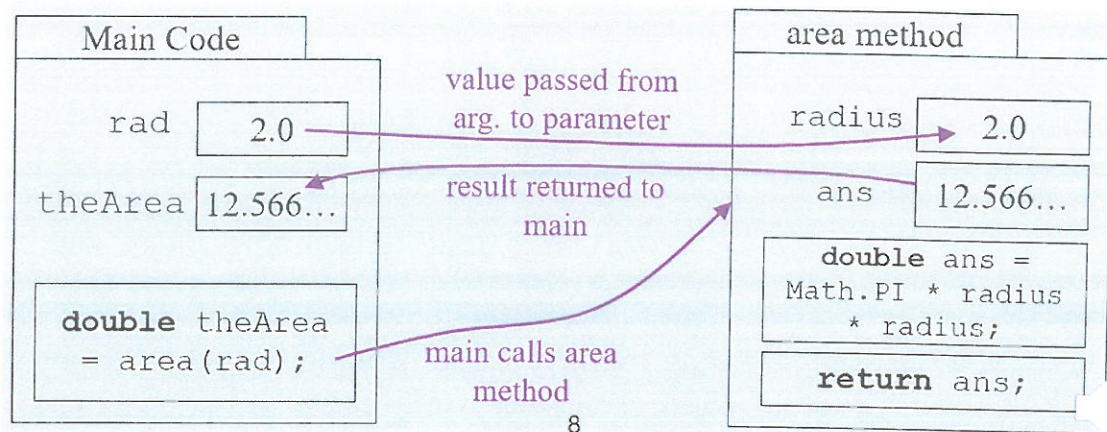
形参
parameter
实参
argument

形参由 value 传递.

一 形参是实参求值的副本

一 任何对形参的改变都不影响实参

在方法中输入实参为 2
先给定义方法中的形参
定义方法计算出 ans
将 ans 作为返回值
主方法调用定义方法
得到返回值.



Effect of value parameters

传递给定义方法的实参在方法内不能改变 (防止方法带来意外副作用)

如果我们想要改变实参? Swap 方法.

- Ex: swap(A, B)

> Method to swap the values in A and B

> Let A = 10 and B = 20 and we call swap(A, B);

public static void swap(int X, int Y)

```
{  
    int temp = X;  
    X = Y;  
    Y = temp;  
}
```

A 10 → X 20

B 20 → Y 20

方法完成, 参数消失后, A 和 B 都没变.

在一个方法内声明的变量是局部变量 — 通常也被称为 method variable

一 它们只存在于方法的上下文中

一 这同样包含形参

[把形参视为在方法调用时初始化的局部变量

方法结束时, 变量消失]

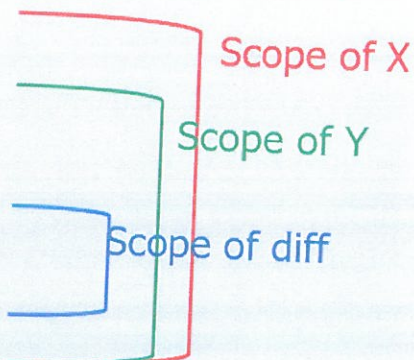
Scope of variables 变量的作用域 — 在一个方法里定义的变量

只在该方法中有效.

另: Java 变量同样可以在方法内部的块中被声明.

此时, 变量的作用域是所在块的范围

```
public void foo(int X)
{
    int Y = 0;
    while (Y < X)
    {
        int diff = X - Y;
        // output diff
        Y++;
    }
}
```



注意: local variable 局部变量是不能跨越方法分享的

→ 一个方法中声明的局部变量不被另一个方法接受
[即使它们名字相同, 仍是不同变量]

但我们仍可以跨方法获得数据

— 为了在方法间 share variables, 我们使用面向对象编程.



Review: Java has primitive type & reference type
基本类型 引用类型

直接存储于与变量相关的内存位置.

var1 100

值是对 object 的引用



Reference ?

存储在变量中的数据只是 "address" of location where the object is stored

— 数据/分离于对象本身

例: 我电话里有你的家庭地址, 我可以给你寄东西 & 拜访你;

但我电话中没有你的房子, 也不能改变电话中文件来帮你办的房子.

这就是存储 address 与存储 actual object 的区别

↓ 我无法通过 address 直接到你的房子

直接改变无意义

但有了 address, 我可以间接接触改变 object 的 data.

例: 知道 address, 我能去你家拍张照片 → get some information

知道 address, 我能去你家偷辆车 → change it in some way

Object

- Classes are blueprint for data

封装

[Class structure shows a good way to ENCAPSULATE the data and operation.
类结构将数据与操作封装.

如何表示 string? string 由什么构成? — DATA

可以对 string 做什么? 如何调用? — OPERATION

Class 定义了一个 type 的数据和操作是什么.

封装了一个 type 的属性

↓ Object is an actual value of type ^{一个类的实际值}

Object is instance of class (对象是类的实例化)

因此 Java 的类决定了对象的结构与行为

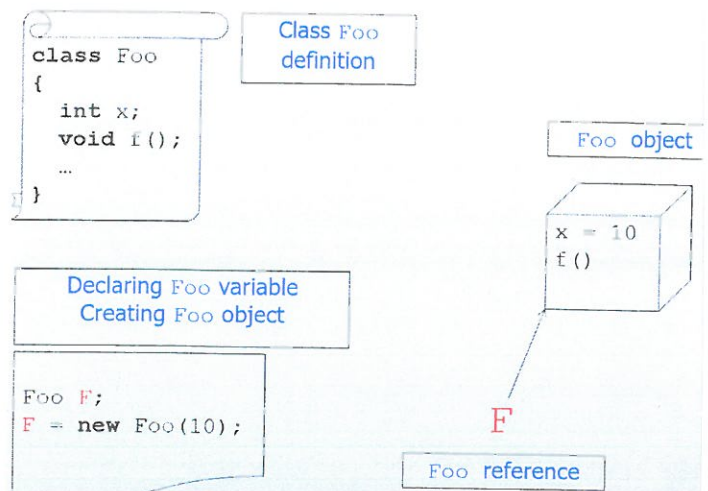
简例: 给一个 class 叫 Foo

其中我们封装了 int variable x 和 void method f()

现在可以制造 an instance of a Foo

It is an object
满足 have int x & method f()

可以制造很多个, 相互分离



Lecture 7

implications of reference vars

① 声明一个 var 并不创建一个 object

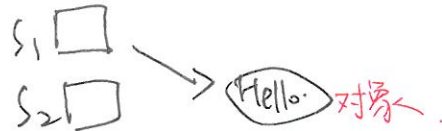
StringBuilder S₁, S₂;

S₁ = new StringBuilder("Hello");

→ 实际操作

S₁ references an instance of StringBuilder object while S₂ = X.

→ 'have no actual StringBuilder objects, just 2 vars
'use the new operator or call a method that will create an object



此时 S₂ 无 value, access it 会导致错误

② 多个变量可以访问和更改同一对象: S₂ = S₁; // copy reference from S₁ to S₂

③ Properties of objects are accessed via "dot" notation

(public methods & instance variables)

S₁.append("Friends!");

- 这种中访问对象的方式是 reference

S₁ is a reference (address)

S₁.<xx> =

① go to the object whose address is stored in S₁
② access/call the specified variable or method

由于 S₂ 是 S₁ 的 copy, 同样改变

④

引用变量的比较比的是引用, 而不是对象

→ address

```
StringBuilder S3 =
```

```
    new StringBuilder("Hello Friends!");
```

```
if (S1 == S2) System.out.println("Equal"); // yes
```

```
if (S1 == S3) System.out.println("Equal"); // no
```

== compares those address values
They X have the same location.

如果我们想要比较 objects (data)?

use equals() method

↓ equals 方法

— 用于比较对象内的 data

— define it for our own classes

— StringBuilder 无自己的 equals, convert to string

> 比较 individual characters

return true if same & false otherwise

```
if (S1.toString().equals(S2.toString()))  
    System.out.println("Same Value");  
    // yes.
```

另: compareTo() 方法用于 inequality

差异: — == 表示这是完全相同的 object (因为只有一个对象位于该位置)

— equals 表示值在某种程度上相同 (取决于如何定义)

Reference 可以被设置为 null 来 (re) 初始化变量

此时无法用 "dot" access

S1 = null;

S1.append(); → 导致 run-time error

方法调用与被访问的对象相关联, 而不是变量.

[如果没有对象, 就没有方法可调用]

面向对象编程的 3 个基本原则

① 封装 & 数据抽象化

↓ 对数据的操作被看作数据类型的一部分

↓ 使用数据类型而不需知道实现细节

— 只要知道 interface (or method headers)

— 与方法的 functional abstraction 比较

② Inheritance 继承

— 数据类型的属性可以传递给子类型

— 可以构建具有多个继承级别的 class 层次结构

③ Polymorphism 多态

- 与变量一起使用的操作基于被访问对象的类, 而不是变量的类
 - 父类与子类的对象可以用同一方式访问
- (多态指的是同一接口, 由于不同 instance 而导致不同 operation)

Consider primitive type

每个变量代表单一、简单的数据
任何实施于数据的操作基于类数据外部



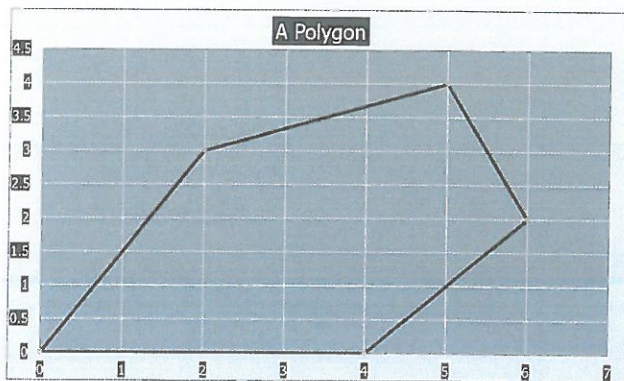
Consider data

data 很复杂

例: 多边形 - 多个点相连

int [] xpnts x 坐标数组
int [] ypnts y 坐标数组
int npoints 多边形点数量

它们共同组成多边形



As a single Polygon object

Consider operations

考虑一个多边形能做什么的操作

- Polygon 能做什么; 而不是能对 Polygon 做什么

- Object 是主动的而不是被动的 (例)

- void addPoint(int x, int y) // 加入一个新点, add a new point to Polygon
- boolean contains(double x, double y) // is point (x,y) within the boundaries of the Polygon 点在形内
- void translate(int deltaX, int deltaY) // move all points in the Polygon by deltaX and deltaY

通过 deltaX & deltaY 移动所有形内点

在编程中: 操作与数据相分离

如果我有一个多边形 P, 加个点我得 addPoint(P, 0, 2); P 作为形参传给 method addPoint
| addPoint 作用于 P

我们拥有的 P 就是多边形 P 和一个作用于 P 的方法 addPoint.