面向对象程序设计

第 7 讲 设计模式

刘进

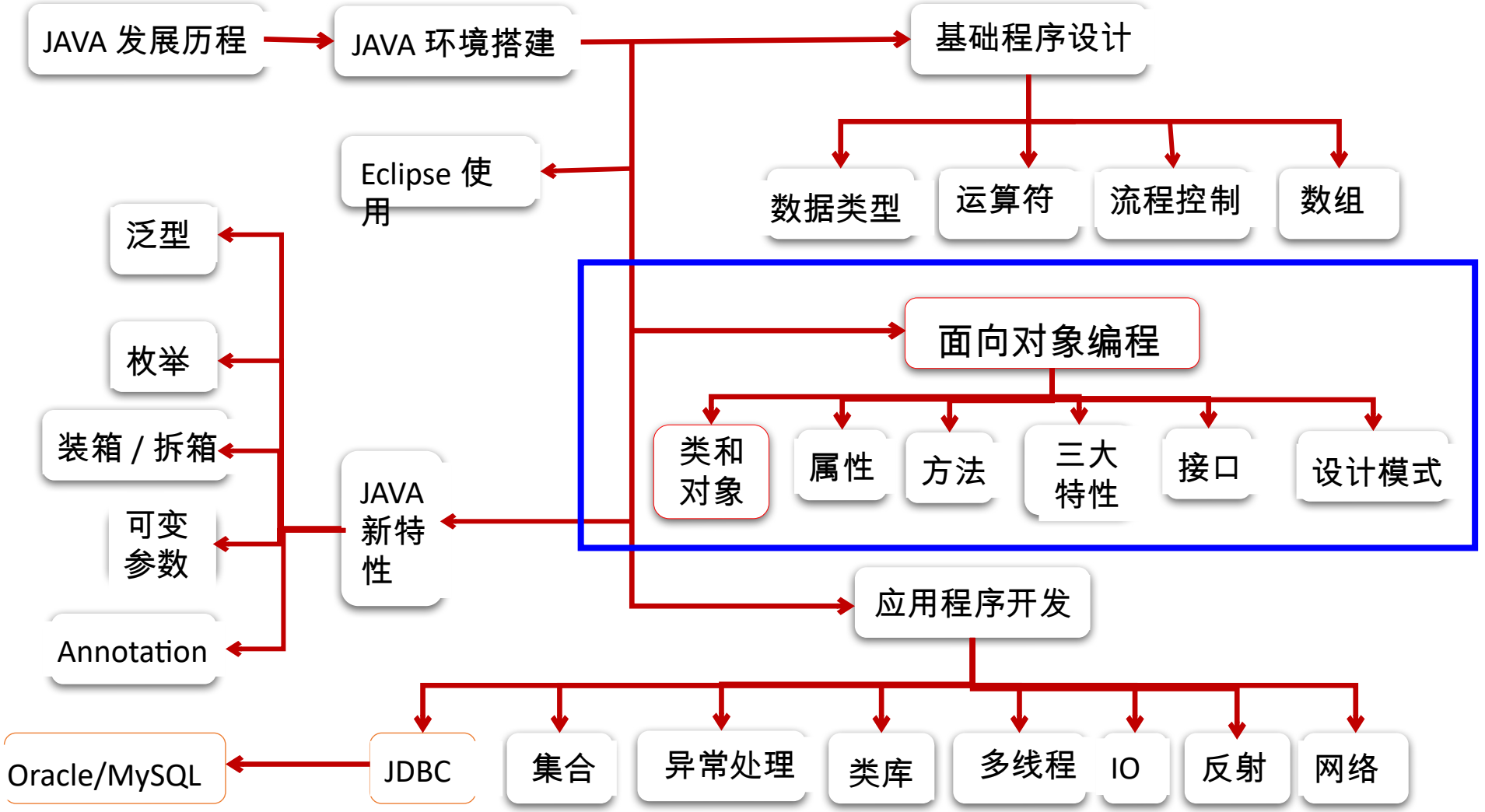
2230652597@qq.com

OOP 教辅 2023 秋季 QQ 群：

837966056

Java 基础知识图解

主要知识点



第 7 讲 Java OOP- 设计模式

7.1 设计模式

7.2 Static 、类变量 / 方法与单例设计模式 Singleton

7.3 抽象类与模板方法设计模式 TemplateMethod

7.4 接口与简单工厂模式

什么是设计模式

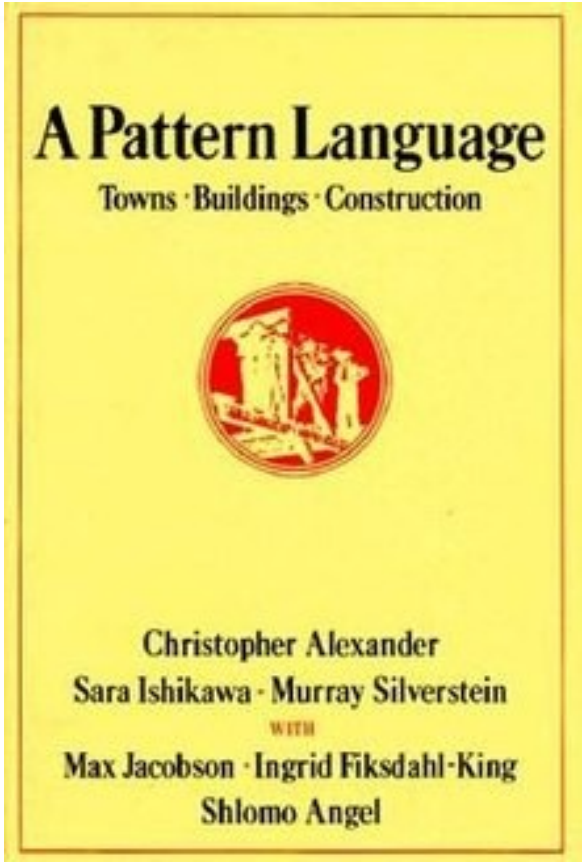
针对特定场景
可重复的解决方案

每一个设计模式描述一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次一次地使用该方案而不必做重复劳动。

设计模式的起源

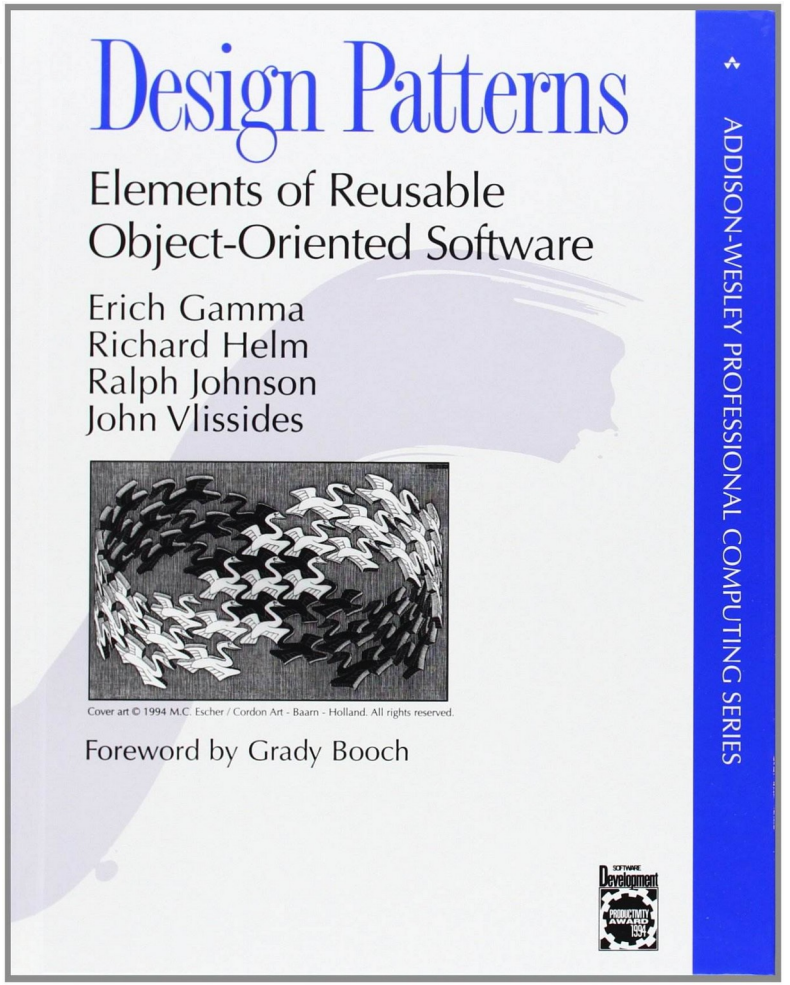
软件领域模式起源于建筑学。

1977 年，建筑大师 Alexander 出版了《A Pattern Language：Towns, Building, Construction》一书。受 Alexander 著作的影响，Kent Beck 和 Ward Cunningham 在 1987 年举行的一次面向对象的会议上发表了论文：《在面向对象编程中使用模式》。的设计



GOF 之著作

目前，被公认在设计模式领域最具影响力的著作是 Erich Gamma 、 Richard Helm 、 Ralph Johnson 和 John Vlissides 在 1994 年合作出版的著作：《 Design Patterns ： Elements of Reusable Object-Oriented Software 》（中译本《设计模式：可复用的面向对象软件的基本原理》或《设计模式》），该书被广大喜爱者昵称为 GOF （ Gang of Four ）之书，被认为是学习设计模式的必读著作， GOF 之书已经被公认为是设计模式领域的奠基之作。



学习设计模式的重要性

学习设计模式不仅可以使我们使用好这些成功的模式，更重要的是可以使我们更加深刻地理解面向对象的设计思想，非常有利于我们更好地使用面向对象语言解决设计中的问题。

第 7 讲 Java OOP- 设计模式

单例模式

7.1 设计模式

7.2 Static 、类变量 / 方法与单例设计模式 Singleton

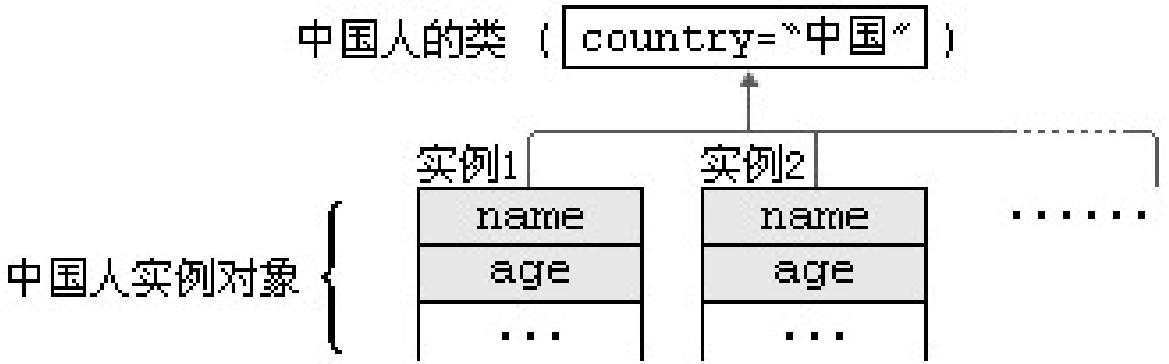
7.3 抽象类与模板方法设计模式 TemplateMethod

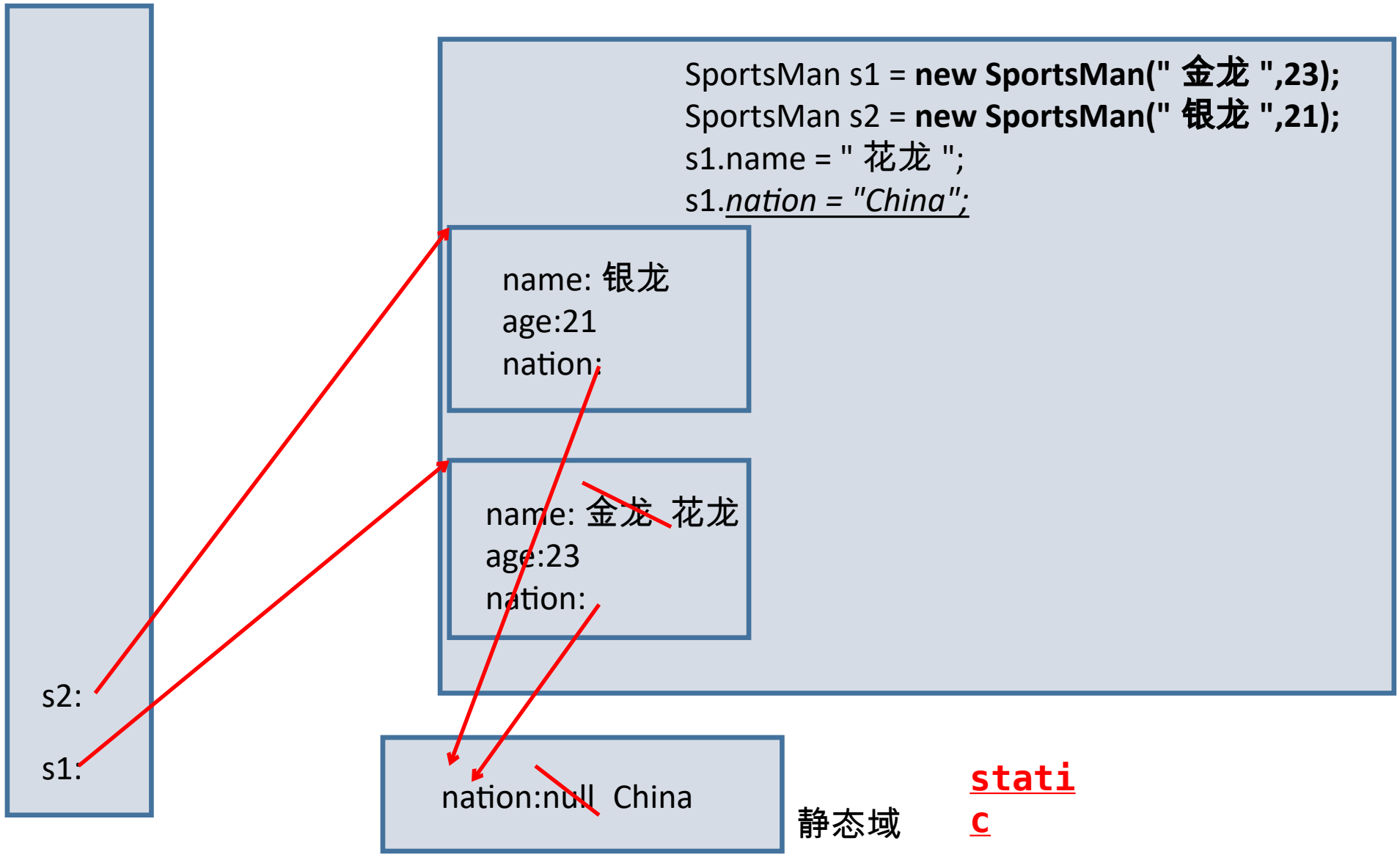
7.4 接口与简单工厂模式

关键字 static

static

当编写一个类时，其实就是在描述其对象的属性和行为，而并没有产生实质上的对象，只有通过 new 关键字才会产生出对象，这时系统才会分配内存空间给对象，其方法才可以供外部调用。有时候希望无论是否产生了对象或无论产生了多少对象的情况下，某些特定的数据在内存空间里只有一份，例如所有的中国人都有个国家名称，每一个中国人都共享这个国家名称，不必在每一个中国人的实例对象中都单独分配一个用于代




SportsMan s1 = new SportsMan(" 金龙 ",23);
SportsMan s2 = new SportsMan(" 银龙 ",21);
s1.name = " 花龙 ";
s1.nation = "China";
name: 银龙
age:21
nation:
name: 金龙 花龙
age:23
nation:
s2:
s1:
nation:null China
静态域
stati
c

关键字 static

- class Circle{

  private double radius;

  public Circle(double radius){this.radius=radius;}

  public double findArea(){return Math.PI*radius*radius;}}

- 创建两个 Circle 对象
  - Circle c1=new Circle(2.0); //c1.radius=2.0
  - Circle c2=new Circle(3.0); //c2.radius=3.0
- Circle 类中的变量 radius 是一个实例变量 (instance variable)，它属于类的每一个对象，不能被同一个类的不同对象所共享。
- 上例中 c1 的 radius 独立于 c2 的 radius ，存储在不同的空间。 c1 中的 radius 变化不会影响 c2 的 radius ，反之亦然。

如果想让一个类的所有实例共享数据，就用类变量！

类属性、类方法的设计思想

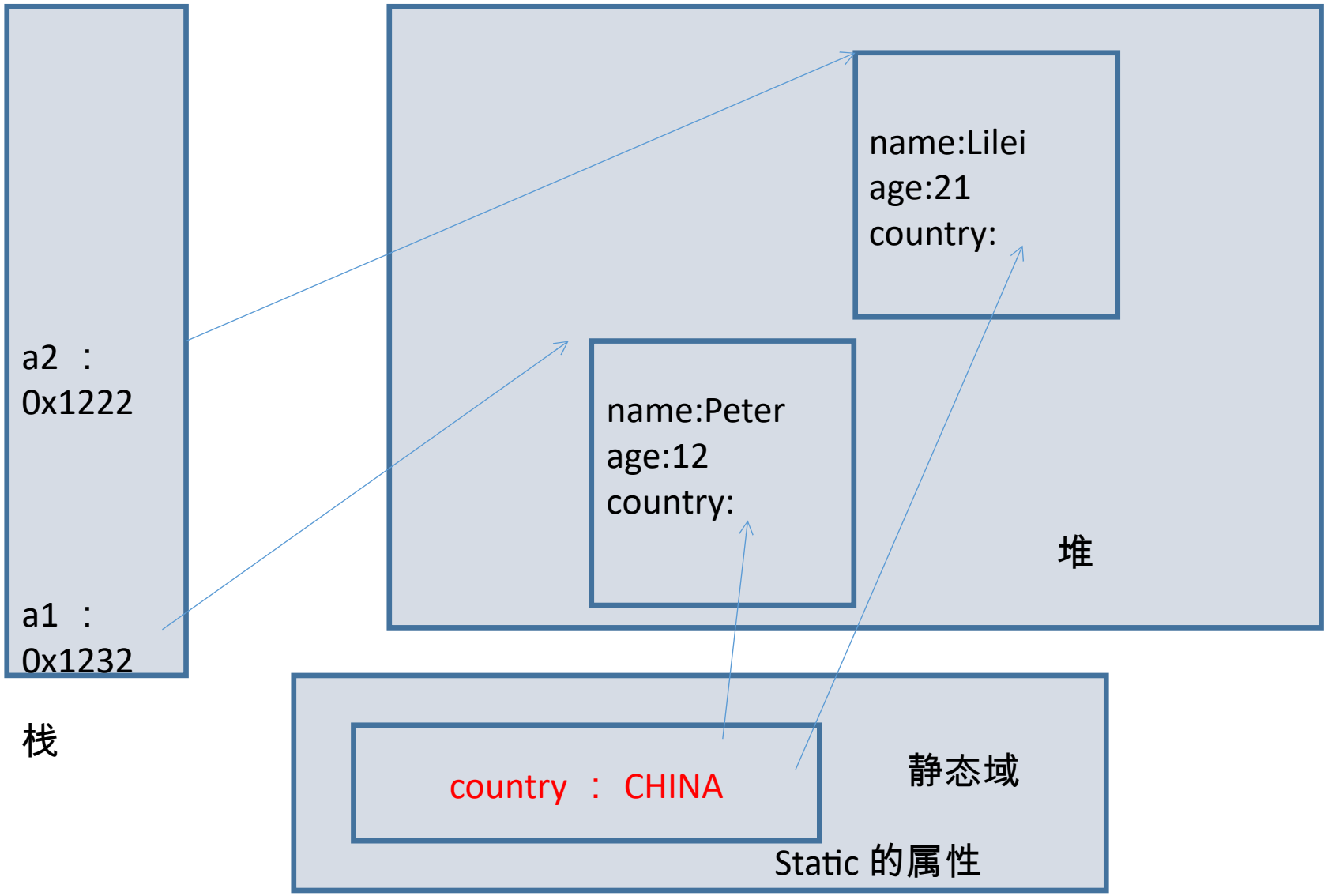
static

- 类属性作为该类各个对象之间共享的变量。**在设计类时，分析哪些类属性不因对象的不同而改变，将这些属性设置为类属性。相应的方法设置为类方法。**
- **如果方法与调用者无关，则这样的方法通常被声明为类方法，由于不需要创建对象就可以调用类方法，从而简化了方法的调用**

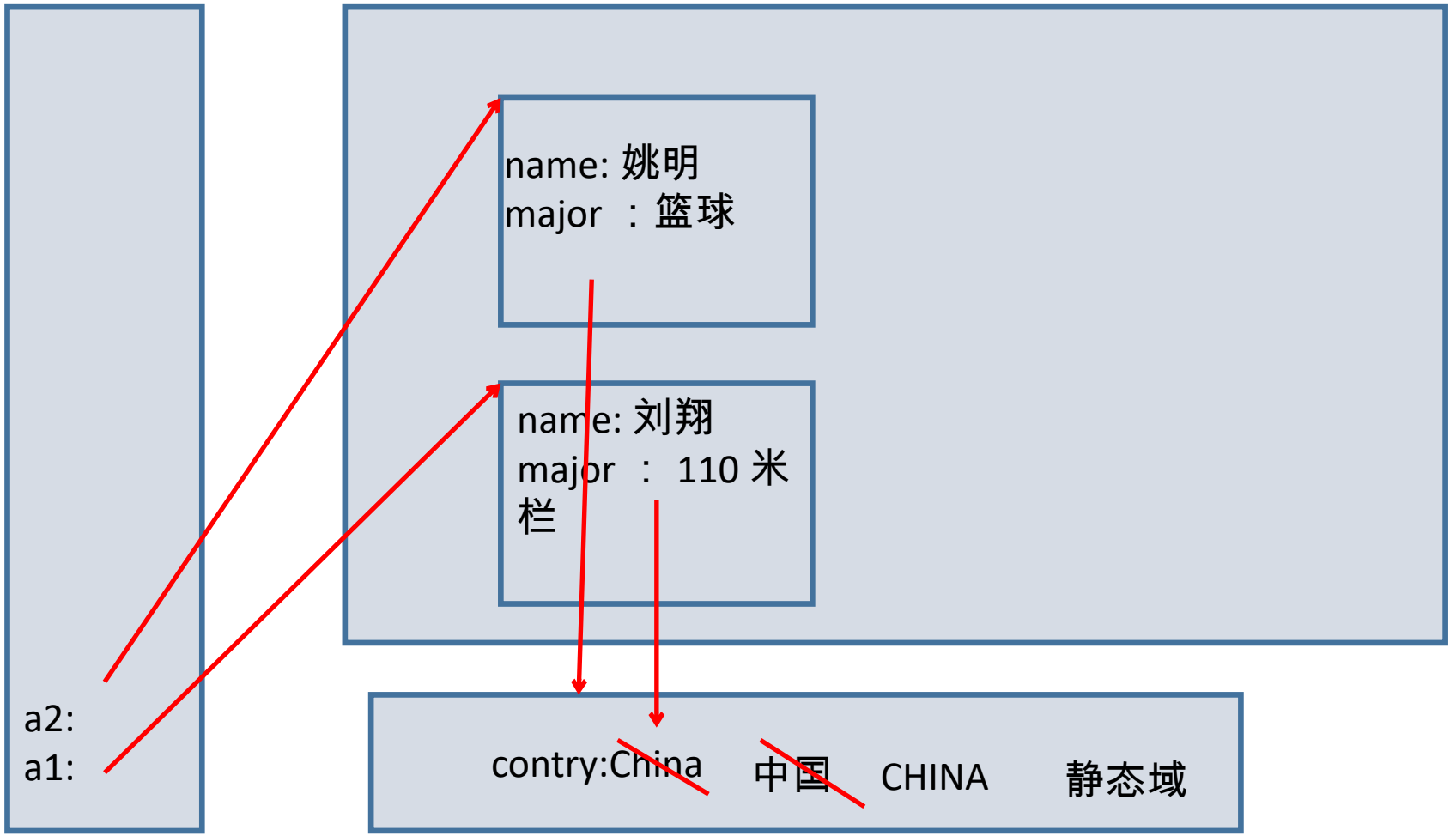
关键字 static

- 使用范围：
  - ✓在 Java 类中，可用 static 修饰属性、方法、代码块、内部类
- **被修饰后的成员具备以下特点：**
  - **✓随着类的加载而加载**
  - **✓优先于对象存在**
  - **✓修饰的成员，被所有对象所共享**
  - **✓访问权限允许时，可不创建对象，直接被类调用**

stati
c


a2 ：
0x1222
a1 ：
0x1232
栈
name:Lilei
age:21
country:
name:Peter
age:12
country:
堆
country ： CHINA
静态域
Static 的属性

static



Athlete a1 = **new Athlete(" 刘翔 ","110 米栏 ","China");**
Athlete a2 = **new Athlete(" 姚明 "," 篮球 ","China");**

Outline | Circle.java | TestStatic.java

```java
//Circle.java
package ch004;

public class Circle {
    private double radius;
    public static String name = "这是一个圆";
    public static String getName(){
      return name;
    }
    public Circle(double radius) {
      getName();
      this.radius = radius;
    }
    public double findArea() {
      return Math.PI * radius * radius;
    }
    public void display(){
      System.out.println("name:"+name+"radius:"+radius);
    }
}
```

stati
c

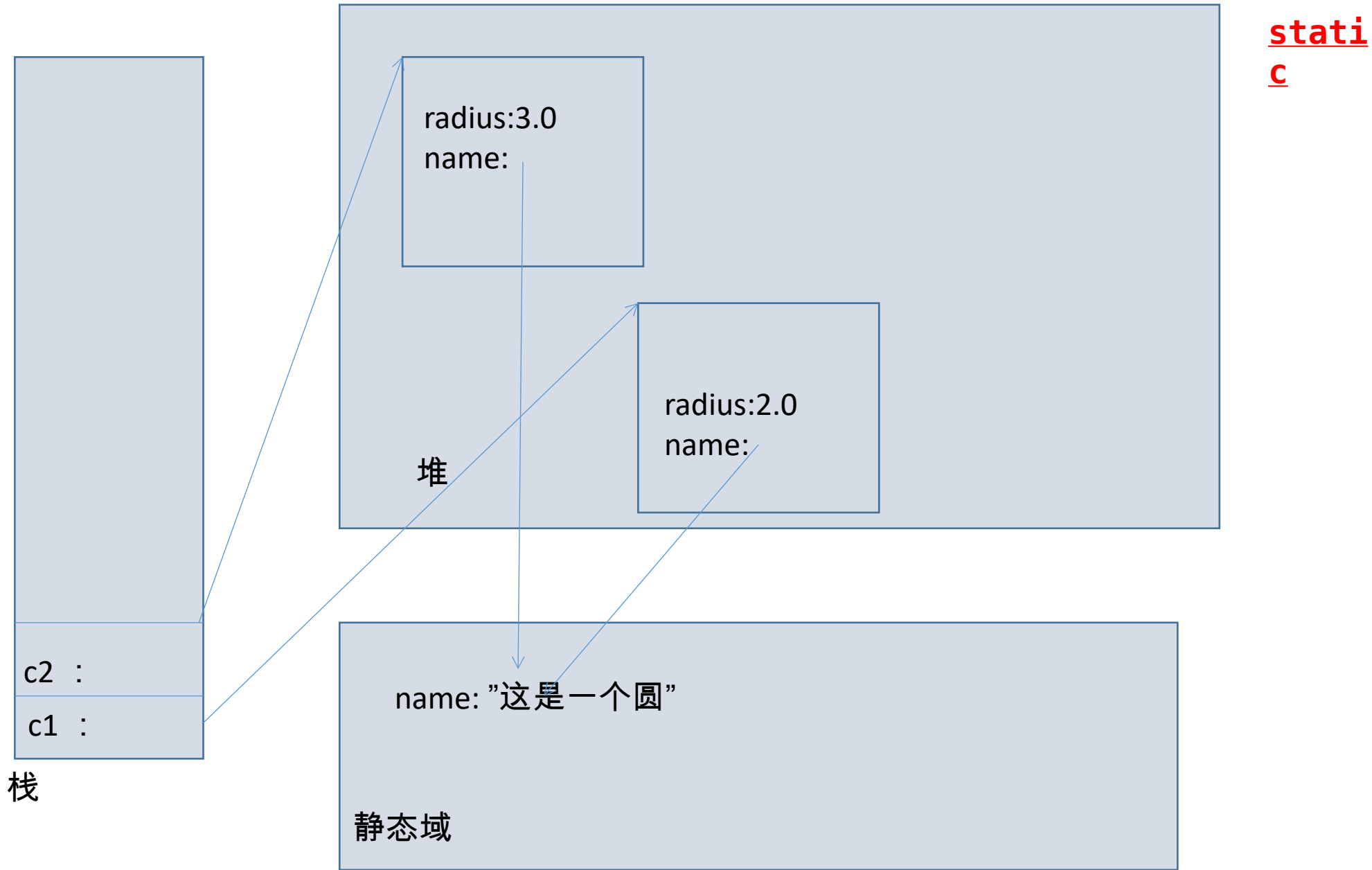
Outline Circle.java TestStatic.java

```java
//TestStatic.java
package ch004;

public class TestStatic {
    public static void main(String[] args) {
        Circle c1 = new Circle(2.0);
        Circle c2 = new Circle(3.0);
        c1.display();
        c2.display();
    }
}
```

Problems Console Debug

<terminated> TestStatic [Java Application] C:\Program Files\Java\jdk1.8

```
name:这是一个圆radius:2.0
name:这是一个圆radius:3.0
```

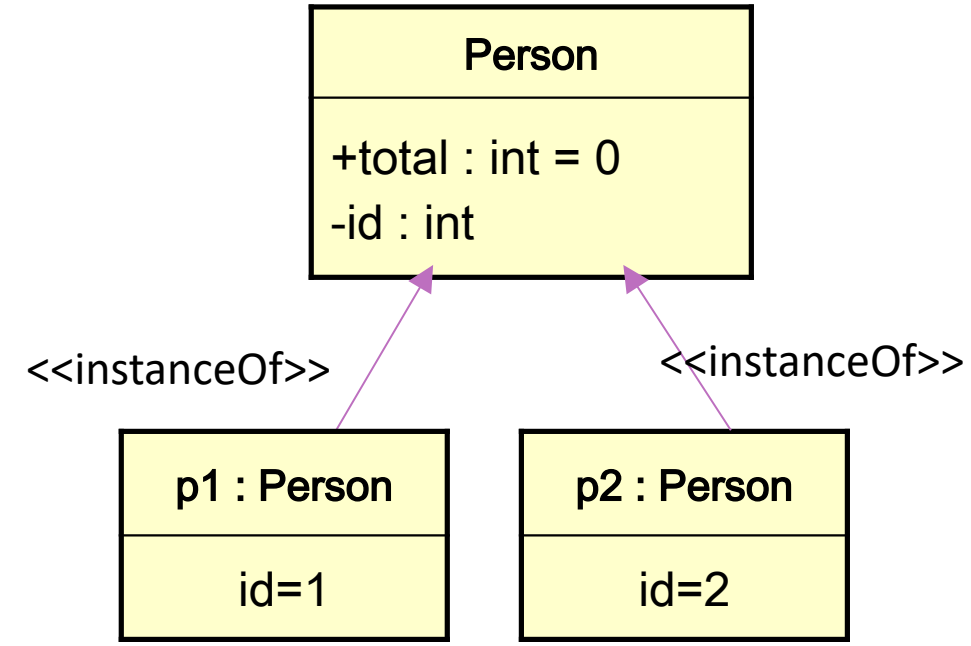


stati
c

类变量 (class Variable)

static

- 类变量（类属性）由该类的所有实例共享




```
public class Person {
    private int id;
    public static int total = 0;
    public Person() {
        total++;
        id = total;
    }
}
```

类变量应用举例

```
class Person {
        private int id;
        public static int total = 0;
        public Person() {
                total++;
                id = total;
        }
        public static void main(String args[]){
            Person Tom=new Person();
            Tom.id=0;
            total=100; //  不用创建对象就可以访问静态成员
       }
}
```

```
public class OtherClass {// 测试类
        public static void main(String args[]) {
                Person.total = 100;  //  不用创建对象就可以访问静态成员
                 // 访问方式：类名 . 类属性，类名 . 类方法
                System.out.println(Person.total); // 输出 100
                Person c = new Person();
                System.out.println(c.total);      // 输出 101
        }}
```

stati
c

类方法 (class Method)

- 没有对象的实例时，可以用**类名 . 方法名 ()** 的形式访问由 static 标记的类方法。
- **在 static 方法内部只能访问类的 static 属性，不能访问类的非 static 属性。**

static 方法

```
class Person {
    private int id;
    private static int total = 0;
    public static int getTotalPerson() {
        id++;      // 非法
        return total;
    }
    public Person() {
        total++;
        id = total;
    }}
public class TestPerson {// 测试类
    public static void main(String[] args) {
        System.out.println("Number of total is " +Person.getTotalPerson());
        // 没有创建对象也可以访问静态方法
        Person p1 = new Person();
        System.out.println( "Number of total is "+ Person.getTotalPerson());
    }}
```

The output is:
Number of total is 0
Number of total is 1

类方法

- 因为不需要实例就可以访问 static 方法，因此 static 方法内部不能有 this 。（也不能有 super）
- 重载的方法需要同时为 static 的或者非 static 的。

```
class Person {
    private int id;
    private static int total = 0;
    public static void setTotalPerson(int total){
        this.total=total;   // 非法，在 static 方法中不能有 this ，也不能有 super
    }
    public Person() {
        total++;
        id = total;
    }}
public class TestPerson {
    public static void main(String[] args) {
        Person.setTotalPerson(3);
    } }
```

推荐练习

编写一个类实现银行账户的概念，包含的属性有“帐号”、“密码”、“存款余额”、“利率”、“最小余额”，定义封装这些属性的方法。账号要自动生成。

编写主类，使用银行账户类，输入、输出 3 个储户的上述信息。

考虑：哪些属性可以设计成 static 属性。

单例（Singleton）设计模式

Singleton 模式

类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。如果要让类在一个虚拟机中只能产生一个对象，必须将**（1）类的构造方法的访问权限设置为 private**，这样，就不能用 new 操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能**（2）调用该类的某个静态方法以返回类内部创建的对象**，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的**（3）该类对象的变量也必须定义成静态的**。

常见应用场景（某个类只能存在一个对象实例）：线程池、缓存、日志、配置文件、打印机 / 显卡等硬件设备的驱动程序对象等。

单例 (Singleton) 设计模式 - 饿汉式

Singleton 模式

```
class Single{
  //private 的构造器，不能在类的外部创建该类的对象
   private Single() {} //(1) 类的构造方法的访问权限设置为 private
   // 私有的，只能在类的内部访问
   private static Single onlyone = new Single();//(3) 该类对象的变量定义成静态的
  //getSingle() 为 static ，不用创建对象即可访问
   public static Single getSingle() {//(2a) 该类的某个静态方法返回类内部创建的对象
          return onlyone;
  }
}
 public class TestSingle{
  public static void main(String args[]) {
          Single  s1 = Single.getSingle(); //(2b) 调用该类的某个静态方法以返回类内部创建的对象
          Single  s2 = Single.getSingle();
          if (s1==s2){
                  System.out.println("s1 is equals to s2!");
          }}}
```

单例 (Singleton) 设计模式 - 懒汉式

Singleton 模式

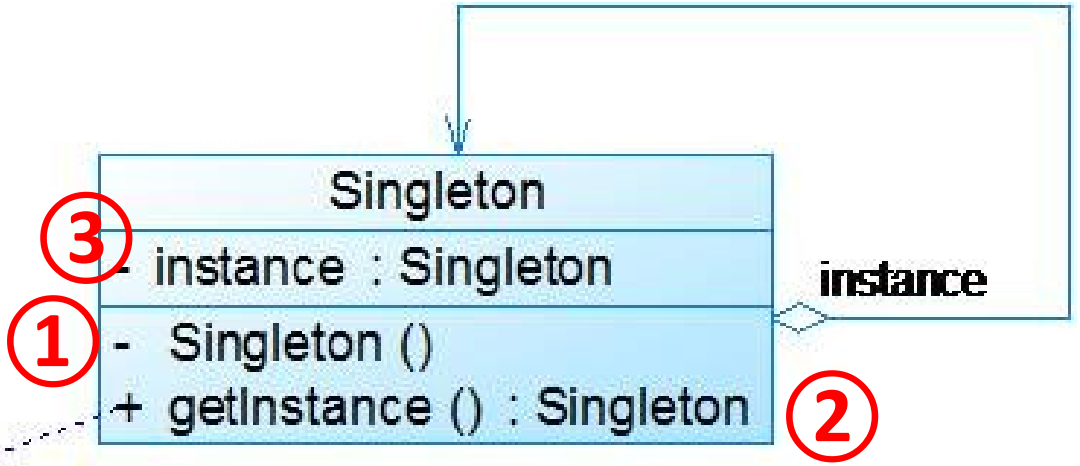
```
class Singleton{
  //1. 将构造器私有化，保证在此类的外部，不能调用本类的构造器。
  private Singleton(){//(1) 类的构造方法的访问权限设置为 private
  }
  //2. 先声明类的引用
  //4. 也需要配合 static 的方法，用 static 修饰此类的引用。
  private static Singleton  instance = null;  //(3) 该类对象的变量定义成静态的
  //3. 设置公共的方法来访问类的实例
  public static Singleton  getInstance(){//(2a) 某个静态方法返回类内部创建的对象
  //3.1 如果类的实例未创建，那些先要创建，然后返回给调用者：本类。因此，需要 static 修饰。
          if(instance == null){
                  instance = new Singleton();  //(3) 该类对象变量定义成静态的
          }
  //3.2  若有了类的实例，直接返回给调用者。
          return instance;  //(2b) 某个静态方法返回类内部创建的对象
  } }
```

举例： java.lang.Runtime

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    /**
     * Returns the runtime object associated with the current Java application.
     * Most of the methods of class <code>Runtime</code> are instance
     * methods and must be invoked with respect to the current runtime object.
     *
     * @return  the <code>Runtime</code> object associated with the current
     *          Java application.
     */
    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}

    /**
```

③ ② ①

单例模式（ Singleton Pattern ）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。这种模式涉及到一个单一的类，**该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象**。



懒汉式

```
if(instance==null)
    instance=new Singleton();
return instance;
```

Singleton 模式

单例模式：

1、保证一个类仅有一个实例，并提供一个访问它的全局访问点；

2、当一个全局使用的类被频繁的创建和销毁；或者你要控制实例的数目，节省系统资源的时候，可以考虑使用单例模式；

3、实现：判断系统是否存在这个单例，如果存在则返回，否则，创建；

4、构造函数私有；

注意：

1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

第 7 讲 Java OOP- 设计模式

7.1 设计模式

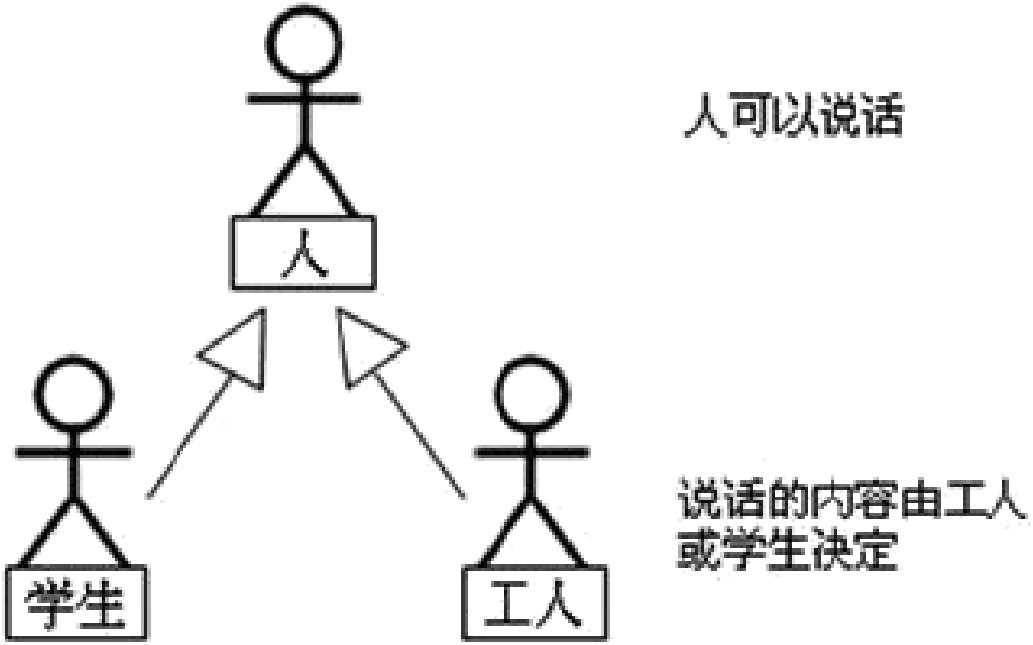
7.2 Static 、类变量 / 方法与单例设计模式 Singleton

7.3 抽象类与模板方法设计模式 TemplateMethod

7.4 接口与简单工厂模式

抽象类 (abstract class)

- 随着继承层次中一个个新子类的定义，类变得越来越具体，而父类则更一般，更通用。类的设计应该保证父类和子类能够共享特征。有时**将一个父类设计得非常抽象，以至于它没有具体的实例，这样的类叫做抽象类。**



抽象类

- 用 abstract 关键字来修饰一个类时，这个类叫做抽象类；
- 用 abstract 来修饰一个方法时，该方法叫做抽象方法。
  - 抽象方法：只有方法的声明，没有方法的实现。以分号结束：

```
abstract int abstractMethod( int a );
```

- **含有抽象方法的类必须被声明为抽象类。**
- 抽象类不能被实例化。抽象类是用来被继承的，**抽象类的子类必须重写父类的抽象方法，并提供方法体。若没有重写全部的抽象方法，仍为抽象类**。
- 不能用 abstract 修饰属性、私有方法、构造器、静态方法、 final 的方法。

抽象类举例

```
abstract class A{
        abstract void m1( );
        public void m2( ){
                System.out.println("A 类中定义的 m2 方法 ");
        }
}

class B extends A{
        void m1( ){
                System.out.println("B 类中定义的 m1 方法 ");
        }
}

public class Test{
        public static void main( String args[ ] ){
                A a = new B( );
                a.m1( );
                a.m2( );
        }
}
```

Outline | Circle.java | TestStatic.java | A.java | B.java

```java
//A.java
package ch004;

abstract class A{
    abstract void m1( );
    public void m2( ){
        System.out.println("A类中定义的m2方法");
    }
}
```

Outline | Circle.java | TestStatic.java | A.java | B.java

```java
//B.java
package ch004;

class B extends A{
    void m1( ){
        System.out.println("B类中定义的m1方法");
    }
}
```

Outline | Circle.java | TestStatic.java | A.java

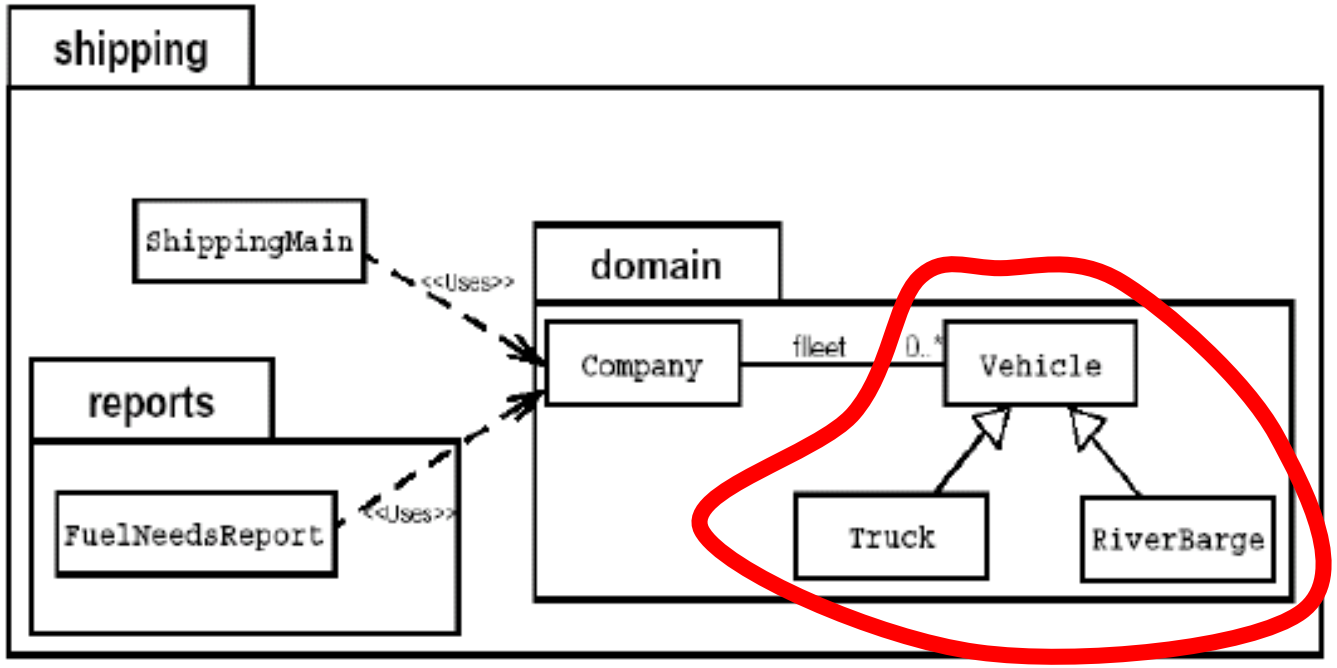
```java
//TestAB.java
package ch004;

public class TestAB{
    public static void main( String args[ ] ){
        A a = new B( );
        a.m1( );
        a.m2( );
    }
}
```

Problems | Console | Debug

```
<terminated> TestAB [Java Application] C:\Program Files\Java\
B类中定义的m1方法
A类中定义的m2方法
```

抽象类应用

抽象类是用来模型化那些父类无法确定全部实现，而是由其子类提供具体实现的对象的类。



在航运公司系统中，Vehicle 类需要定义两个方法分别计算运输工具的燃料效率和行驶距离。

问题：卡车 (Truck) 和驳船 (RiverBarge) 的燃料效率和行驶距离的计算方法完全不同。 Vehicle 类不能提供计算方法，但子类可以。

抽象类应用

- **解决方案**

  Java 允许类设计者指定：超类声明一个方法但不提供实现，该方法的实现由子类提供。这样的方法称为抽象方法。有一个或更多抽象方法的类称为抽象类。

- **Vehicle 是一个抽象类，有两个抽象方法。**

```
public abstract class Vehicle{
   public abstract double calcFuelEfficiency();          // 计算燃料效率的抽象方法
   public abstract double calcTripDistance();            // 计算行驶距离的抽象方法
}
public class Truck extends Vehicle{
   public double calcFuelEfficiency( ) { // 写出计算卡车的燃料效率的具体方法 }
   public double calcTripDistance( ) { // 写出计算卡车行驶距离的具体方法 }
}

public class RiverBarge extends Vehicle{
    public double calcFuelEfficiency( ) { // 写出计算驳船的燃料效率的具体方法 }
    public double calcTripDistance( ) { // 写出计算驳船行驶距离的具体方法 }
}
```

注意：抽象类不能实例化　new Vihicle() 是非法的

思 考

问题 1 ：为什么抽象类不可以使用 final 关键字声明？

问题 2 ：一个抽象类中可以定义构造器吗？

推荐练习

编写一个 Employee 类，声明为抽象类，包含如下三个属性： name ， id ， salary 。提供必要的构造器和抽象方法： work() 。对于 Manager 类来说，他既是员工，还具有奖金 (bonus) 的属性。请使用继承的思想，设计 CommonEmployee 类和 Manager 类，要求类中提供必要的方法进行属性访问。

模板方法设计模式 (TemplateMethod)

抽象类体现的就是一种模板模式的设计，**抽象类作为多个子类的通用模板**，子类在抽象类的基础上进行扩展、改造，但子类总体上会保留抽象类的行为方式。

解决的问题：

- **当功能内部一部分实现是确定，一部分实现是不确定的。这时可以把不确定的部分暴露出去，让子类去实现。**
- **编写一个抽象父类，父类提供了多个子类的通用方法，并把一个或多个方法留给其子类实现，就是一种模板模式。**

模板方法设计模式 (TemplateMethod)

模板模式

```
abstract class Template{
        public final void getTime(){
                long start = System.currentTimeMillis();
                code();
                long end = System.currentTimeMillis();
                System.out.println(" 执行时间是： "+(end - start));
        }
        public abstract void code();
}

class SubTemplate extends Template{
        public void code(){
                for(int i = 0;i<10000;i++){
                System.out.println(i);
} } }
```

第 7 讲 Java OOP- 设计模式

7.1 设计模式

7.2 Static 、类变量 / 方法与单例设计模式 Singleton

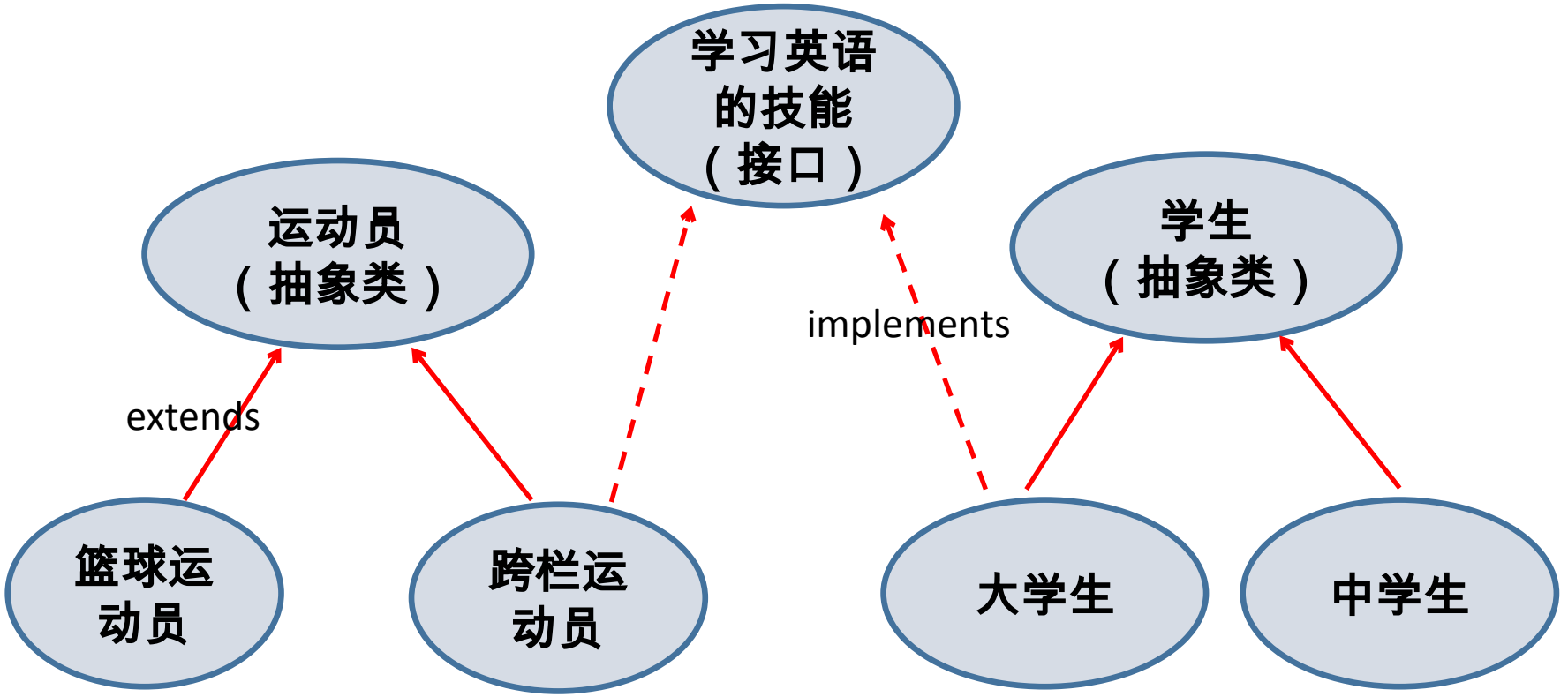
7.3 抽象类与模板方法设计模式 TemplateMethod

7.4 接口与简单工厂模式

接 口 (1)

- 有时必须从几个类中派生出一个子类，继承它们所有的属性和方法。但是， Java 不支持多重继承。**有了接口，就可以得到多重继承的效果。**
- **接口 (interface) 是抽象方法和常量值的定义的集合**。
- 从本质上讲，**接口是一种特殊的抽象类，这种抽象类中只包含常量和方法的定义，而没有变量和方法的实现**。
- 实现接口类：
  - class SubClass **implements** InterfaceA{ }
- 一个类可以实现多个接口，接口也可以继承其它接口。

接 口 (2)


学习英语
的技能
（接口）
运动员
（抽象类）
学生
（抽象类）
implements
extends
篮球运
动员
跨栏运
动员
大学生
中学生

接口

接 口 (3)

- **接口的特点：**
  - 用 interface 来定义。
  - 接口中的所有成员变量都默认是由 public static final 修饰的。
  - 接口中的所有方法都默认是由 public abstract 修饰的。
  - 接口没有构造器。
  - 接口采用多继承机制。
- 接口定义举例

```
public interface Runner {
        int ID = 1;
        void start();
        public void run();
        void stop();
}
```

```
public interface Runner {
  public static final int ID = 1;
  public abstract void start();
  public abstract void run();
  public abstract void stop();
  }
```

接口

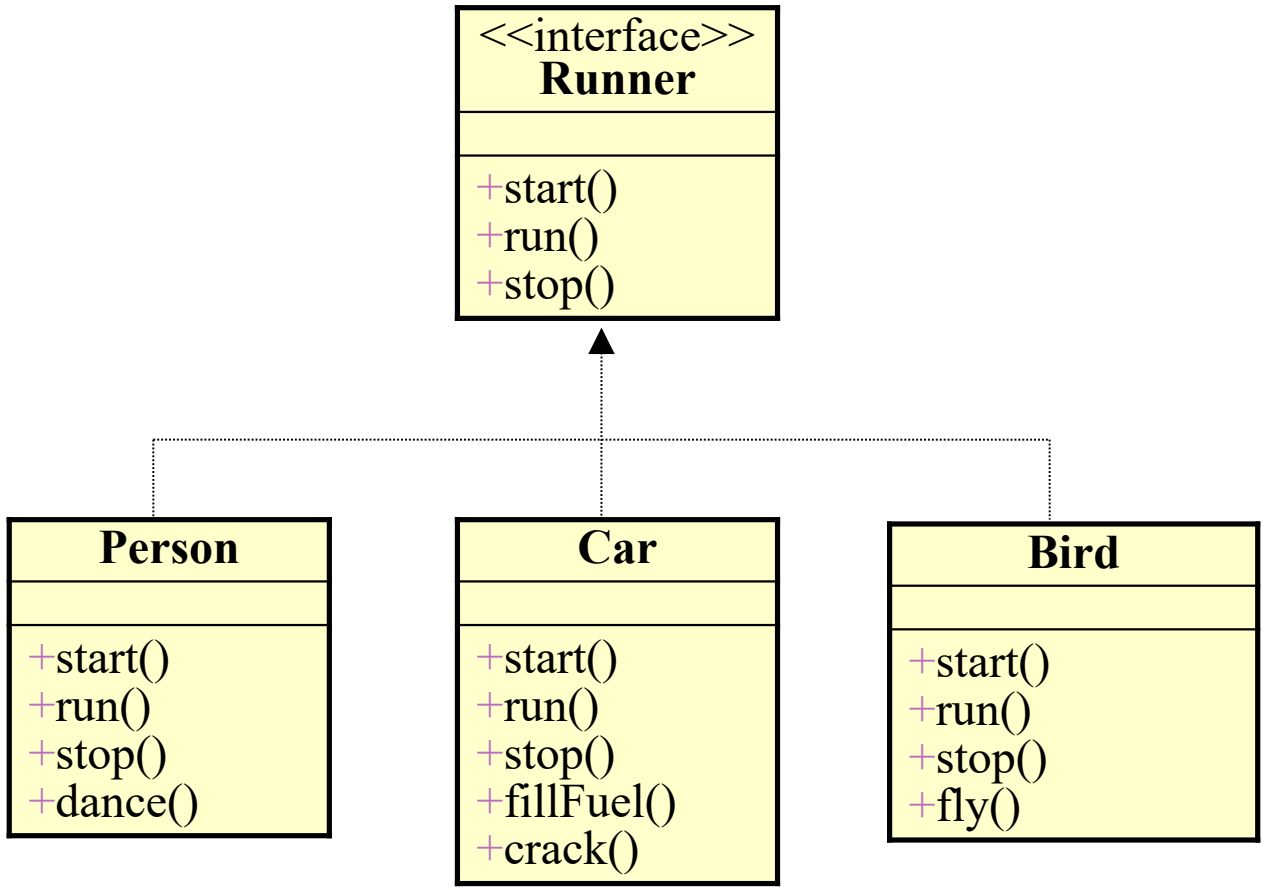
接 口 (4)

- 实现接口的类中必须提供接口中所有方法的具体实现内容，方可实例化。否则，仍为抽象类。
- **接口的主要用途就是被实现类实现。（面向接口编程）**
- 与继承关系类似，接口与实现类之间存在多态性
- 定义 Java 类的语法格式：先写 extends ，后写 implements

接口

```
< modifier> class < name> [extends < superclass>]
[implements < interface> [,< interface>]* ] {
        < declarations>*
}
```

接口应用举例 (1)


<<interface>>
Runner
+start()
+run()
+stop()
Person
+start()
+run()
+stop()
+dance()
Car
+start()
+run()
+stop()
+fillFuel()
+crack()
Bird
+start()
+run()
+stop()
+fly()

接口

接口应用举例 (1)

```
public interface Runner {
        public void start();
        public void run();
        public void stop();
}
public class Person implements Runner {
        public void start() {
                // 准备工作：弯腰、蹬腿、咬牙、瞪眼
// 开跑
        }
        public void run() {
                // 摆动手臂
                // 维持直线方向
        }
        public void stop() {
                // 减速直至停止、喝水。
        }
}
```

接口

接口应用举例(2)

- 一个类可以实现多个无关的接口

```
interface Runner { public void run();}

interface Swimmer {public double swim();}

class Creator{public int eat(){…}}

class Man extends Creator implements Runner ,Swimmer, Creator{
                public void run() {……}
                public double swim()  {……}
                public int eat() {……}
}
```

- **与继承关系类似，接口与实现类之间存在多态性**

```
public class Test{
   public static void main(String args[]){
                Test t = new Test();
                Man m = new Man();
                t.m1(m);
                t.m2(m);
                t.m3(m);
   }
   public String m1(Runner f) { f.run(); }
   public void  m2(Swimmer s) {s.swim();}
   public void  m3(Creator a) {a.eat();}
}
```

接口的其他问题

- **如果实现接口的类中没有实现接口中的全部方法，必须将此类定义为抽象类**
- 接口也可以继承另一个接口，使用 extends 关键字。
- interface MyInterface{
  String s="MyInterface";
  public void absM1();
  }
  interface SubInterface extends MyInterface{
  public void absM2();
  }
  public class SubAdapter implements SubInterface{
  public void absM1(){System.out.println("absM1");}
  public void absM2(){System.out.println("absM2");}
  }

实现类 SubAdapter 必须给出接口 SubInterface 以及父接口 MyInterface 中所有方法的实现。

工厂方法 (FactoryMethod)

概述：
定义一个用于创建对象的接口，让子类决定实例化哪一个类。FactoryMethod 使一个类的实例化延迟到其子类。

适用性：
1. 当一个类不知道它所必须创建的对象的类的时候
2. 当一个类希望由它的子类来指定它所创建的对象的时候
3. 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候

工厂模式

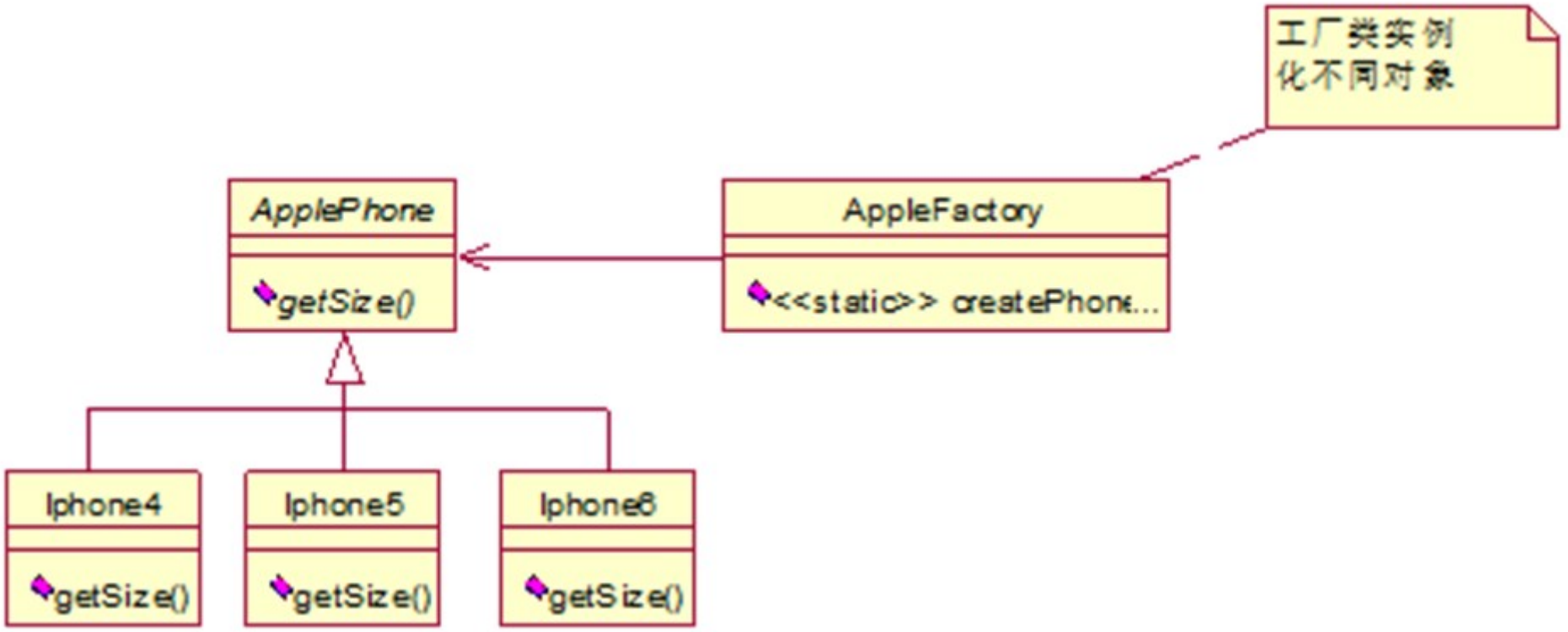
工厂方法 (FactoryMethod)

总结：

FactoryMethod 模式是设计模式中应用最为广泛的模式，**在面向对象的编程中，对象的创建工作非常简单，对象的创建时机却很重要。FactoryMethod 解决的就是这个问题**，它通过面向对象的手法，**将所要创建的具体对象的创建工作延迟到了子类**，从而提供了一种扩展的策略，较好的解决了这种紧耦合的关系。

工厂模式

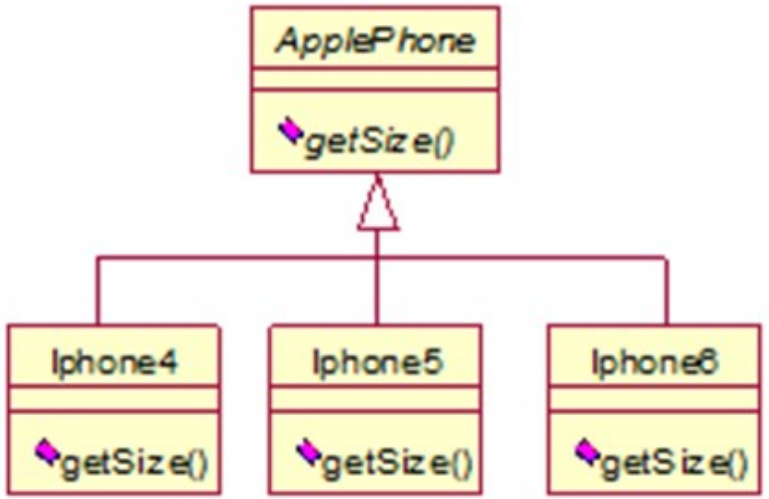
工厂方法举例



工厂模式

1. 由工厂决定生产哪种型号的的手机，苹果公司的工厂就是一个工厂类，是简单工厂模式的核心类。
2. iPhone5 、 iPhone5S 、 iphone6 都是苹果手机，只是型号不同。苹果手机类满足抽象的定义，各个型号的手机类是其具体实现。

```java
public abstract class ApplePhone {

    /**
     *  获取尺寸
     */
    protected abstract void getSize();
}
```




```java
public class Iphone4 extends ApplePhone{
    public void getSize() {
        System.out.println("iPhone4 屏幕 :3.5 英寸 ");
    }
}
```

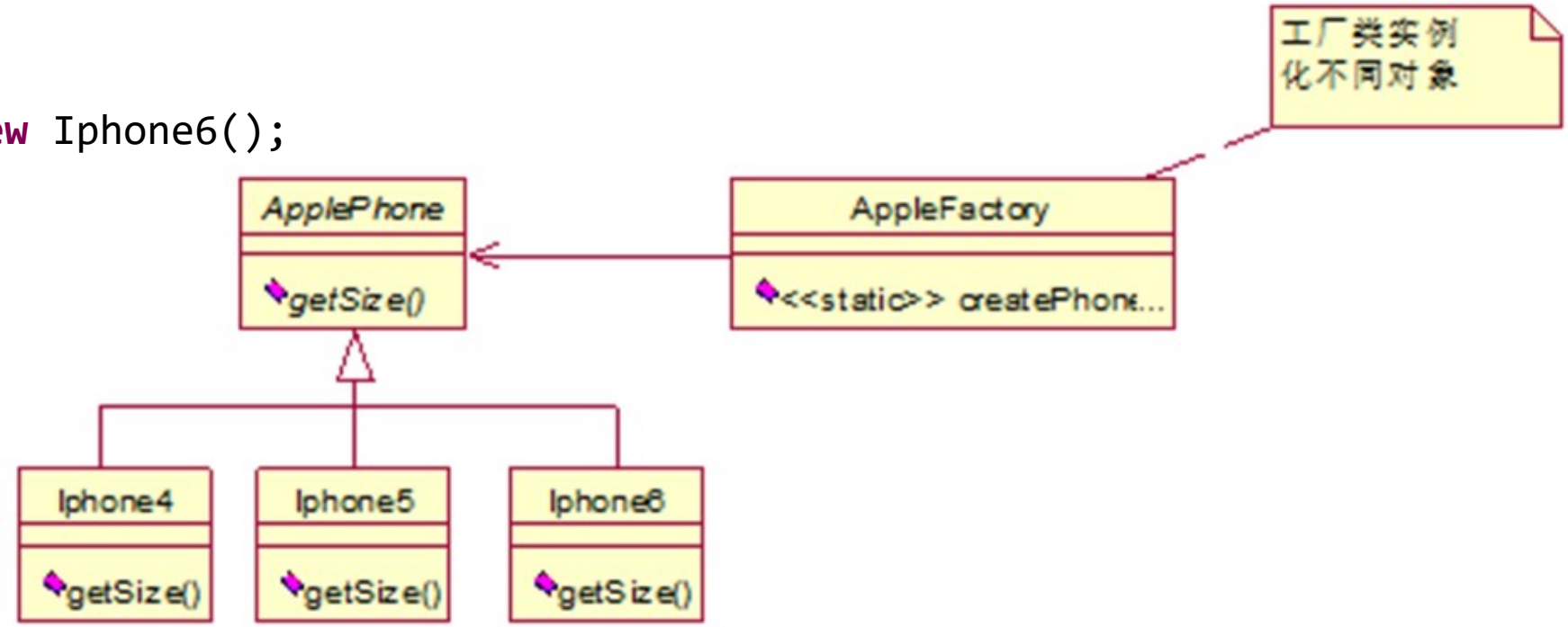
```java
public class Iphone5 extends ApplePhone{
    public void getSize() {
        System.out.println("iPhone5 屏幕 :4 英寸 ");
    }
}
```

工厂模式

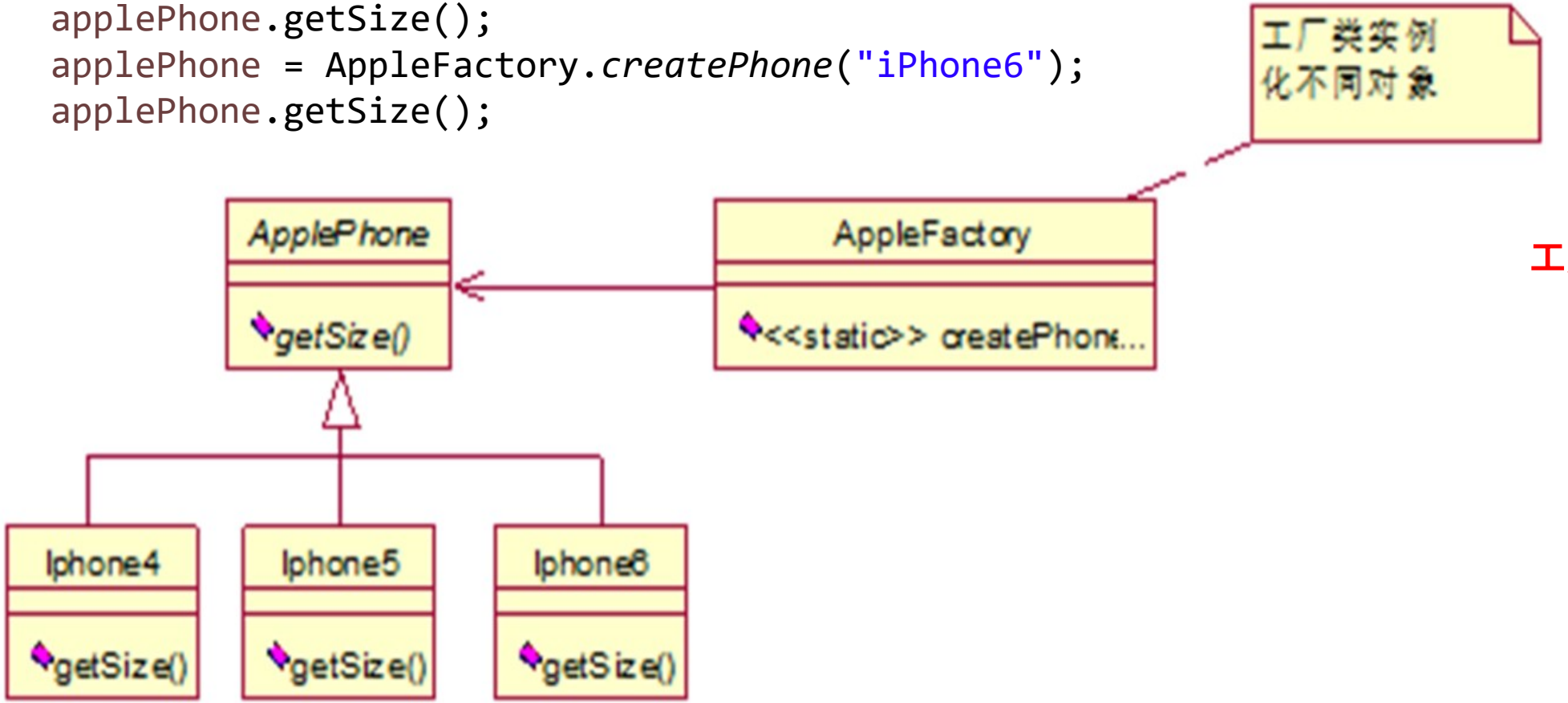
```java
public class Iphone6 extends ApplePhone{
    public void getSize() {
        System.out.println("iPhone6 屏幕 :4.7 英寸 ");
    }
}
```

```java
public class AppleFactory {

    public static ApplePhone createPhone(String model){
        ApplePhone applePhone = null;

        switch (model) {
        case"iPhone4":
            applePhone = new Iphone4();
            break;
        case"iPhone5":
            applePhone = new Iphone5();
            break;
        case"iPhone6":
            applePhone = new Iphone6();
            break;
        default:
            break;
        }

        return applePhone;
    }
}
```

工厂模式



```java
public class Test{

    public static void main(String[] args) {
        ApplePhone applePhone ;
        applePhone = AppleFactory.createPhone("iPhone4");
        applePhone.getSize();
        applePhone = AppleFactory.createPhone("iPhone5");
        applePhone.getSize();
        applePhone = AppleFactory.createPhone("iPhone6");
        applePhone.getSize();
    }
}
```



工厂模式

接口用法总结

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。
- 通过接口可以指明多个类需要实现的方法，一般用于定义对象的扩张功能。
- 接口主要用来定义规范。解除耦合关系。

接口

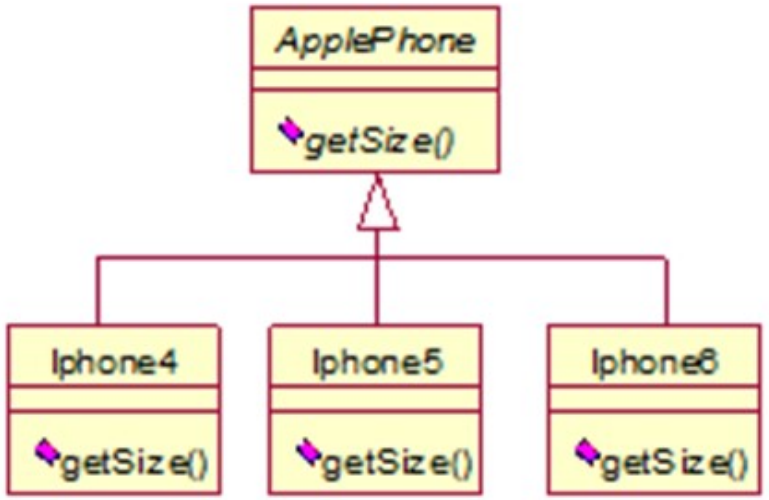
接口和抽象类之间的关系

No.	区别点	抽象类	接口
1	定义	包含一个抽象方法的类	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	常量、抽象方法
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，但允许继承多个接口
5	常见设计模式	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	作为一个模板	是作为一个标准或是表示一种能力
9	选择	如果抽象类和接口都可以使用的话，优先使用接口，因为避免单继承的局限	
10	特殊	一个抽象类中可以包含多个接口，一个接口中可以包含多个抽象类	

在开发中，一个类不要去继承一个已经实现好的类，要么继承抽象类，要么实现接口。

修改

```
public interface ApplePhone {

    /**
     *  获取尺寸
     */
    void getSize();
}
```




```
public class Iphone4 implements ApplePhone{
    public void getSize() {
        System.out.println("iPhone4 屏幕 :3.5 英
寸 ");
    }
}
```

```
public class Iphone5  implements  ApplePhone{
    public void getSize() {
        System.out.println("iPhone5 屏幕 :4 英寸 ");
    }
}
```

```
public class Iphone6  implements  ApplePhone{
    public void getSize() {
        System.out.println("iPhone6 屏幕 :4.7 英
寸 ");
    }
}
```

```
public interface IApplePhoneFactory {
    ApplePhone getApplePhone();
}

public class Iphone4Factory implements IApplePhoneFactory{
    public ApplePhone getApplePhone(){
        return new Iphone4();
    }
}

public class Iphone5Factory implements IApplePhoneFactory{
    public ApplePhone getApplePhone(){
        return new Iphone5();
    }
}

public class Iphone6Factory implements IApplePhoneFactory{
    public ApplePhone getApplePhone(){
        return new Iphone6();
    }
}
```

```
public void class AppleFactory {
    public static ApplePhone createPhone(String model){
        IAppleFactory factory = null;

        switch (model) {
        case“iPhone4":
            factory = new iPhone4Factory();
            ApplePhone iphone4 = factory.getApplePhone();
            iphone4.getSize();
            break;
        case"iPhone5":
            factory = new iPhone5Factory();
            ApplePhone iphone5 = factory.getApplePhone();
            iphone5.getSize();
            break;
        case"iPhone6":
            factory = new iPhone4Factory();
            ApplePhone iphone4 = factory.getApplePhone();
            iphone4.getSize();
            break;
        default:
            break;
        }
        return applePhone;
    }
}
```

推荐练习

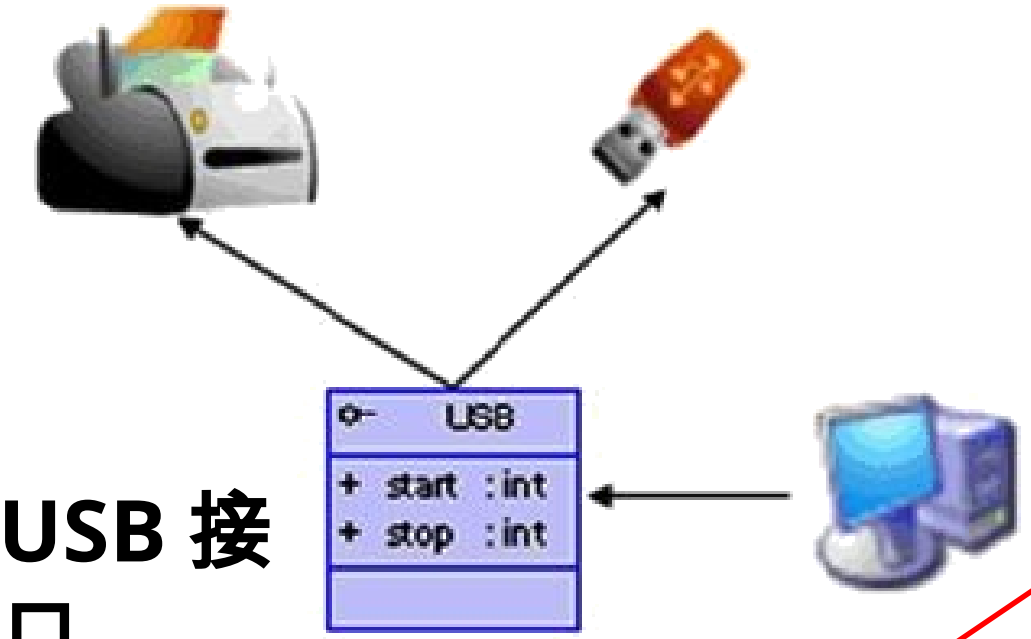
- 定义一个接口用来实现两个对象的比较。
  - interface CompareObject{

    public int compareTo(Object o); // 若返回值是 0，代表相等；若为正数，代表当前对象大；负数代表当前对象小

    }
- 定义一个 Circle 类。
- 定义一个 ComparableCircle 类，继承 Circle 类并且实现 CompareObject 接口。在 ComparableCircle 类中给出接口中方法 compareTo 的实现体，用来比较两个圆的半径大小。
- 定义一个测试类 TestInterface ，创建两个 ComparableCircle 对象，调用 compareTo 方法比较两个类的半径大小。
- 思考：参照上述做法定义矩形类 Rectangle 和 ComparableRectangle 类，在 ComparableRectangle 类中给出 compareTo 方法的实现，比较两个矩形的面积大小。

接口的应用体会

USB 接口



JDBC 接口

面向接口编程的思想

面向对象特性，是 java 学习的核心，及时梳理、总结