



| Java技术

第三章 Java面向对象

路 强

luqiang@hfut.edu.cn

合肥工业大学计算机与信息学院

目 录



1

面向对象概念

2

面向对象的三大特性

3











抽象类与接口













4

“设计模式”初体验 – 简单工厂模式

计算机语言发展史



<div>  <h2>PROGRAMMING LANGUAGES</h2> </div>				
1957-1959	FORTRAN (FORMULA TRANSLATION), LISP (LIST PROCESSOR), AND COBOL (COMMON BUSINESS-ORIENTED LANGUAGE) Considered the oldest languages that are still used today. High-level languages created for scientific, mathematical, and business computing.	PRIMARY USES Supercomputing applications, AI development, business software	USED BY NASA, credit cards, ATMs 	FUN FACT Action movie The Terminator used samples of Cobol source code for the text shown in the Terminator's vision display.
1970	PASCAL (AFTER FRENCH MATHEMATICIAN/PHYSICIST BLAISE PASCAL) High-level. For teaching structured programming and data structuring. Commercial versions widely used throughout the '80s.	CREATOR NIKLAUS WIRTH 	PRIMARY USES Teaching programming USED BY Apple Lisa (1983), Skype 	
1972	C (BASED ON AN EARLIER LANGUAGE CALLED "B") General-purpose, low-level. Created for Unix systems. Second most popular language (behind Java). Many leading languages are derivatives, including C#, Java, JavaScript, Perl, PHP, and Python.	CREATOR DENNIS RITCHIE 	PRIMARY USES Cross-platform programming, system programming, Unix programming, computer game development USED BY Unix (rewritten in C in 1973), early WWW servers and clients 	
1983	C++ (FORMERLY "C WITH CLASSES"; ++ IS THE INCREMENT OPERATOR IN "C") Intermediate-level, object-oriented. An extension of C, with enhancements such as classes, virtual functions, and templates.	CREATOR Bjarne STROUSTRUP 	PRIMARY USES Commercial application development, embedded software, server/client applications, video games USED BY Adobe, Google Chrome, Mozilla Firefox, Microsoft Internet Explorer 	
1983	OBJECTIVE-C (OBJECT-ORIENTED EXTENSION OF "C") General-purpose, high-level. Expanded on C, adding message-passing functionality based on Smalltalk language.	CREATOR BRAD COX AND TOM LOVE 	PRIMARY USES Apple programming USED BY Apple's OS X and iOS operating systems 	

1987	PERL ("PEARL" WAS ALREADY TAKEN) General-purpose, high-level. Created for report processing on Unix systems. Today it's known for high power and versatility.	CREATOR LARRY WALL 	PRIMARY USES CGI, database applications, system administration, network programming, graphics programming USED BY IMDb, Amazon, Ticketmaster 
1991	PYTHON (FOR BRITISH COMEDY TROUPE MONTY PYTHON) General-purpose, high-level. Created to support a variety of programming styles and be fun to use. Tutorials, sample code, and instructions often contain Monty Python references.	CREATOR GUIDO VAN ROSSUM 	PRIMARY USES Web applications, software development, information security USED BY Google, Yahoo, Spotify 
1993	RUBY (THE BIRTHSTONE OF ONE OF THE CREATOR'S COLLABORATORS) General-purpose, high-level. A teaching language influenced by Perl, Ada, Lisp, Smalltalk, etc. Designed for productive and enjoyable programming.	CREATOR YUKIHIRO MATSUMOTO 	PRIMARY USES Web application development, Ruby on Rails USED BY Twitter, Hulu, Groupon 
1995	JAVA (FOR THE AMOUNT OF COFFEE CONSUMED WHILE DEVELOPING THE LANGUAGE) General-purpose, high-level. Made for an interactive TV project. Cross-platform functionality. Currently the world's most popular programming language.	CREATOR JAMES GOSLING 	PRIMARY USES Network programming, web application development, software development, Graphical User Interface development USED BY Android OS/apps 
1995	PHP (FORMERLY "PERSONAL HOME PAGE," NOW IT STANDS FOR "HYPertext PREPROCESSOR") Open-source, general-purpose. For building dynamic web pages. Most widely used open-source software by enterprises.	CREATOR RASMUS LERDORF 	PRIMARY USES Building/maintaining dynamic web pages, server-side development USED BY Facebook, Wikipedia, Digg, WordPress, Joomla 
1995	JAVASCRIPT (FINAL CHOICE AFTER "MOCHA" AND "LIVESCRIPT") High-level. Created to extend web page functionality. Used by dynamic web pages for form submission/validation, interactivity, animations, user activity tracking, etc.	CREATOR BRENDAN EICH 	PRIMARY USES Dynamic web development, PDF documents, web browsers, desktop widgets USED BY Gmail, Adobe Photoshop, Mozilla Firefox 

<https://zhuanlan.zhihu.com/p/57698890>

第一代和第二代早期程序拓扑结构

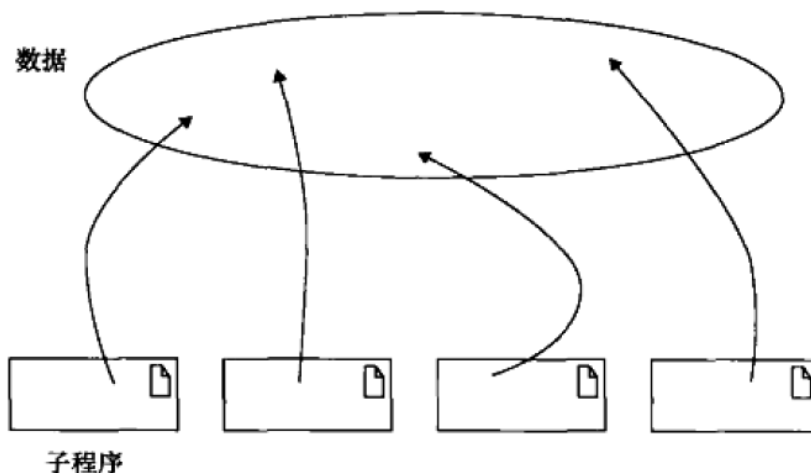


图 2-1 第一代和第二代早期程序设计语言的结构

- 程序基本构成单元是子程序。扁平的物理结构，只包含全局数据和子程序。图中箭头表示子程序对不同数据依赖关系。
- 尽管设计时，设计者可以在逻辑上将不同类型的数据分开，但语言中没有强制机制。
- 全局数据结构对所有子程序都可见，程序中某部分的错误可能给系统其他部分带来毁灭性的连带影响。

第二代后和第三代早期程序拓扑结构

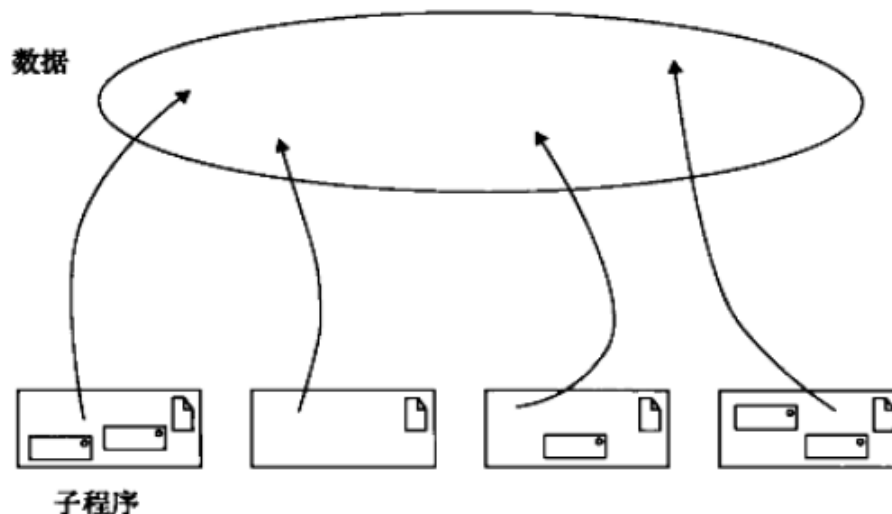


图 2-2 第二代后期和第三代早期程序设计语言的结构

- 20世纪60年代中期，将子程序作为一种抽象机制，即现在所称的“过程化”抽象。带来三个结果：
 - 发明了一些语言支持各种参数传递机制
 - 奠定结构化程序设计基础，语言上支持嵌套子程序，并在控制结构和声明的可见性范围方面发展了一些理论
 - 出现了结构化程序设计方法。为试图构建大型系统的设计提供了指导，利用子程序作为基本构建块。

第三代后期程序拓扑结构

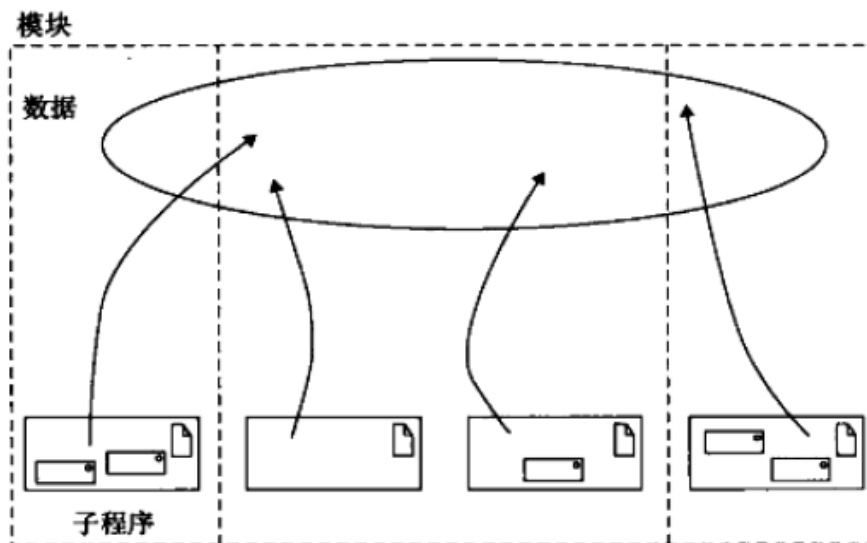


图 2-3 第三代后期程序设计语言的结构

- 大规模编程项目意味着大型的开发团队，因此需要独立地开发同一个程序的不同部分。解决方案是能够独立编译的模块
- 这一代语言虽然支持某些模块化结构，但很少有规则要求模块间的语义一致性。（如现代语言中“函数参数列表”）
- 这些语言中的大部分对数据抽象和强类型支持都不太好，很多错误只有在执行程序时才能检查出来。

面向对象程序拓扑结构

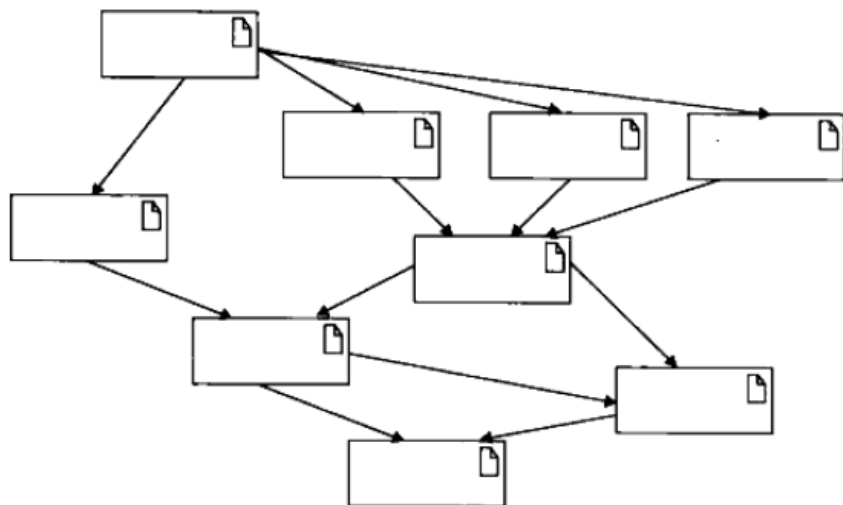


图 2-4 使用基于对象或面向对象编程语言的小型或中型应用的结构

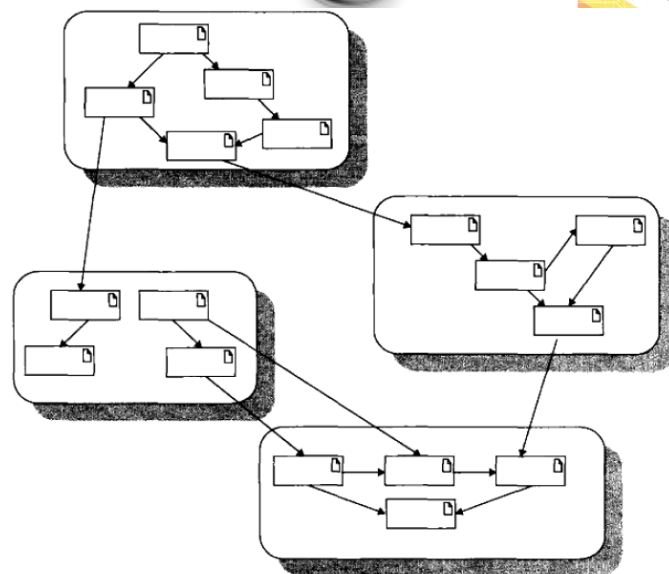


图 2-5 使用基于对象或面向对象编程语言的大型应用的结构

- 通过**过程**的抽象本质上适合描述**抽象操作**，不适合描述**抽象的对象**
- 这类语言的构件块是模块，逻辑上表现为**一组类或对象**，而非早期的**子程序**
- 小型或中型OO程序物理结构表现为一个图，而面向算法语言那样通常是一棵树。数据和操作放在一个单元中，系统的基本逻辑构件块不再是算法，而是类或对象。

OOA、OOD 和OOP



○ 面向对象分析（OOA）

- 面向对象分析是一种分析方法，利用从问题域的词汇表中找到的类和对象来分析需求。

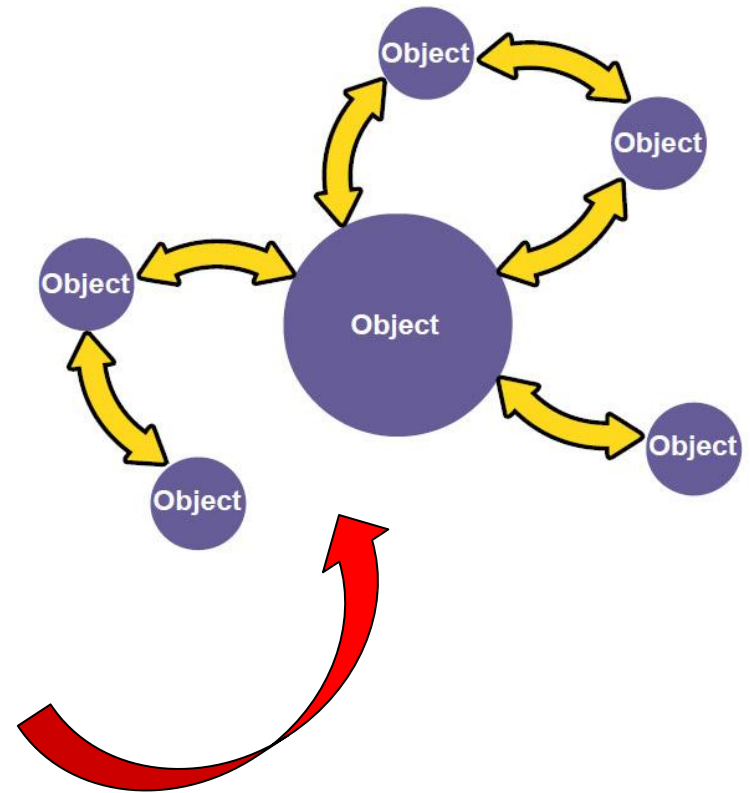
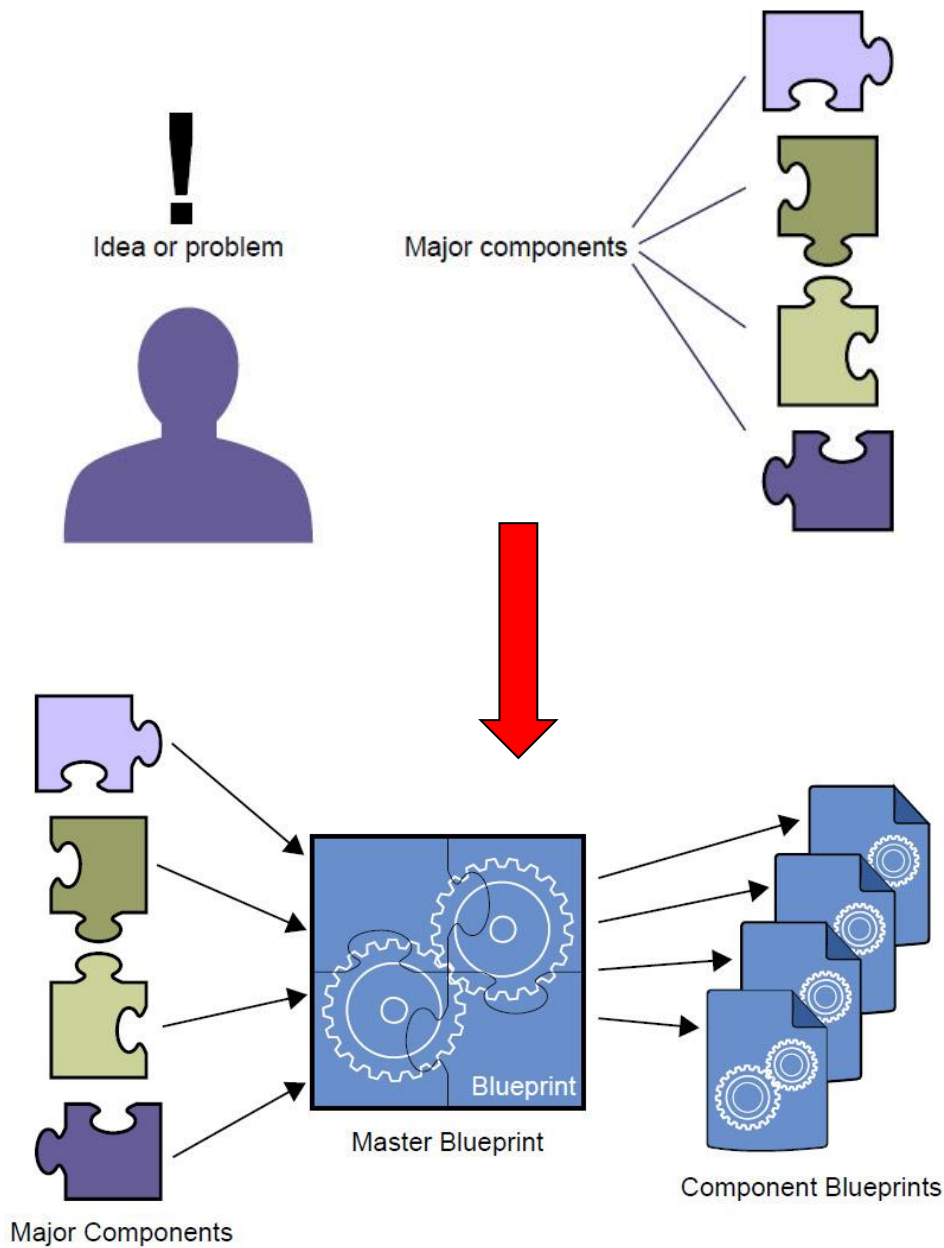
○ 面向对象设计（OOD）

- 面向对象设计是一种设计方法，包括面向对象分解的过程和一种表示法，用于展现被设计系统的逻辑模型和物理模型、静态模型和动态模型。

○ 面向对象编程（OOP）

- 面向对象编程是一种实现的方法，程序被组织成许多组相互协作的对象，每个对象代表某个类的一个实例，而类则属于一个通过继承关系形成的层次结构。

OOAD



目 录



1

面向对象概念

2

面向对象的三大特性

3

抽象类与接口

4

“设计模式”初体验 – 简单工厂模式

面向对象的三大特性



- 封装 (Encapsulation)

- 继承 (Inheritance)

- 多态 (Polymorphism)

 - 重载、覆盖

抽象 - 类的概念



- 从柏拉图时代开始，无数的哲学家、语言学家、认知科学家和数学家都考虑过**分类的问题**。
- 从历史来看，存在三种一般的分类方式：
 - 经典分类
 - 概念聚集
 - 原型理论
- 类是对象的“**模板**”，对象是类的实例。类与对象之间的关系可以看成是抽象与具体的关系。其中**类是抽象的**，**对象是具体的**
 - 类描述了对对象的属性和行为
 - 对象是类的一个实例

面向对象的基本特征



○ 封装性：

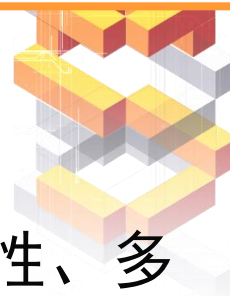
基本思想：

把客观世界中联系紧密的元素及其相关操作组织在一起，使其相互作用隐藏、封装在内部，而对外部对象只提供单一的功能接口

目的： 将对象的使用者和设计者分开

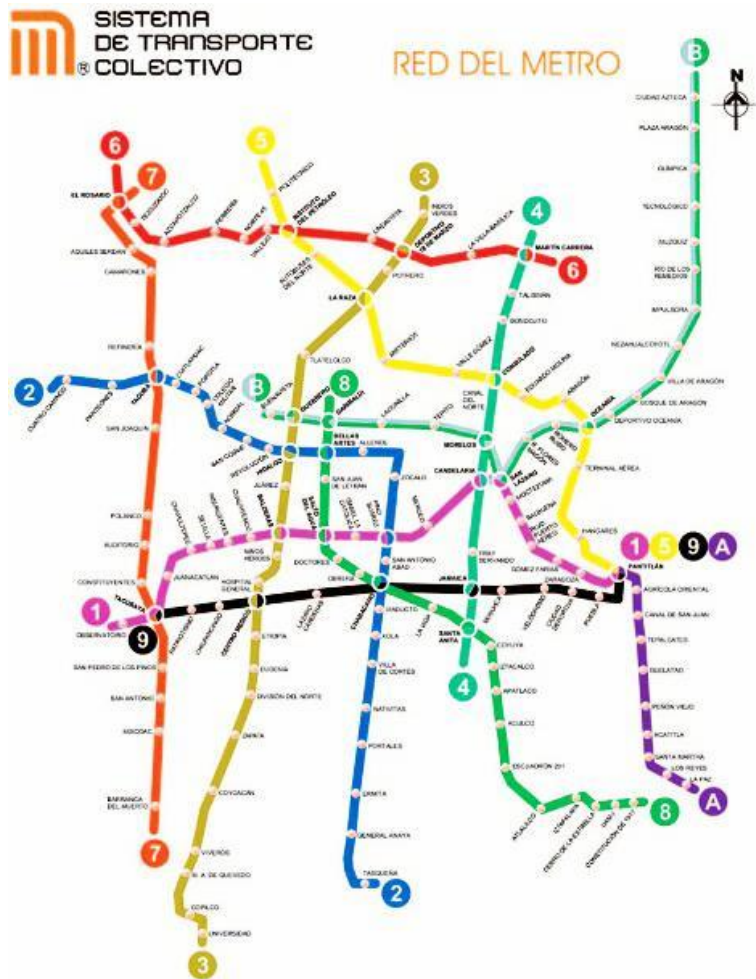
类比： 汽车的左转、右转（方向盘 与 车轮）；
被陶瓷封装的集成电路

封装

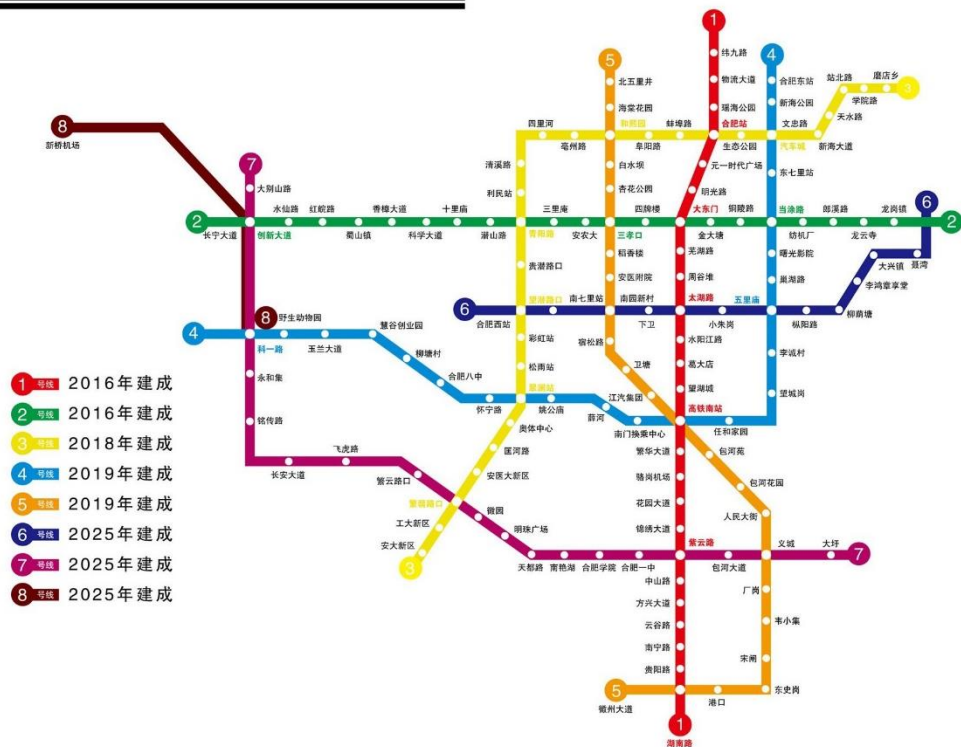


- 面向对象的程序设计 以类作为基本处理单元，对象是类的实例。面向对象程序设计的重要特征是具有封装性、多态性和继承性。
- 所谓**封装**表现在以下几个方面
 1. 在类的定义中设置对对象中的成员变量和方法进行访问的权限。
 2. 提供一个统一供其它类引用的方法。
 3. 其它对象不能直接修改本对象所拥有的属性和方法。
- 对象及成员变量的访问权限

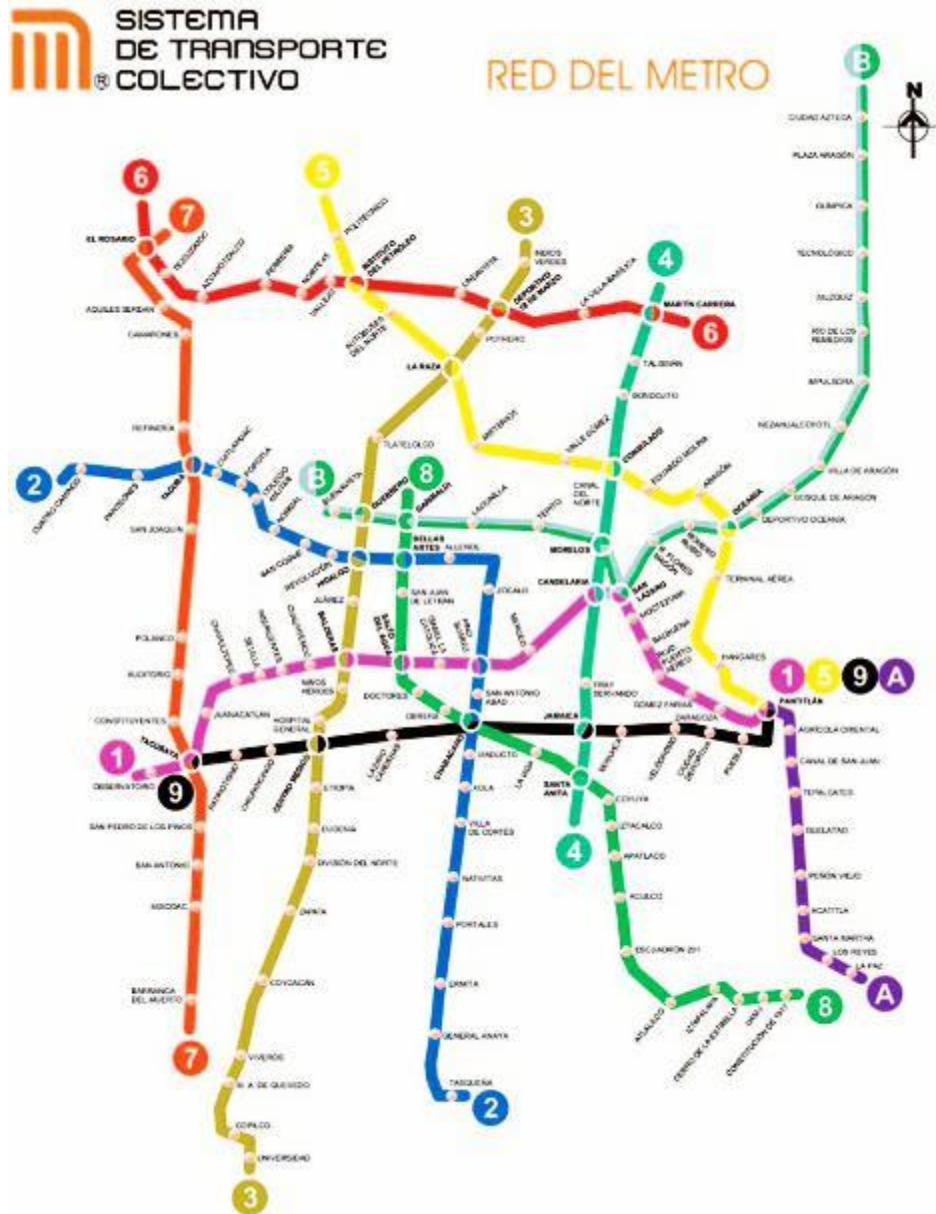
这样的设计...



合肥地铁站点信息图



这样的设计...



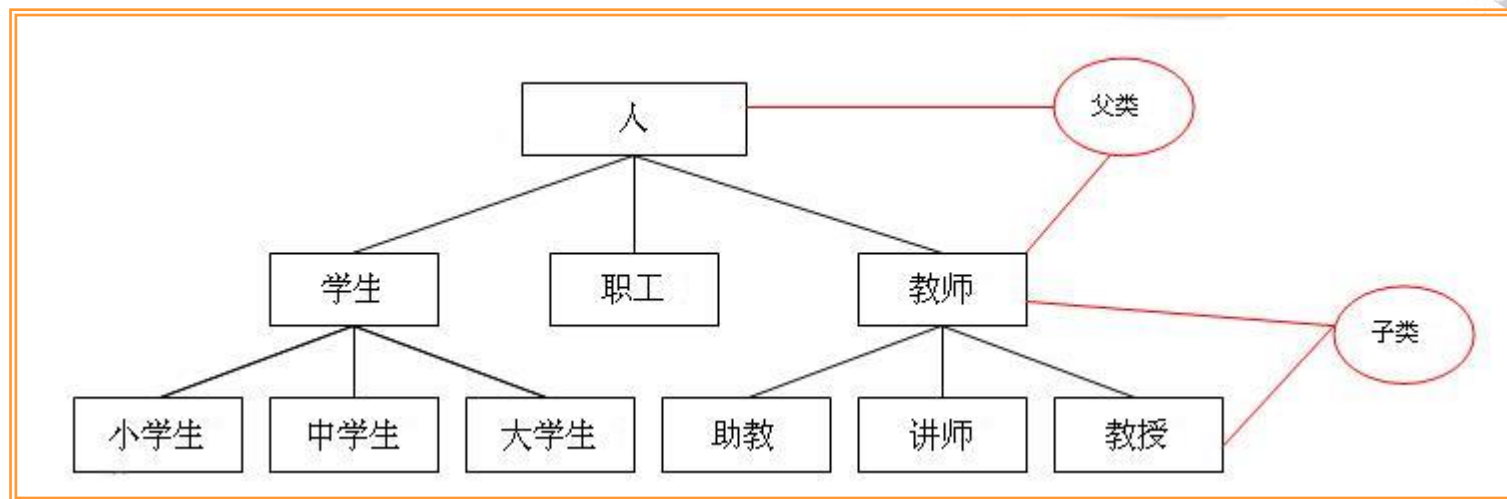
- **main class:** Line对象
布局算法
- **Class Station:** 站点属性
- **Class Label**
- **Class Line:** 属性
Station对象



类的继承

- 类之间的继承关系是现实世界中遗传关系的直接模拟，它表示类之间的内在联系以及对属性和操作的共享，即子类可以沿用父类（被继承类）的某些特征。子类也可以具有自己独立的属性和操作。
- 子类从父类继承有两个主要的方面：
 1. **属性的继承**。例如，公司是一个父类，一个公司有名称、地址、经理、雇员等，这些都属于结构方面。
 2. **方法的继承**。一个父类定义了若干操作，如一个公司要有项目开发、利润创造、经理任命、员工录用等操作，子公司也将继承这些行为。

继承性



- ✓ 继承是OOP中一种**由已有的类创建新类的机制**。
- 一个父类（公共属性的集合）可以有多个子类（是父类的特殊化），一个子类只可以有一个父类
- 子类可以直接使用父类的方法，也可以根据需求修改父类中已经定义的方法（即不改变方法名，而修改方法的参数个数、参数类型）。父类和子类可有同名的方法（多态性）
- 单继承 vs 多重继承

Practice - 类的定义



- 定义一个People类，包括：
 - 两个成员属性：年龄、姓名
 - 构造方法，用于设置两个成员属性的初值
 - 成员方法 `setinfo`，供外部调用，改变两个成员属性的值
 - 成员方法 `showname`，供外部调用，返回“姓名”的值



类的定义

```
1. //创建一个新人物
2. class People {
3.     private int age;
4.     private String name;
5.     public void setinfo(int myage age, String myname){
6.         age = myage; name = myname;
7.     }
8.     public String showname(){
9.         return "["+ age + ", " + name + "]";
10.    }
11. }
12. public class TestPeople{
13.     public static void main(String args[]){
14.         People someguy = new People();
15.         someguy.setinfo( 18, "SUN" );
16.         System.out.println(someguy.showname());
17.     }
18. }
```



声明对象时的内存模型

- 当用People类声明一个对象**someguy**:

People someguy = new People();

- 内存模型

- 在“**栈**”里仅建立了“**someguy**”对象的**引用(reference)**

someguy

0xAB12

未分配存储空间的对象

Code – 类的继承



```
1. class Point2D{
2.     int X, Y;        //Point2D私有属性
3.     public Point2D(){ //无参的Point2D构造方法
4.         X=0; Y=0;
5.     }
6.     Point2D(int x, int y){//有参的Point2D构造方法
7.         X=x; Y=y;
8.         System.out.println("Point2D Constructed With Parameters!");
9.     }
10.    public String toString(){
11.        return ("Point position " + this.X + " " + this.Y);
12.    }
13. }
14. class Point3D extends Point2D {
15.     int X, Y, Z;    //Point3D私有成员属性
16.     public Point3D(int x, int y, int z){ //有参构造函数
17.         super(x, y);  Z=z;
18.     }
19.     public String toString(){
20.         return (super.toString()+" "+ this.Z);
21.     }
22. }
```

Code - 类的继承(续)



```
1. public class Test_Point {  
2.     public static void main(String args[]){  
3.         Point3D p1, p2;  
4.         p1=new Point3D(1, 2, 3); this.X ; super.X  
5.         System.out.println(p1.toString());  
6.     }  
7. }
```

课件源码：

PointTest.java

PointTest程序验证的知识点



○ 设计父类Point2D

- 定义Point2D成员属性 X, Y
- 定义无参、有参构造方法：重载
- 定义改变坐标、输出坐标的成员方法

○ 设计子类Point3D

- 继承父类所有的成员属性、构造函数
- 重新定义的Point3D成员属性 X, Y, Z
- 定义无参、2种不同的构造方法：重载
- 定义改变坐标、输出坐标的成员方法：覆盖

○ 创建子类对象的过程中究竟做了什么？

- 例1：创建Point3D对象（无参）
- 例2：创建Point3D对象（3个整形参数）
- 例3：创建Point3D对象（1个Point2D对象，1个整形参数）

○ 父类引用reference可以指向子类对象

类的继承（类的初始化顺序）

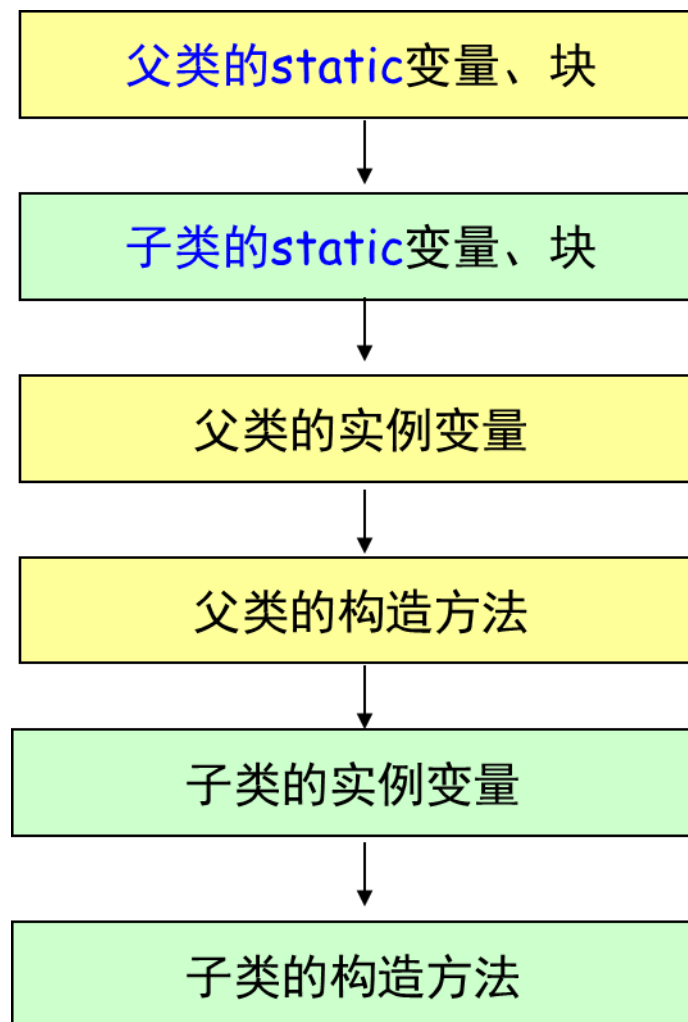


- 总是先初始化父类的成员；
- 总是先初始化static成员；
- 总是先初始化变量，后初始化方法；

课件源码：

[InitialOrderTest.java](#)

[InitialOrderTest2.java](#)



属性继承与隐藏



- 并不因为父类就意味着它有更多的功能。
- 恰恰相反，子类比它们的父类具有更多的功能。因为子类是父类的扩展，增加了父类没有的属性和方法。
 - 子类不能访问父类的**private**成员，但子类可以访问其父类的**public**成员。
 - **protected**访问是**public**和**private**访问之间一个保护性的中间层次。
 - 虽然被继承的父类成员没有子类声明中列出，但是这些成员确实存在于子类中。



构造方法的继承

○ 构造方法的继承应遵守以下原则

1. 子类可以**无条件的继承父类不含参数的构造方法**
2. 如果子类没有构造方法，则它继承父类无参数的构造方法作为自己的构造方法；
如果子类有构造方法，那么在创建子类对象时，则将先执行继承下来的父类的构造方法，然后再执行自己的构造方法
3. 对于父类中包含有参数的构造方法，子类可以通过在自己的构造方法中使用**super**关键字来引用，而且**必须是子类构造方法中的第一条语句**。

子类中与父类同名的成员变量的使用



- 在子类和父类中都存在同名的非私有成员变量时，对子类，只存在子类的成员变量，而不能引用父类的成员变量，如果一定要引用父类的成员变量，需要加上“**super**”关键字。
- 下面给出一个例子说明如何使用子类来引用父类的成员变量
 - // 定义父类为Point2D，再定义子类Point3D
 - // 子类与父类具有相同的成员变量
 - // 若使用父类的 成员变量，应在变量前增加super前缀
- 所以子类的实例在进行赋值时，只改变子类成员变量的值，对父类成员变量值没有影响。
- 子类覆盖父类中方法时，不能缩小“访问范围”。
 - 如：父类中方法是public的，子类在覆盖时不能改为protected 或private

访问控制符



- 允许类（和包）创建者声明哪些东西是**程序员**可以使用的，哪些是不可使用的。
- 分别包括：**public**，**默认**（无关键字），**protected**以及**private**。下面的列表说明访问控制修饰符含义：

类的访问控制修饰符			类成员的访问控制修饰符			
符号	public	无关键字	public	无关键字	protected	private
含义	公共访问	默认访问	公共访问	默认访问	保护访问	私有访问

```
package somePackage;
```

```
public class A
{
    public int v1;
    protected int v2;
    int v3.//package
           //access
    private int v4;
```

```
public class B
{
    can access v1.
    can access v2.
    can access v3.
    cannot access v4.
```

In this diagram, "access" means access directly, that is, access by name.

```
public class C
    extends A
{
    can access v1.
    can access v2.
    can access v3.
    cannot access v4.
```

```
public class D
    extends A
{
    can access v1.
    can access v2.
    cannot access v3.
    cannot access v4.
```

```
public class E
{
    can access v1.
    cannot access v2.
    cannot access v3.
    cannot access v4.
```

A line from one class to another means the lower class is a derived class of the higher class.

If the instance variables are replaced by methods, the same access rules apply.

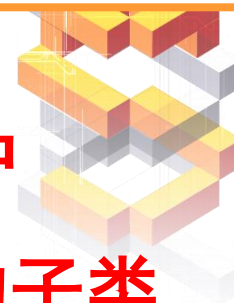


公共访问控制符public



- 无论是类或者类成员，只要被声明成**public**，就意味着它可以被从任何地方访问
 - 一个类作为整体对程序的其他部分可见，并不能代表类内的所有属性和方法也同时对程序的其他部分可见，前者只是后者的必要条件，类的属性和方法能否为所有其他类所访问，还要看这些属性和方法自己的访问控制符
 - 实例变量采用**public关键字**，这是一种**非常糟糕**的做法，破坏封装

保护访问控制符protected



- 用protected修饰的类成员不但对于**同一个包中的其他类**是可见的，而且对于**其他包中该类的子类**也是可见的。

比较项目	private	默认	protected	public
同一类中可见	是	是	是	是
同一个包中对子类可见	否	是	是	是
同一个包中对非子类可见	否	是	是	是
不同包中对子类可见	否	否	是	是
不同包中对非子类可见	否	否	否	是

Code - protected方法



1. //定义类的**protected** 方法，通过创建对象来引用此方法

```
2. class Max {  
3.     private int x,y;  
4.     protected int play(int s, int t) {  
5.         int m;  
6.         x=s;  
7.         y=t;  
8.         m=(x>y)?x/y:y/x;  
9.         return m;  
10.    }  
11. }  
12. public class TestMax {  
13.     public static void main(String args[]) {  
14.         int result;  
15.         Max ss = new Max();  
16.         result=ss.play(5,45);  
17.         System.out.println(" result= "+result);  
18.     }  
19. }
```

使用**protected**
定义的成员变量
或方法具有二重
性，类本身、子
类或包中的方法
可以访问它们，
而其它类没有访
问权限



默认访问控制符

- 假如类或者类成员**不含有**一个明确的访问说明，说明它具有**默认的访问控制特性**。
- 默认的访问控制权规定类或者类成员**只能被同一个包中的其他代码所访问，在包外不可见**。这种访问特性称为**包访问性**。
- Java语言中的包是类和接口的集合，它从更高级别上提供了对类和接口的存取保护和命名空间管理。

包



- 未设定public或private访问权限, **包内的所有类可以访问它们, 包外的无法访问。**
- 一个Java程序文件只能定义一个public类, 而且程序文件必须与它同名。 **为其它程序共享的类须经过编译进行打包, 形成一个包文件, 然后用import语句加以引用。**
- 打包
 - 打包是在Java程序编译时进行的, 注意参数 -d
 - 被编译程序所在路径 **javac -d** 被编译程序所在路径 被编译程序名.Java
 - 被编译程序的第一行用 **package 包名** ; 给出它被打入的包, 该包与系统创建的子目录同名

多态性



- 定义:由于父类和子类可以有同名的方法，在运行时JVM根据方法的参数个数和类型的不同来查找、决定执行哪个版本的方法，称为多态性
- 在程序执行时，JVM对对象某一方法的查找是从该对象类所在层次开始，沿类等级逐级向上进行，把第一个方法作为所要执行的方法。所以，子类的方法可以屏蔽父类的方法
- 例如：
 - `java.io.InputStream.read(byte[] b)`
 - `java.io.FileInputStream.read(byte[] b)`

重载 - overwrite



- 所谓**方法的重载**，是在类中创建了多个方法，它们具有相同的名称，但有不同的参数和不同的实现。
- 在调用方法时依据其参数个数及类型**自动选择相匹配的方法**去执行。达到各种对象处理类似问题时具有统一的接口目的。
- 可以定义多个构造方法，根据类实例初始化时给定的**参数列表**决定使用哪一个
- 如：`java.util.Vector.remove(int index)`
`java.util.Vector.remove(Object o)`

覆盖 - override



- Java 允许子类对父类的同名方法进行重新定义，因此出现了同名方法而方法的内容不同的情况。
出现这种情况时，子类引用方法需要指明引用的父类的方法还是子类的方法。
- 引用父类方法时采用 **super** 加上成员变量或方法名
如: **super.成员变量** 或 **super.方法名(参数表)**
- 如: **java.io.BufferedReader.read()**
覆盖了
java.io.Reader.read()

this & super 的区别



- 在引用父类的方法时，为了明确的是父类方法需在方法前加**super**。
- 有时，因为继承虽然对子类的方法进行了重新定义，但是当前的方法中的形参名与成员变量名相同或与方法中的局部变量名相同，为了明确其含义，就要采用**this关键字**加以区别。
- 下面的例子中在创建Point3D类对象时引用Point2D类的构造器，使用super关键字来调用父类的方法**super(a,b)**，对于限制继承的成员变量加上protected 说明，由于Point类和Circle类都存在**toString()**方法，所以在引用**toString()**方时用**this**和**super**加以区别。



多态机制

- 多态机制是面向对象程序设计的一个重要特征。
- 多态的特点是采用同名的方式，根据调用方法时传送的参数的多少以及传送参数类型的不同，调用不同的方法，这样对于类的编制而言，可以采用同样的方法获得不同的行为特征。
- 多态性
 1. 编译时多态
表现为方法名相同，而参数表不同。
典型：System.out.println(); 它的参数有很多种。
 2. 运行时多态
程序运行的时候判断到底是哪个类（父类还是子类的），进而确定实际调用的方法
 3. 对象状态的判断
由于子类对象可以作为父亲类对象使用，有时需要判断这个对象究竟属于哪个类。Java提供 instanceof 关键字

目 录



1

面向对象概念

2

面向对象的三大特性

3

抽象类与接口

4

“设计模式”初体验 – 简单工厂模式



什么是抽象类

- 在面向对象的概念中，**所有的对象都是通过类来描绘的**，但是反过来却不是这样。**并不是所有的类都是用来描绘对象**的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是**抽象类**。
- 抽象类往往用来表征我们在对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。
- 正是因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是**不能实例化**的。



抽象类和方法

- 抽象类是专门设计为子类继承的类，抽象类通常都包括一个或多个抽象方法（**只有方法说明，没有方法体**），抽象方法体中内容，根据继承抽象类的子类的实际情况，有子类完成其抽象方法的代码。

- 定义抽象类的一般形式

```
abstract class 类名称 {
```

```
    成员变量;
```

```
    方法( ){ /* do something */ }; //定义一般方法
```

```
    abstract 方法( ); //定义抽象方法
```

```
}
```

注意无论抽象类和抽象方法都以关键字 **abstract** 开头

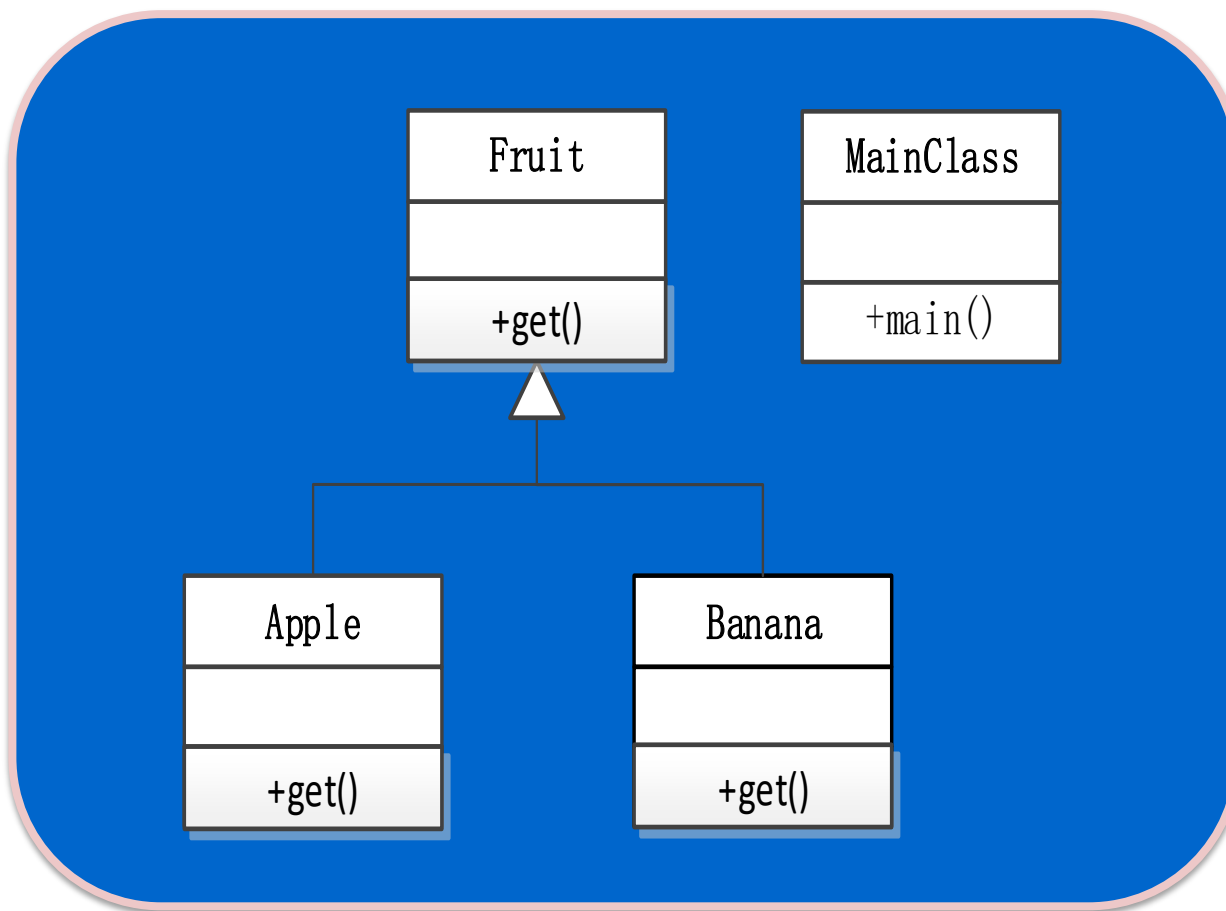


其它说明

说明：

- 抽象类 里 可以有 非抽象的方法。
- 抽象类 里 可以没有 抽象方法。
- 即使没有抽象方法的抽象类也不允许实例化。
- 一个类里有abstract方法的话，该类必须为abstract。
- 抽象方法一定不能有方法体（哪怕是空方法体）。

抽象类例



接口(interface)——基本概念



- 接口是一个更“纯”的抽象类。(不能含有非abstract方法，变量)
- 接口的语法格式
- 接口里面定义的抽象方法被自动赋予public abstract，接口里面的变量自动被赋予public static final；
- 接口中声明方法时，不能使用native、static、final、synchronized、private、protected等修饰符；

接口定义



- Java中接口定义的一般形式如下：

```
[访问控制符] interface 接口名 {  
    抽象方法声明  
    成员变量声明  
}
```

- 从上面的语法规则可以看出来，定义接口与定义类非常相似，实际上**可以把接口理解成为一种特殊的抽象类**。
- 接口体现了程序设计的多态性和高内聚低耦合的设计思想。

课件代码：
TestInterface.java



接口定义的说明

- 接口不能被实例化。
- 和类的访问级别类似，接口的访问控制符是 **public** 或者 **默认访问**。
 - 如果接口本身定义成 **public**，所有抽象方法和成员变量都是 **public** 的，不论是否使用了 **public** 访问修饰符
- 接口中的 **所有方法** 都是抽象方法，在接口中只能给出这些抽象方法的方法名、返回值类型和参数列表，没有方法体。
(类似C++的“纯虚类”)
- 接口中可以有成员变量（但变量 **不能用** **private** 和 **protected** 修饰）
 - 接口中的变量，本质上都是 **static**，而且是 **final** 的，无论是否用 **static** 修饰，必须以常量值初始化
 - 实际开发中，常在接口中定义常用的变量，作为全局变量使用
 - 访问形式：接口名.变量名
- 一个接口不能继承其它的类，但是可以继承别的接口
- 一个类可以实现多个接口



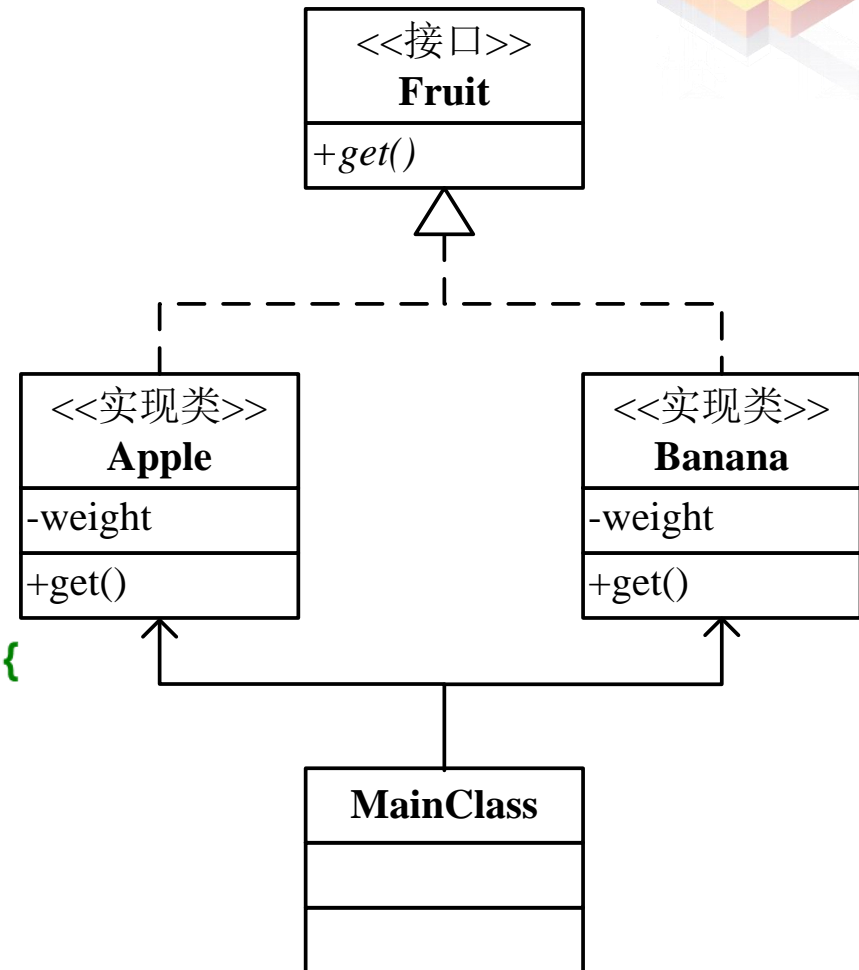
接口引用与实现对象的关系

- 同子类对象可以赋给父类引用一样，**接口引用变量可以指向实现对象。**
 - 可以定义一个接口类型的引用变量，并将任何实现了该接口的类的对象赋给它。当通过接口引用变量调用方法时，将根据**动态绑定**的原则来决定方法的正确调用。
 - 需要注意的是，虽然实现对象可以赋给接口引用，但该引用只能访问被它的接口定义声明的内容，而隐藏实现对象中的特殊内容。



Code - 多态

```
1. public interface Fruit {
2.     public void get(); //采集
3. }
4. public class Apple implements Fruit {
5.     public void get(){ //采集
6.         System.out.println("采集苹果");
7.     }
8. }
9. public class Banana implements Fruit {
10.    public void get(){ //采集
11.        System.out.println("采集香蕉");
12.    }
13. }
14. public class MainClass {
15.    public static void main(String[] args) {
16.        //实例化一个Apple
17.        Apple apple = new Apple();
18.        //实例化一个Banana
19.        Banana banana = new Banana();
20.        apple.get();
21.        banana.get();
22.    }
23. }
```



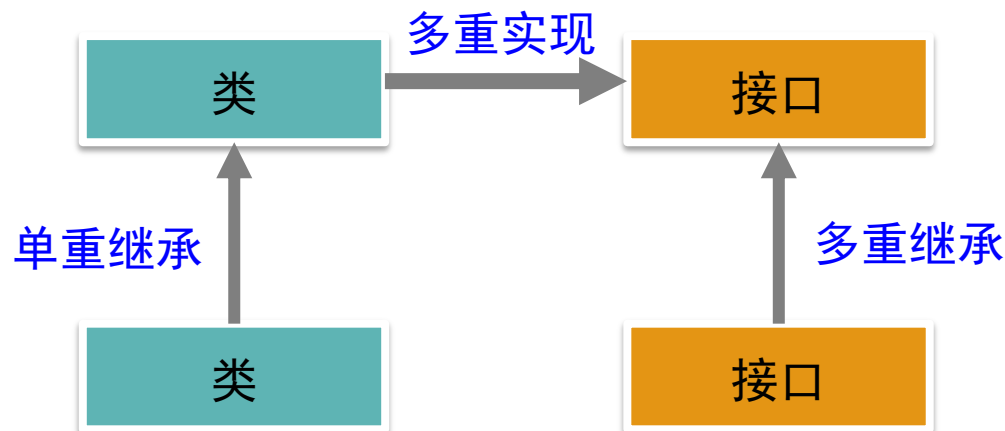
课件代码：“多态_水果之例”

abstract class 和 interface



从语法的角度

- 一个类实现一个接口，使用关键字 `implements`.
- 一个类可以实现(`implements`)多个接口、一个接口可以继承(`extends`)多个接口；



abstract class 和 interface



○ 从设计层面 《Effective java》

1. 接口比抽象类好，一般情况下，如果能用接口就不用抽象类。
2. 从设计上讲和抽象类有很大不同。
 - 接口表示实现类**尊崇接口的“协议”**，**并不是接口的特征**，接口既然定义了就不能随便修改。所以设计上，接口不应该很“大”。
(后面的线程实现就是例子)
 - 如果一个子类继承了抽象类，已经决定了这个类的主要特征。

抽象类 VS 接口



	区别点	抽象类	接口
1	定义	abstract修饰的类	抽象方法和全局常量的集合
2	组成	构造方法、抽象方法、普通方法、常量、变量	抽象方法、常量
3	使用	子类继承抽象类（extends）	子类实现接口（implements）
4	关系	抽象类可以实现多个接口	接口不能继承抽象类，但允许实现多个接口
5	设计模式	模板设计	工厂设计、代理设计
6	对象	都通过对象的多态性产生实例化对象	
7	局限	抽象类有单继承的局限	接口没有此局限
8	实际使用	作为一个模板	是作为一个标准或是表示一种能力
9	选择	若抽象类和接口都可以使用，优先使用接口，避免单继承的问题	

目 录



1

面向对象概念

2

面向对象的四大特性

3

抽象类与接口

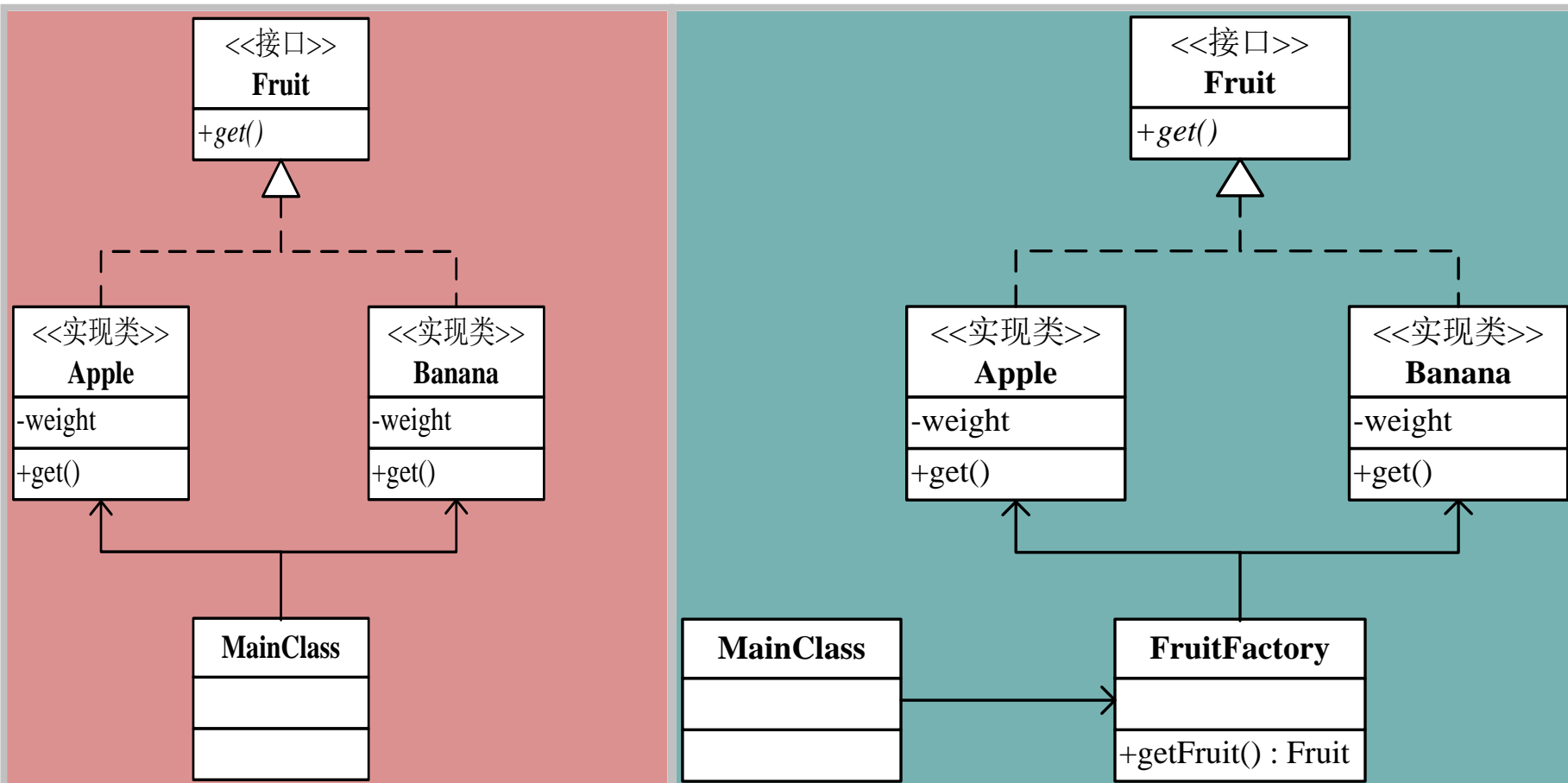
4

“设计模式”初体验 – 简单工厂模式



工厂模式

- 简单工厂模式属于类的创建型模式,又叫做**静态工厂**方法模式。通过专门定义一个类来负责创建其他类的实例,被创建的实例通常都具有共同的父类。



代码 - 1 - 水果类



```
1. interface Fruit {  
2.     //采集  
3.     public void get();  
4. }  
5. class Apple implements Fruit{  
6.     //采集  
7.     public void get(){  
8.         System.out.println("采集苹果");  
9.     }  
10. }  
11. class Banana implements Fruit{  
12.     //采集  
13.     public void get(){  
14.         System.out.println("采集香蕉");  
15.     }  
16. }
```

课件代码:

SimpleFactory_水果之例

代码 – 2 – 简单工厂类



```
1. class FruitFactory {
2.     //get方法, 获得所有产品对象
3.     public static Fruit getFruit( String type ) throws Exception {
4.         Fruit fruit = null;
5.         if ( type.equalsIgnoreCase( "apple" ) ) {
6.             fruit = new Apple();
7.             return fruit;
8.         } else if ( type.equalsIgnoreCase( "banana" ) ) {
9.             fruit = new Banana();
10.            return fruit;
11.        } else {
12.            System.out.println("找不到相应的实例化类");
13.            return null;
14.        }
15.    }
16. }
```

代码 – 3 – 主类



```
1. public class MainClass {  
2.     public static void main(String[] args) throws Exception {  
3.         //简单工厂方法创建水果对象  
4.         Fruit apple = FruitFactory.getFruit("Apple");  
5.         Fruit banana = FruitFactory.getFruit("Banana");  
6.         //调用水果对象方法  
7.         apple.get();  
8.         banana.get();  
9.     }  
10. }
```

模式中包含的角色及其职责



1. 抽象（Product）角色

简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

2. 具体产品（Concrete Product）角色

简单工厂模式所创建的具体实例对象

3. 工厂（Creator）角色

简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类可以被外界直接调用，创建所需的产品对象。

简单工厂模式的优缺点



○ 优点：

- 在这个模式中，工厂类是整个模式的关键所在。
- 它包含必要的判断逻辑，能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。用户在使用时可以直接根据工厂类去创建所需的实例，无需了解对象是如何创建以及如何组织的。
- 有利于整个软件体系结构的优化。

○ 缺点：

- 简单工厂模式的缺点也正体现在其工厂类。
- 工厂类集中了所有实例的创建逻辑，“高内聚”方面不佳。
- 另外，当系统中的具体产品类不断增多时，可能会出现要求工厂类也要做相应的修改，扩展性并不很好。

课后思考



1. 什么是对象？如何创建对象？
2. 什么是类的封装？如何对成员变量和方法的访问权限的设置达到数据封装的目的？
3. 按要求编写一个Java应用程序：(1)定义一个抽象类CanCry，描述会吼叫的方法`public void cry()`。(2)分别定义狗类（Dog）和猫类（Cat），实现`cry()`方法。实现方法的功能分别为：打印输出“我是狗，我的叫声是汪汪汪”、“我是猫，我的叫声是喵喵喵”。(3)定义一个主类，定义一个`void makeCry(CanCry c)`方法，其中让会吼叫的动物吼叫。在主类main方法中创建狗类对象（dog）、猫类对象（cat），调用`makecry()`方法，让狗和猫吼叫。
4. 补充阅读：
 - 《大话设计模式》，程杰，清华大学出版社



Thank You !