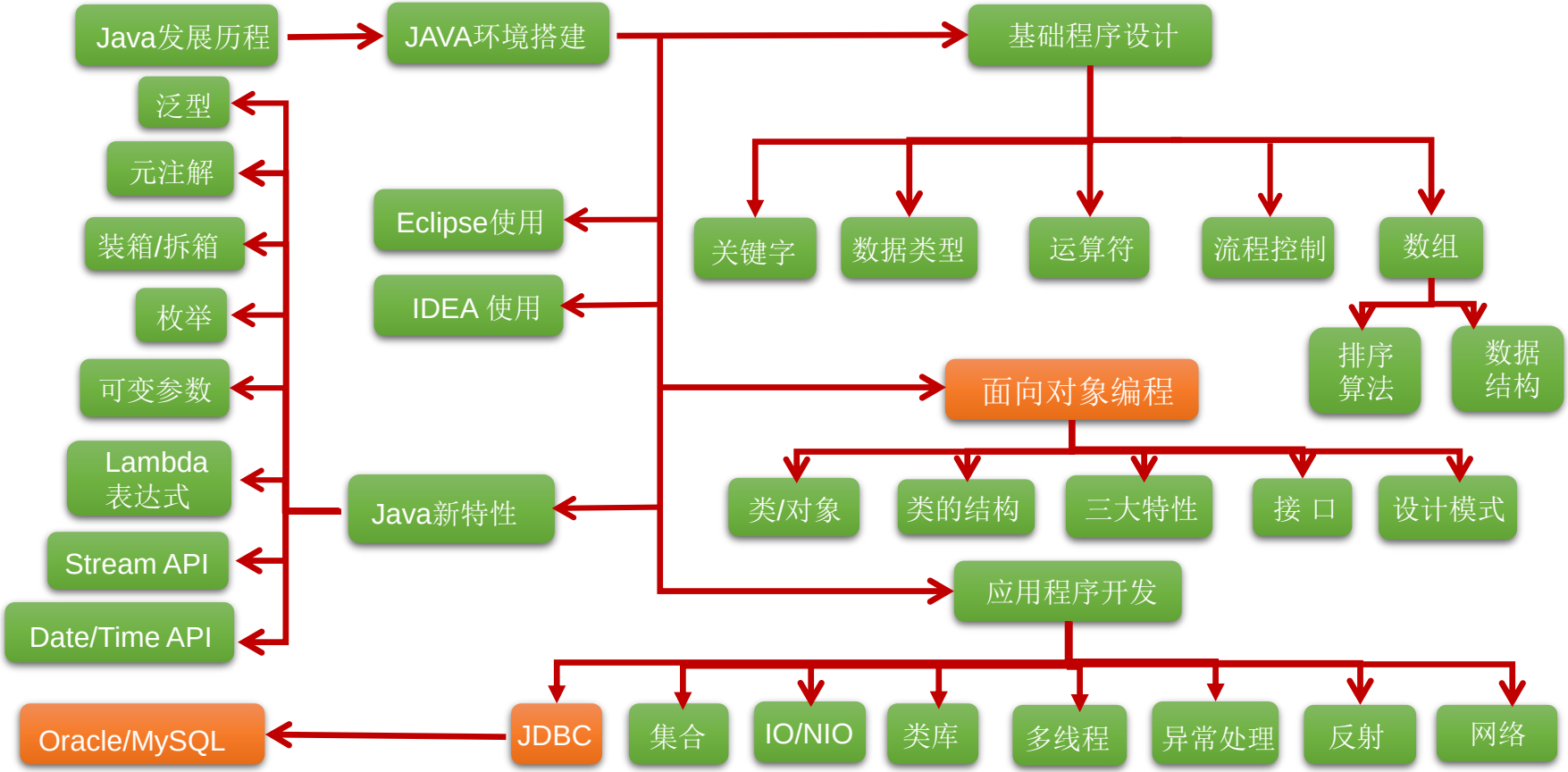
第6章
面向对象编程(下)

讲师：宋红康
新浪微博：尚硅谷-宋红康

Java基础知识图解

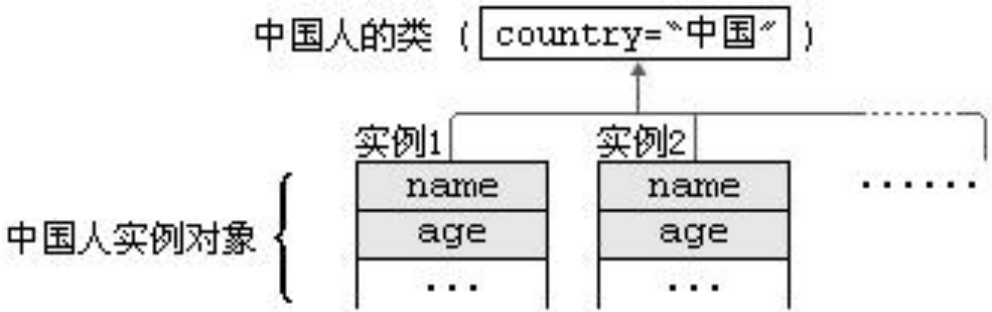


目录

6-1 关键字：static

6.1 关键字：static

当我们编写一个类时，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。我们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表国家名称的变量。



- 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

- 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。

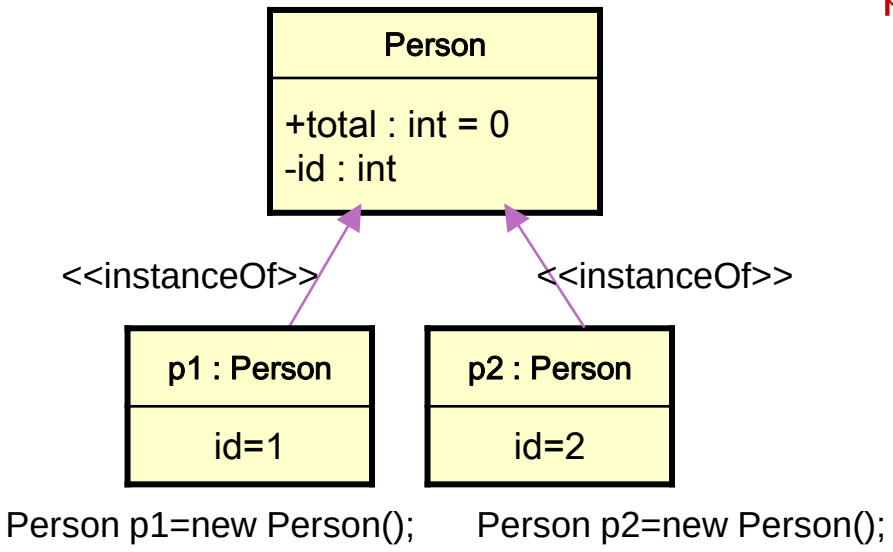
- 使用范围：
  - 在Java类中，可用static修饰属性、方法、代码块、内部类
- 被修饰后的成员具备以下特点：
  - 随着类的加载而加载
  - 优先于对象存在
  - 修饰的成员，被所有对象所共享
  - 访问权限允许时，可不创建对象，直接被类调用

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

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

类变量(class Variable)

- 类变量（类属性）由该类的所有实例共享



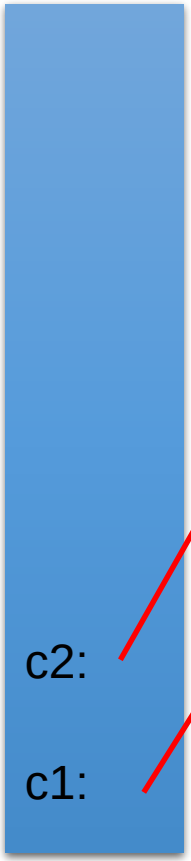

```
public class Person {
    private int id;
    public static int total = 0;
    public Person() {
        total++;
        id = total;
    }
}
```

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 StaticDemo {
        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);
                Person c = new Person();
                System.out.println(c.total); //输出101
        }}
```

类变量 vs 实例变量内存解析



栈：局部变量

堆：new出来的结构：对象、数组

name:马龙
age:30

name:姚明
age:40

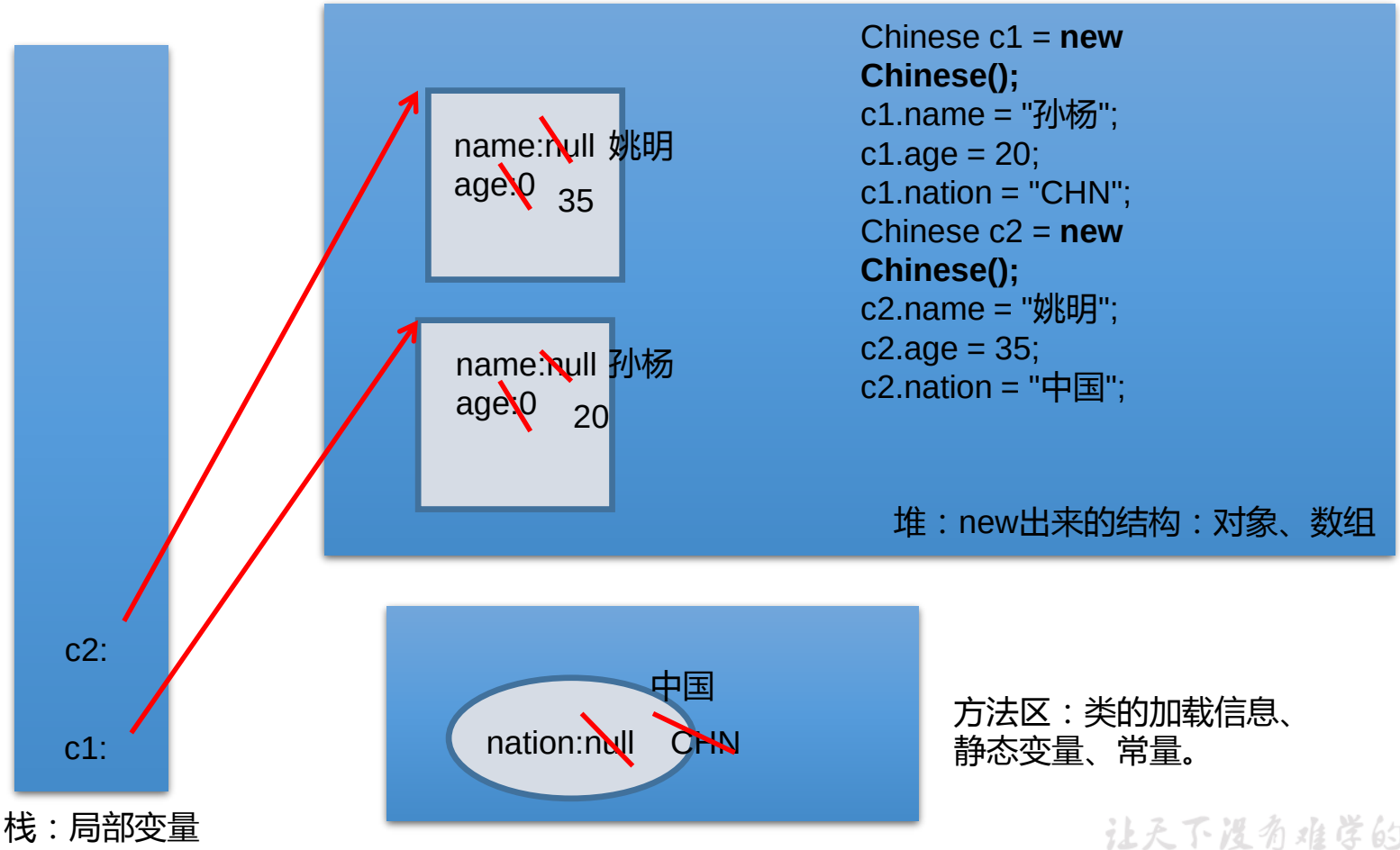
```
Chinese.nation = "中国";
Chinese c1 = new Chinese();
c1.name = "姚明";
c1.age = 40;
Chinese c2 = new Chinese();
c2.name = "马龙";
c2.age = 30;
c1.nation = "CHN";
c2.nation = "CHINA";
```

```
class Chinese{
String name;
int age;
static String nation;
}
```

nation:null 中国 CHINA CHN

方法区：类的加载信息、静态域、常量池

静态变量的内存解析



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修饰的类方法。
- **在static方法内部只能访问类的static修饰的属性或方法，不能访问类的非static的结构。**

```
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 PersonTest {
    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(class method)

- 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 ? YES!)
- static修饰的方法不能被重写

```
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 PersonTest {
         public static void main(String[] args) {
                Person.setTotalPerson(3);
         } }
```

练习1

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

- 设计模式是在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及解决问题的思考方式。设计模免去我们自己再思考和摸索。就像是经典的棋谱，不同的棋局，我们用不同的棋谱。”套路”

- 所谓类的单例设计模式，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。如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的构造器的访问权限设置为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，指向类内部产生的该类对象的变量也必须定义成静态的。

```
class Singleton {
    // 1.私有化构造器
    private Singleton() {
    }

    // 2.内部提供一个当前类的实例
    // 4.此实例也必须静态化
    private static Singleton single = new Singleton();

    // 3.提供公共的静态的方法，返回当前类的对象
    public static Singleton getInstance() {
        return single;
    }
}
```

单例(Singleton)设计模式-懒汉式

```
class Singleton {
    // 1.私有化构造器
    private Singleton() {
    }
    // 2.内部提供一个当前类的实例
    // 4.此实例也必须静态化
    private static Singleton single;
    // 3.提供公共的静态的方法，返回当前类的对象
    public static Singleton getInstance() {
        if(single == null) {
            single = new Singleton();
        }
        return single;
    }
}
```

懒汉式暂时还存在线程安全问题，讲到多线程时，可修复

单例(Singleton)设计模式

- **单例模式的优点：**

由于单例模式只生成一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。

- **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() {}

    /**
```

单例(Singleton)设计模式-应用场景

- 网站的计数器，一般也是单例模式实现，否则难以同步。
- 应用程序的日志应用，一般都使用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。
- 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。
- 项目中，读取配置文件的类，一般也只有一个对象。没有必要每次使用配置文件数据，都生成一个对象去读取。
- Application 也是单例的典型应用
- Windows的Task Manager (任务管理器)就是很典型的单例模式
- Windows的Recycle Bin (回收站)也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

6-2 理解main方法的语法

- 由于Java虚拟机需要调用类的main()方法，所以该方法的访问权限必须是public，又因为Java虚拟机在执行main()方法时不必创建对象，所以该方法必须是static的，该方法接收一个String类型的数组参数，该数组中保存执行Java命令时传递给所运行的类的参数。

- 又因为main() 方法是静态的，我们不能直接访问该类中的非静态成员，必须创建该类的一个实例对象后，才能通过这个对象去访问类中的非静态成员，这种情况，我们在之前的例子中多次碰到。

命令行参数用法举例

```
public class CommandPara {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println("args[" + i + "] = " + args[i]);
        }
    }
}
```

//运行程序CommandPara.java

java CommandPara “Tom" “Jerry" “Shkstart"

输出结果：

args[0] = Tom
args[1] = Jerry
args[2] = Shkstart

【面试题】

此处，Something类的文件名叫OtherThing.java

```
class Something {
    public static void main(String[] something_to_do) {
        System.out.println("Do something ...");
    }
}
```

形参随便取

默认访问权限也可以

上述程序是否可以正常编译、运行？

6-3 类的成员之四：代码块

- 代码块(或初始化块)的作用：
  - **对Java类或对象进行初始化**
- 代码块(或初始化块)的分类：
  - 一个类中代码块若有修饰符，则只能被static修饰，称为**静态代码块**(static block)，没有使用static修饰的，为**非静态代码块。**
- **static代码块通常用于初始化static的属性**

```
class Person {
    public static int total;
    static {
        total = 100;//为total赋初值
    }
    …… //其它属性或方法声明
}
```

- **静态代码块：用static 修饰的代码块**

1. 可以有输出语句。
2. 可以对类的属性、类的声明进行初始化操作。
3. 不可以对非静态的属性初始化。即：不可以调用非静态的属性和方法。
4. 若有多个静态的代码块，那么按照从上到下的顺序依次执行。
5. 静态代码块的执行要先于非静态代码块。
6. 静态代码块随着类的加载而加载，且只执行一次。

- **非静态代码块：没有static修饰的代码块**

1. 可以有输出语句。
2. 可以对类的属性、类的声明进行初始化操作。
3. 除了调用非静态的结构外，还可以调用静态的变量或方法。
4. 若有多个非静态的代码块，那么按照从上到下的顺序依次执行。
5. 每次创建对象的时候，都会执行一次。且先于构造器执行。

静态初始化块举例

```
class Person {
    public static int total;
    static {
        total = 100;
        System.out.println("in static block!");
    }
}

public class PersonTest {
    public static void main(String[] args) {
        System.out.println("total = " + Person.total);
        System.out.println("total = " + Person.total);
    }
}
```

输出：
in static block
total=100
total=100

举例二：LeafTest.java
举例三：Son.java

总结：程序中成员变量赋值的执行顺序

声明成员变量的默认初始化

↓

显式初始化、多个初始化块依次被执行（同级别下按先后顺序执行）

↓

构造器再对成员进行初始化操作

↓

通过”对象.属性”或”对象.方法”的方式，可多次给属性赋值

6-4 关键字：final

6.4 关键字：final

- 在Java中声明类、变量和方法时，可使用关键字final来修饰,表示“最终的”。
  - **final标记的类不能被继承。**提高安全性，提高程序的可读性。
    - ✓String类、System类、StringBuffer类
  - **final标记的方法不能被子类重写。**
    - ✓比如：Object类中的getClass()。
  - **final标记的变量(成员变量或局部变量)即称为常量。**名称大写，且只能被赋值一次。
    - ✓final标记的成员变量必须在声明时或在每个构造器中或代码块中显式赋值，然后才能使用。
    - ✓final double MY_PI = 3.14;

1. final修饰类

```
final class A{
}

class B extends A{        //错误，不能被继承。
}
```

中国古代，什么人不能有后代，就可以被final声明，称为“太监类”！

2. final修饰方法

```
class A {
    public final void print() {
        System.out.println("A");
    }
}

class B extends A {
    public void print() { // 错误，不能被重写。
        System.out.println("尚硅谷");
    }
}
```

3. final修饰变量——常量

```
class A {
    private final String INFO = "atguigu";  //声明常量

    public void print() {
        //The final field A.INFO cannot be  assigned
        //INFO = "尚硅谷";
    }
}
```

常量名要大写，内容不可修改。——如同古代皇帝的圣旨。

- static final：全局常量

关键字final应用举例

```
public final class Test {
    public static int totalNumber = 5;
    public final int ID;

    public Test() {
        ID = ++totalNumber; // 可在构造器中给final修饰的“变量”赋值
    }
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println(t.ID);
        final int I = 10;
        final int J;
        J = 20;
        J = 30; // 非法
    }
}
```

【面试题】排错：

题目一

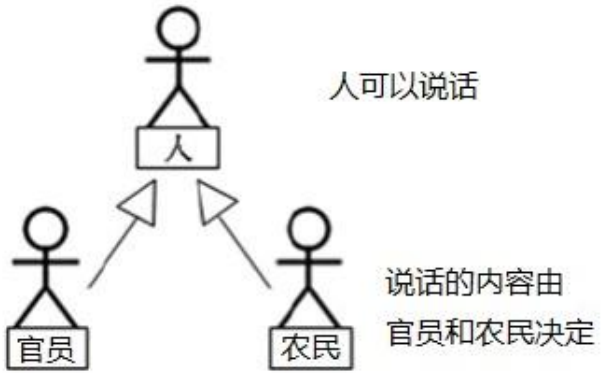
```
public class Something {
    public int addOne(final int x) {
        return ++x;
        // return x + 1;
    }
}
```

题目二

```
public class Something {
    public static void main(String[] args) {
        Other o = new Other();
        new Something().addOne(o);
    }
    public void addOne(final Other o) {
        // o = new Other();
        o.i++;
    }
}
class Other {
    public int i;
}
```

6-5 抽象类与抽象方法

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。



6.5 抽象类与抽象方法

- 用abstract关键字来修饰一个类，这个类叫做抽象类。
- 用abstract来修饰一个方法，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：

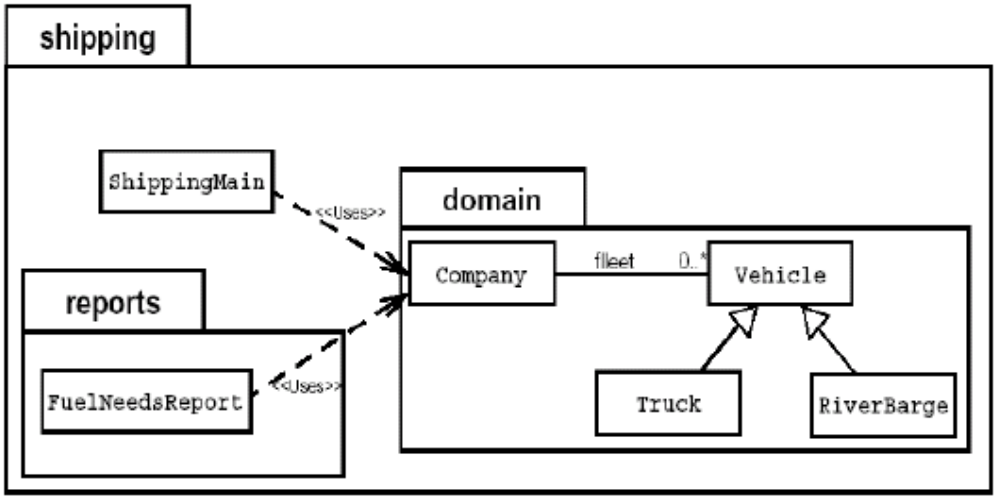
    比如：public abstract void talk();
- 含有抽象方法的类必须被声明为抽象类。
- 抽象类不能被实例化。抽象类是用来被继承的，抽象类的子类必须重写父类的抽象方法，并提供方法体。若没有重写全部的抽象方法，仍为抽象类。
- 不能用abstract修饰变量、代码块、构造器；
- 不能用abstract修饰私有方法、静态方法、final的方法、final的类。

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();
    }
}
```

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？

问题3：是否可以这样理解：抽象类就是比普通类多定义了抽象方法，除了不能直接进行类的实例化操作之外，并没有任何的不同？

练 习2

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。

多态的应用：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的，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。
- 换句话说，在软件开发中实现一个算法时，整体步骤很固定、通用，这些步骤已经在父类中写好了。但是某些部分易变，易变部分可以抽象出来，供不同子类实现。这就是一种模板模式。

设计模式：模板方法 (TemplateMethod)

```
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);
        }
    }
}
```

另例：TemplateMethodTest.java

设计模式：模板方法 (TemplateMethod)

模板方法设计模式是编程中经常用得到的模式。各个框架、类库中都有他的影子，比如常见的有：

- 数据库访问的封装
- Junit单元测试
- JavaWeb的Servlet中关于doGet/doPost方法调用
- Hibernate中模板程序
- Spring中JDBCTemlate、HibernateTemplate等

练习3

编写工资系统，实现不同类型员工(多态)的按月发放工资。如果当月出现某个Employee对象的生日，则将该雇员的工资增加100元。

实验说明：

（1）定义一个Employee类，该类包含：

private成员变量name,number,birthday，其中birthday 为MyDate类的对象；

abstract方法earnings()；

toString()方法输出对象的name,number和birthday。

（2）MyDate类包含:

private成员变量year,month,day ；

toDateString()方法返回日期对应的字符串：xxxx年xx月xx日

（3）定义SalariedEmployee类继承Employee类，实现按月计算工资的员工处理。该类包括：private成员变量monthlySalary；

实现父类的抽象方法earnings(),该方法返回monthlySalary值；toString()方法输出员工类型信息及员工的name，number,birthday。

练习3

（4）参照SalariedEmployee类定义HourlyEmployee类，实现按小时计算工资的员工处理。该类包括：
private成员变量wage和hour；
实现父类的抽象方法earnings(),该方法返回wage*hour值；
toString()方法输出员工类型信息及员工的name，number,birthday。
（5）定义PayrollSystem类，创建Employee变量数组并初始化，该数组存放各类雇员对象的引用。利用循环结构遍历数组元素，输出各个对象的类型,name,number,birthday,以及该对象生日。当键盘输入本月月份值时，如果本月是某个Employee对象的生日，还要输出增加工资信息。

提示：

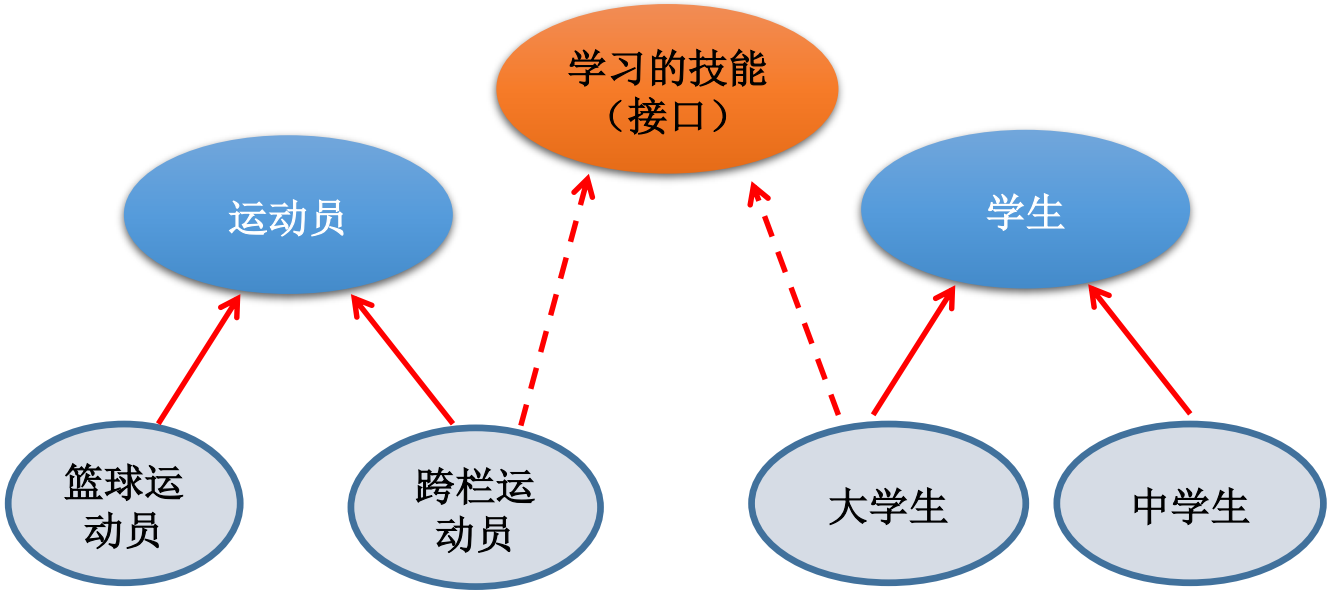
```
//定义People类型的数组People c1[]=new People[10];
//数组元素赋值
c1[0]=new People("John","0001",20);
c1[1]=new People("Bob","0002",19);
//若People有两个子类Student和Officer，则数组元素赋值时，可以使父类类型的数组元素指向子类。
c1[0]=new Student("John","0001",20,85.0);
c1[1]=new Officer("Bob","0002",19,90.5);
```

6-6 接口(interface)

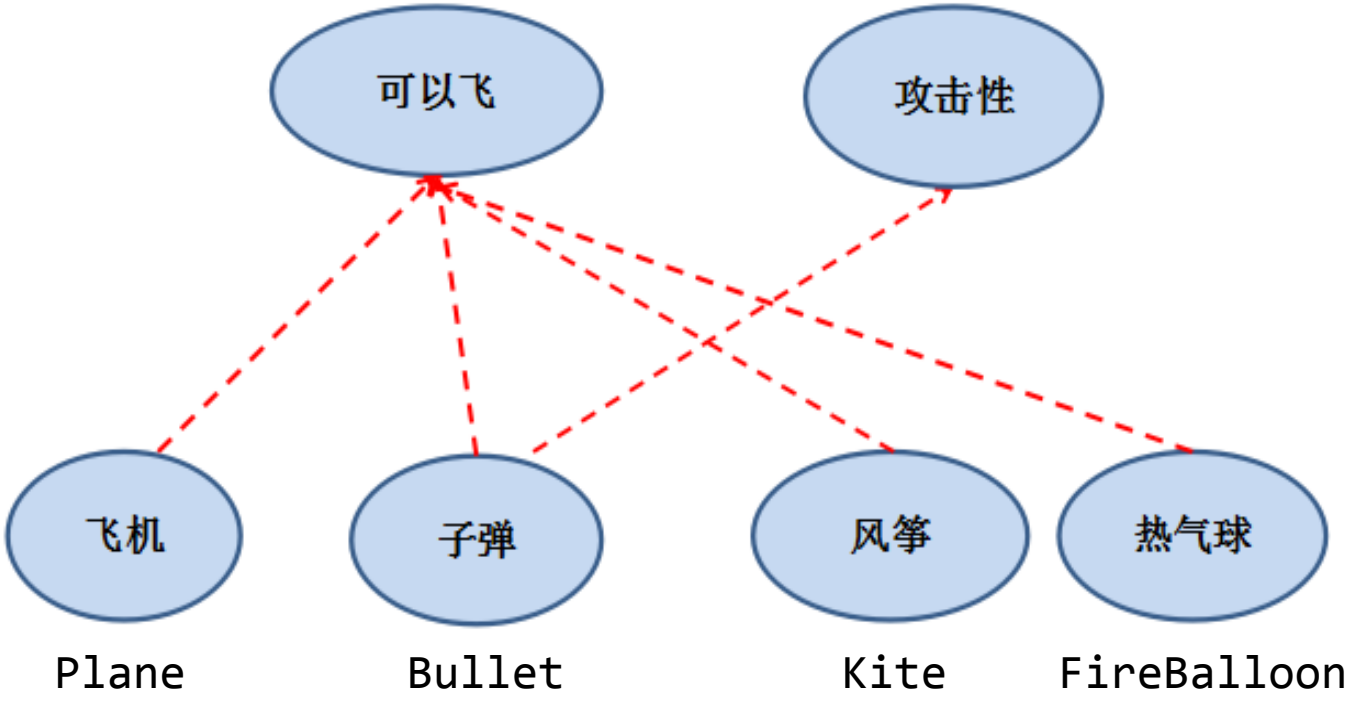
6.6 接 口：概述

- 一方面，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。
- 另一方面，有时必须从几个类中抽取出一些共同的行为特征，而它们之间又没有is-a的关系，仅仅是具有相同的行为特征而已。例如：鼠标、键盘、打印机、扫描仪、摄像头、充电器、MP3机、手机、数码相机、移动硬盘等都支持USB连接。
- 接口就是规范，定义的是一组规则，体现了现实世界中“如果你是/要...则必须能...”的思想。继承是一个"是不是"的关系，而接口实现则是 "能不能"的关系。
- 接口的本质是契约，标准，规范，就像我们的法律一样。制定好后大家都要遵守。

6.6 接 口：举例



6.6 接 口：举例



- 接口(interface)是抽象方法和常量值定义的集合。
- **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。
  - 接口中的所有抽象方法都默认是由public abstract修饰的。
  - 接口中没有构造器。
  - 接口采用多继承机制。
- 接口定义举例

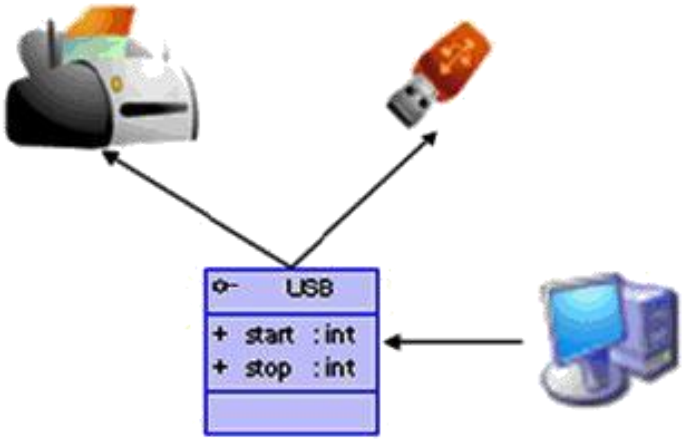
```
public interface Runner {
    int ID = 1;
    void start();
    public void run();
    void stop();
}
```

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

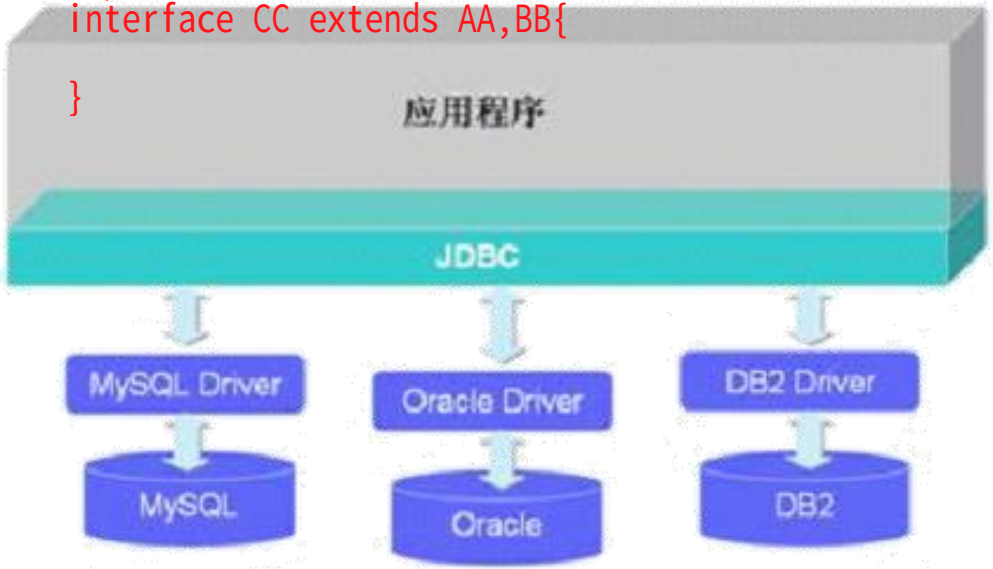
- 定义Java类的语法格式：先写extends，后写implements
  - class SubClass extends SuperClass **implements** InterfaceA{ }
- 一个类可以实现多个接口，接口也可以继承其它接口。
- 实现接口的类中必须提供接口中所有方法的具体实现内容，方可实例化。否则，仍为抽象类。
- 接口的主要用途就是被实现类实现。（面向接口编程）
- 与继承关系类似，接口与实现类之间存在多态性
- 接口和类是并列关系，或者可以理解为一种特殊的类。从本质上讲，接口是一种特殊的抽象类，这种抽象类中只包含常量和方法的定义(JDK7.0及之前)，而没有变量和方法的实现。

  java8之后可以定义静态方法和默认方法

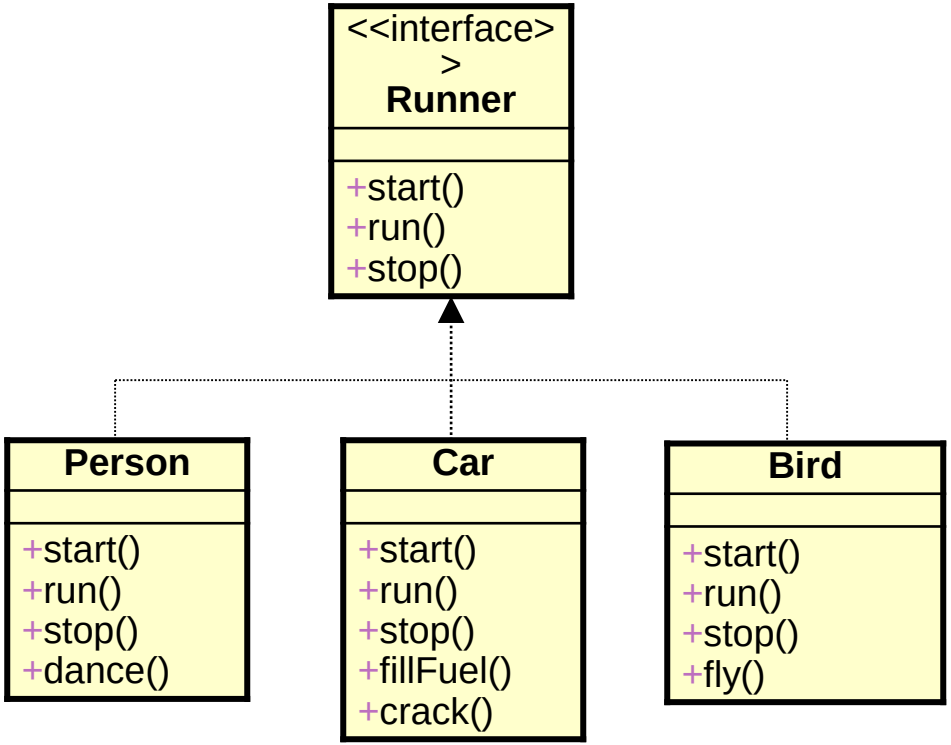
6.6 接 口：应用举例




```
/**
 * 接口可以多继承
 */
interface CC extends AA,BB{

}
```



接口应用举例(1)



接口应用举例(2)

```
interface Runner {
    public void start();
    public void run();
    public void stop();
}
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。
    }
}
```

接口应用举例(3)

● 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{
        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();}
}
```

接口应用举例(4)

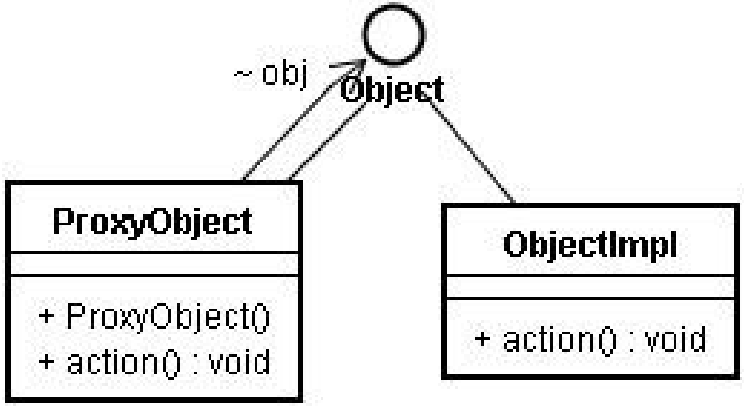
```
interface MyInterface{
        String s="MyInterface";
        public void absM1();
}
interface SubInterface extends MyInterface{
        public void absM2();
}
public class SubAdapter implements SubInterface{
        public void absM1(){System.out.println("absM1");}
        public void absM2(){System.out.println("absM2");}
}
```

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。否则，SubAdapter仍需声明为abstract的。

接口的应用：代理模式(Proxy)

概述：

代理模式是Java开发中使用较多的一种设计模式。代理设计就是为其他对象提供一种代理以控制对这个对象的访问。



接口的应用：代理模式(Proxy)

```
interface Network {
    public void browse();
}

// 被代理类
class RealServer implements   Network {

    @Override
    public void browse() {
        System.out.println("真实服务器上网浏览信息");
    }
}
```

另例：StaticProxyTest.java

```
// 代理类
class ProxyServer implements Network {
    private Network network;
    public ProxyServer(Network network) {
        this.network = network;
    }
    public void check() {
        System.out.println("检查网络连接等操作");
    }
    public void browse() {
        check();
        network.browse();
    }
}
public class ProxyDemo {
    public static void main(String[] args) {
        Network net = new ProxyServer(new RealServer());
        net.browse();
    }
}
```

接口的应用：代理模式(Proxy)

- 应用场景：
  - 安全代理：屏蔽对真实角色的直接访问。
  - 远程代理：通过代理类处理远程方法调用（RMI）
  - 延迟加载：先加载轻量级的代理对象，真正需要再加载真实对象

比如你要开发一个大文档查看软件，大文档中有大的图片，有可能一个图片有100MB，在打开文件时，不可能将所有的图片都显示出来，这样就可以使用代理模式，当需要查看图片时，用proxy来进行大图片的打开。

- 分类
  - 静态代理（静态定义代理类）
  - 动态代理（动态生成代理类）
    - ✓ JDK自带的动态代理，需要反射等知识

详见：

《拓展：工厂设计模式.doc》

接口和抽象类之间的对比

No.	区别点	抽象类	接口
1	定义	包含抽象方法的类	主要是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、(jdk8.0:默认方法、静态方法)
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	模板方法	简单工厂、工厂方法、代理模式
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	

在开发中，常看到一个类不是去继承一个已经实现好的类，而是要么继承抽象类，要么实现接口。

【面试题】排错：

```
interface A {
    int x = 0;
}
class B {
    int x = 1;
}
class C extends B implements A {
    public void pX() {
        System.out.println(x);
    }
    public static void main(String[] args) {
        new C().pX();
    }
}
```

【面试题】排错：

```
interface Playable {
    void play();
}

interface Bounceable {
    void play();
}

interface Rollable extends Playable,
Bounceable {
    Ball ball = new Ball("PingPang");

}
```

```
class Ball implements Rollable {
    private String name;

    public String getName() {
        return name;
    }

    public Ball(String name) {
        this.name = name;
    }

    public void play() {
        ball = new Ball("Football");
        System.out.println(ball.getName());
    }
}
```

练习4

- 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，声明redius属性，提供getter和setter方法
- 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InterfaceTest，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。

Java 8中关于接口的改进

Java 8中，你可以为接口添加静态方法和默认方法。从技术角度来说，这是完全合法的，只是它看起来违反了接口作为一个抽象定义的理念。

静态方法：使用 static 关键字修饰。可以通过接口直接调用静态方法，并执行其方法体。我们经常在相互一起使用的类中使用静态方法。你可以在标准库中找到像Collection/Collections或者Path/Paths这样成对的接口和类。

默认方法：默认方法使用 default 关键字修饰。可以通过实现类对象来调用。我们在已有的接口中提供新方法的同时，还保持了与旧版本代码的兼容性。比如：java 8 API中对Collection、List、Comparator等接口提供了丰富的默认方法。

```
public interface AA {
    double PI = 3.14;

    public default void method() {
        System.out.println("北京");
    }

    default String method1() {
        return "上海";
    }

    public static void method2() {
        System.out.println(“hello lambda!");
    }
}
```

接口中的默认方法

- 若一个接口中定义了一个默认方法，而另外一个接口中也定义了一个同名同参数的方法（不管此方法是否是默认方法），在实现类同时实现了这两个接口时，会出现：**接口冲突。**
  - 解决办法：实现类必须覆盖接口中同名同参数的方法，来解决冲突。

- 若一个接口中定义了一个默认方法，而父类中也定义了一个同名同参数的非抽象方法，则不会出现冲突问题。因为此时遵守：**类优先原则。**接口中具有相同名称和参数的默认方法会被忽略。

练习：接口冲突的解决方式

```
interface Filial {// 孝顺的
    default void help() {
        System.out.println("老妈，我来救你了");
    }
}
interface Spoony {// 痴情的
    default void help() {
        System.out.println("媳妇，别怕，我来了");
    }
}

class Man implements Filial, Spoony {
    @Override
    public void help() {
        System.out.println("我该怎么办呢？");
        Filial.super.help();
        Spoony.super.help();
    }
}
```

6-7 类的内部成员之五：内部类

6.7 类的成员之五：内部类

- 当一个事物的内部，还有一个部分需要一个完整的结构进行描述，而这个内部的完整的结构又只为外部事物提供服务，那么整个内部的完整结构最好使用内部类。
- 在Java中，允许一个类的定义位于另一个类的内部，前者称为内部类，后者称为外部类。
- Inner class一般用在定义它的类或语句块之内，在外部引用它时必须给出完整的名称。
  - Inner class的名字不能与包含它的外部类类名相同；
- **分类：** **成员内部类**（static成员内部类和非static成员内部类）个人理解class类型的属性

  局部内部类（不谈修饰符）、匿名内部类

  个人理解，局部内部类类似于class类型的局部变量

6.7 类的成员之五：内部类

- 成员内部类作为类的成员的角色：
  - 和外部类不同，Inner class还可以声明为**private**或**protected**；
  - 可以调用外部类的结构
  - Inner class 可以声明为**static**的，但此时就不能再使用外层类的非static的成员变量；
- 成员内部类作为类的角色：
  - 可以在内部定义属性、方法、构造器等结构
  - 可以声明为**abstract**类，因此可以被其它的内部类继承
  - 可以声明为**final**的
  - 编译以后生成OuterClass$InnerClass.class字节码文件（也适用于局部内部类）

【注意】

1. 非static的成员内部类中的成员不能声明为static的，只有在外部类或static的成员内部类中才可声明static成员。
2. 外部类访问成员内部类的成员，需要“内部类.成员”或“内部类对象.成员”的方式
3. 成员内部类可以直接使用外部类的所有成员，包括私有的数据
4. 当想要在外部类的静态成员部分使用内部类时，可以考虑内部类声明为静态的

内部类举例 (1)

```
class Outer {
    private int s;
    public class Inner {
        public void mb() {
            s = 100;
            System.out.println("在内部类Inner中s=" + s);
        }
    }
    public void ma() {
        Inner i = new Inner();
        i.mb();
    }
}
public class InnerTest {
    public static void main(String args[]) {
        Outer o = new Outer();
        o.ma();
    }
}
```

内部类举例 (2)

```
public class Outer {
    private int s = 111;
    public class Inner {
        private int s = 222;
        public void mb(int s) {
            System.out.println(s); // 局部变量s
            System.out.println(this.s); // 内部类对象的属性s
            System.out.println(Outer.this.s); // 外部类对象属性s
        }
    }
    public static void main(String args[]) {
        Outer a = new Outer();
        Outer.Inner b = a.new Inner();
        b.mb(333);
    }
}
```

- 如何声明局部内部类

```
class 外部类{
    方法(){
        class 局部内部类{
        }
    }
    {
        class 局部内部类{
        }
    }
}
```

- 如何使用局部内部类

➢ 只能在声明它的方法或代码块中使用，而且是先声明后使用。除此之外的任何地方都不能使用该类

➢ 但是它的对象可以通过外部方法的返回值返回使用，返回值类型只能是局部内部类的父类或父接口类型

局部内部类的特点

- 内部类仍然是一个独立的类，在编译之后内部类会被编译成独立的.class文件，但是前面冠以外部类的类名和$符号，以及数字编号。
- 只能在声明它的方法或代码块中使用，而且是先声明后使用。除此之外的任何地方都不能使用该类。
- 局部内部类可以使用外部类的成员，包括私有的。
- 局部内部类可以使用外部方法的局部变量，但是必须是final的。由局部内部类和局部变量的声明周期不同所致。
- 局部内部类和局部变量地位类似，不能使用public,protected,缺省,private
- 局部内部类不能使用static修饰，因此也不能包含静态成员

匿名内部类

- 匿名内部类不能定义任何静态成员、方法和类，只能创建匿名内部类的一个实例。一个匿名内部类一定是在new的后面，用其隐含实现一个接口或实现一个类。
- **格式：**

```
new 父类构造器（实参列表）|实现接口(){
    //匿名内部类的类体部分
}
```

- **匿名内部类的特点**
  - 匿名内部类必须继承父类或实现接口
  - 匿名内部类只能有一个对象
  - 匿名内部类对象只能使用多态形式引用

AnonymousTest.java

```
interface  A{
        public  abstract void fun1();
}
public class Outer{
        public static void main(String[] args) {
                new Outer().callInner(new A(){
    //接口是不能new但此处比较特殊是子类对象实现接口，只不过没有为对象取名
                        public void fun1() {
                                System.out.println(“implement for fun1");
                        }
                });// 两步写成一步了
        }
        public void callInner(A a) {
                a.fun1();
        }
}
```

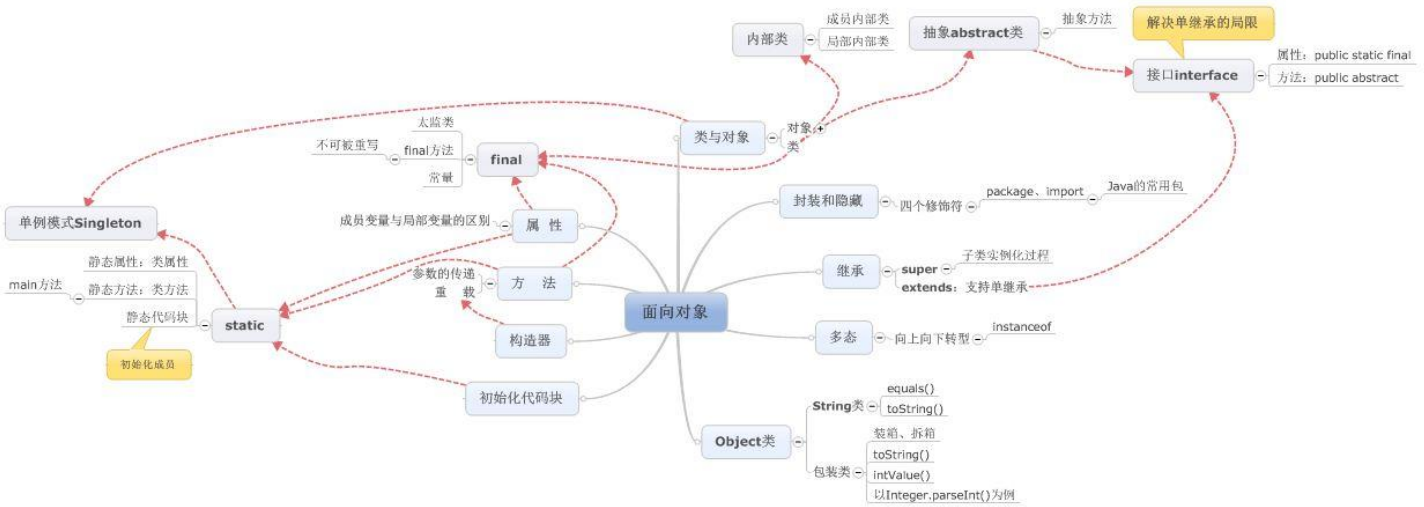
```
public class Test {
    public Test() {
        Inner s1 = new Inner();
        s1.a = 10;
        Inner s2 = new Inner();
        s2.a = 20;
        Test.Inner s3 = new Test.Inner();
        System.out.println(s3.a);
    }

    class Inner {
        public int a = 5;
    }
    public static void main(String[] args) {
        Test t = new Test();
        Inner r = t.new Inner();
        System.out.println(r.a);
    }
}
```

练习4

判断输出结果为何？

面向对象内容总结

面向对象特性，是Java学习的**核心、重头戏**。希望大家及时地梳理、总结




尚硅谷
让天下没有难学的的技术