3天学透 Actionscript（第一天）

第一天：什么是 Actionscript？

1.类的由来

1.1 C 语言中的结构体

这部分属于历史问题，与技术无关，了解历史可以让我们更好地把握现在和将来。C 语言中的结构体 struct 可以说是类的最原始的雏形。只有 int，float，char 这些基本数据类型是不够的，有时需要用不同的数据类型组合成一个有机的整体来使用。例如一个学生有学号和姓名就可以定义一个 Student 的结构体：

```
struct Student {
    int id;
    char[20] name;
} student;

main() {
 // 可以使用“对象名.属性” 的方式来操作数据
    student.id = 5;
    student.name = “ZhangSan”;
}
```

1.2 从结构体到类的演化（C —— C++）

C 中的结构体	C++ 中的结构体

struct 结构名 {
数据成员
};
	struct 结构名 {
	数据成员
	成员函数
	}

C++ 首次允许在结构体中定义成员函数！那么再将 struct 关键字换成 class 不就是我们现在所看到

的类的形态了吗？

```
class Student {
 private:
   int id;
   char[20] name;
 public:
   void gotoSchool() {}
}
```

C++ 最初的名字叫做“C with class”（带类的 C），经过长时间的发展，最终敲定将其命名为 C++，“++”表示加一操作，代表它比 C 语言更进步，更高级。

面向过程的编程就是在处理一个个函数，而现在的面向对象编程处理是函数加数据，形式上就这么点儿差别。也许刚刚接触时它时会感到有些困难，这很正常。一旦你真正了解它，那你一定会爱上它。所以，请大家不要畏惧，技术永远向着更方便，更简单，更高效的方向发展，而不会向越来越难，越来越复杂发展。对于面向对象程序设计（OOP）而言，代表着越来越接近人类的自然语言，越来越接近人类的思维，因此一切都会变得越来越简单。

从结构体到类的演变过程中我们看到，类中是可以定义函数的。因此，引出了面向对象三大特性之一，封装。

2.封装（Encapsulation）

2.1 封装的概念

封装的定义：把过程和数据包围起来，对数据的访问只能通过已定义的界面。在程序设计中，封装是指将数据及对于这些数据有关的操作放在一起。

知道这些定义，并不能代表技术水平有多高。但是如果去参加面试也许会用得着。简单解释一下，它的意思是指把成员变量和成员函数放在一个类里面，外面要想访问该类的成员变量只能通过对外公开的成员函数来访问。用户不需要知道对象行为的实现细节，只需根据对象提供的外部接口访问对象即可。

这里有一个原则叫做“信息隐藏”—— 通常应禁止直接访问成员变量，而应该通过对外公开的接口来访问。下面，看一个小例子：

```
class Father {
    private var money:int = 10000000;
    public takeMoney():int {
        money -= 100;
        return 100;
    }
}
```

定义名为 Father 的类（一个有钱的父亲），类中有一个成员变量 money 它的访问权限为 private 意思是说 money 是 Father 私有的，private 是指只有在这个类里面才能访问，出了这个类就无法访问该成员了，稍后会有关于访问权限更多的解释。

类中还定义了 takeMoney() 这个方法，它是 public 的，可以说是对外公开的一个接口。

从这个例子中可以看出，任何人要想从 Father 类取一些 money 的话，都只能通过 takeMoney() 这个

方法去拿，而不能直接访问 money 这个属性，因为它是私有的。只有通过 takeMoney() 这个公开的方法从能修改 Father 类的成员变量让 money -= 100 —— 每次只能给你 100 元。对外只能看到 takeMoney() 这个方法，该方法如何实现的别人不知道，反正你每次只能得到 100 块。

2.2 封装的好处

封装的好处：保证了模块具有较高的独立性，使得程序的维护和修改更加容易。对应程序的修改仅限于类的内部，将程序修改带来的影响减少到最低。

2.3 封装的目的

（1）隐藏类的实现细节；
（2）迫使用户通过接口去访问数据；
（3）增强代码的可维护性。

2.4 封装的技巧

按照纯面向对象编程的思想，类中所有的成员变量都应该是 private 的，要操作这些私有的成员变量只能通过对外公开的函数来完成。实际工作中，有时也常把变量的访问权限设置为 public 目的就是为了调用起来方便。在 AS 3 中提供了 get/set 关键字，它能让我们以函数调用的方式处理属性。按照封装的原则，再结合 get/set 如何去操作一个成员变量呢？看下面这个例子：

```
class Person {
 private var _age:int;

 public function get age():int {
  return _age;
 }

 public function set age(a:int):void {
  if (a < 0) {
   trace("age 不能小于 0！");
   return;
  }
  _age = a;
 }
}
```

首先，实例化出该类的对象 var person:Peron = new person()。如果要设置成员变量 _age 就要通过调用 set age() 来实现：person.age = 5。实际上，我们是在调用 set age(5) 这个方法，但是由于有了 set

修饰符，操作的方法就是给对象的属性赋值，但实际上是在调用函数。但是如果这么写 person.age = -20，这样合理吗？一个人的年龄等于 -20 岁，显然不对。因此在调用 set age() 方法中，可以进行一下判断，如果给的参数不对就发出提示，这项工作只能由函数来完成，这是属性没办法到的。

回想一下封装的定义，它的用意就在于数据（成员变量）是核心，不可以随随便便去改，要想改只能去调用公开的函数，而在函数可以进行各种判断，保证对数据的修改是经过思考的，合理的。

说到 get/set 现在去看看 ActionScript 3 是怎么做的。我们知道 TextField 类有一个 text 属性，查看帮助文档，会看到：

text 属性

实现

public function get text():String

public function set text(value:String):void

可见这个属性也是通过 get/set 方法实现的，虽然我们是用“对象名.text”方式去操作 text 属性，但实际上是在调用 get/set text()，那么这两个函数具体怎么实现的呢？Who knows！这就叫隐藏实现。AS 3 里面所有类的属性都是这样实现的，只有 get 没有 set 的是只读属性。

封装时还要考虑，这个方法应该归属哪个类。首先，逻辑上要说得过去，例如制作一个贪吃蛇的游戏，有一个方法叫 eat()，“吃”这个方法应该给谁？给文档类？给食物？当然应该放到蛇这个类里面，蛇去吃食物这才合理。然后，还要注意该方法要用到哪些属性。例如，一个人拿粉笔在黑板上画圆，那么画圆这个方法应该归属哪个类。人？黑板？粉笔？画圆的时候需要知道圆的半径和圆心，而这些属性都在圆这个类里面，所以画圆这个方法应该放在圆这个类里面。

下面，学习访问控制修饰符。

3. 访问控制（Access Control）

修饰符	类内可见	包内可见	子类可见	任何地方
private	Yes			
protected	Yes		Yes	
internal（默认）	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

（1）private：仅当前类可访问，称为私有成员；
（2）internal（默认）：包内可见，在同一文件夹下可访问（如果不写权限修饰符，默认权限也是它）；
（3）protected：子类可访问，与包无关。如果不是该类的子类，那么 protected 和 private 是一样的；
（4）public：完全公开。任何地方、任何对象都可以访问这个类的成员。

4. 继承（Inheritance）

4.1 继承的概念

下面进入面向对象的第二大特性 —— 继承。

继承的定义：继承是一个类可以获得另一个类的特性的机制，继承支持层次概念。

继承是一种代码重用的形式，允许程序员基于现有类开发新类。现有类通常称为"基类"或"超类"，新类通常称为"子类"或"派生类"。通过继承还可以在代码中利用多态。

继承的好处：继承最大的好处是代码的重用，与它同样重要的是它带来了多态。关于多态后面会给大家非常详细的讲解，我们现在只讨论继承。

AS 3 和 Java 一样都是单根继承的。在 AS 3 中不论哪个类都有一个相同的唯一的老祖宗 —— Object。拿我们最熟悉的 Sprite 类来说，查看一下帮助文档：

包 flash.display
类 public class Sprite
继承 Sprite→DisplayObjectContainer→InteractiveObject→DisplayObject→EventDispatcher→Object
子类 FLVPlayback，FLVPlaybackCaptioning，MovieClip，UIComponent

从这里可以清楚地看到，Sprite 的父类是 DisplayObjectContainer，而 DisplayObjectContainer 的父类是 InteractiveObject。一直往下找，最终都能够找到 Object 类，不仅是 Sprite，所有的类皆如此，这就叫单根继承。AS 单根继承的思想应该说是从 Java 借鉴来的，实践证明单根继承取得了很大的成功。Java 发明的单根继承摒弃了 C++ 的多继承所带来的很多问题。

虽然在 AS 3 中不能直接利用多继承，但是可以通过实现多个接口（interface）达到相同的目的，关于接口后面会单独讲解。单继承和多继承相比，无非多敲些代码而已，但是它带来的巨大好处是潜移默化的。具体有哪些好处这里就不详细讨论了，总之是非常非常多，大家如果有兴趣可以到网上搜索一下相关内容。

我们知道 Sprite 类有 x 和 y 属性，现在请大家在帮助文档中查找出 Sprite 类的这两个属性。你一定是先去打开 Sprite 这个类，但只看到五个属性，里面没有我没要找到 x, y 属性！怎么回事？这时候应该首先想到的是继承，一定是它的父类里面有，这两个属性是从父类中继承过来的！OK，没错，就是这样，那么就去 DisplayObjectContainer 找找，可是还没找到，再找父类的父类 InteractiveObject 还没有，再找

父类的父类的父类 DisplayObject 找到了吧。学会查帮助文档非常重要，知道某个类有某个属性或方法，如果在这个类中找不到就要去它的父类中去找，再找不到就去父类的父类中找，最远到 Object 不信你找不到。

下面看一些例子。

4.2 TestPerson.as —— 属性与方法的继承

```
package {
 public class TestStudent {
  public function TestPerson() {
   var student:Student = new Student();
   student.name = "ZhangSan";
   student.age = 18;
   student.school = "114";
   trace(student.name, student.age, student.school);
  }
 }
}

class Person {
 private var _name:String;
 private var _age:int;

 public function get name():String{
  return _name;
 }

 public function set name(n:String):void {
  if (n == "") {
   trace("name 不能为空！");
   return;
  }
  _name = n;
 }

 public function get age():int {
  return _age;
 }

 public function set age(a:int):void {
  if (a < 0) {
   trace("age 不能小于 0！");
```

```
    return;
   }
   _age = a;
  }
}

class Student extends Person {
 private var _school:String;

 public function get school():String {
  return _school;
 }

 public function set school(s:String):void {
  _school = s;
 }
}
```

注意，package 中包的是用来测试的类，为了演示方便 Person 和 Student 跟这个测试类放在了一个 as 文件中。

首先定义一个 Person 类，人都有名字（name）和年龄（age），下面让 Student 类继承 Person 类，也就是说学生继承了人的所有特性，或者说“学生是一个人”，这话没错吧！“Student is a Person”，满足 is- ，那么就可以使用继承。Student 在 Person 的基础上还多出了 school 属性，记录着他所在学校的名称。

在测试类中创建了一个 Student 对象，使用 student.name，student.age，student.school，设置学生的姓名，年龄和学校。虽然 Student 类中没有定义 name，age 属性，但这两个属性是它从父类 Person 继承而来的，因此实现了代码的复用，因此 Student 也就拥有了父类属性和方法。

4.3 TestExtends.as —— 继承的限制

```
package {
 public class TestExtends {
  public function TestExtends() {
   new Son();
  }
 }
}

class Father {
 private var money:int = 1000000;
 public var car:String = "BMW";
 private function work():void {
  trace("writing");
```

```
 }
}

class Son extends Father {
 public var bike:String = "YongJiu";

 // 没有继承因此谈不上重写
 private function work():void {
  trace("studying");
 };

 function Son (){
  trace(bike);
  trace(car);
  //trace(money); // private 的属性不能被继承
  work();
 }
}
```

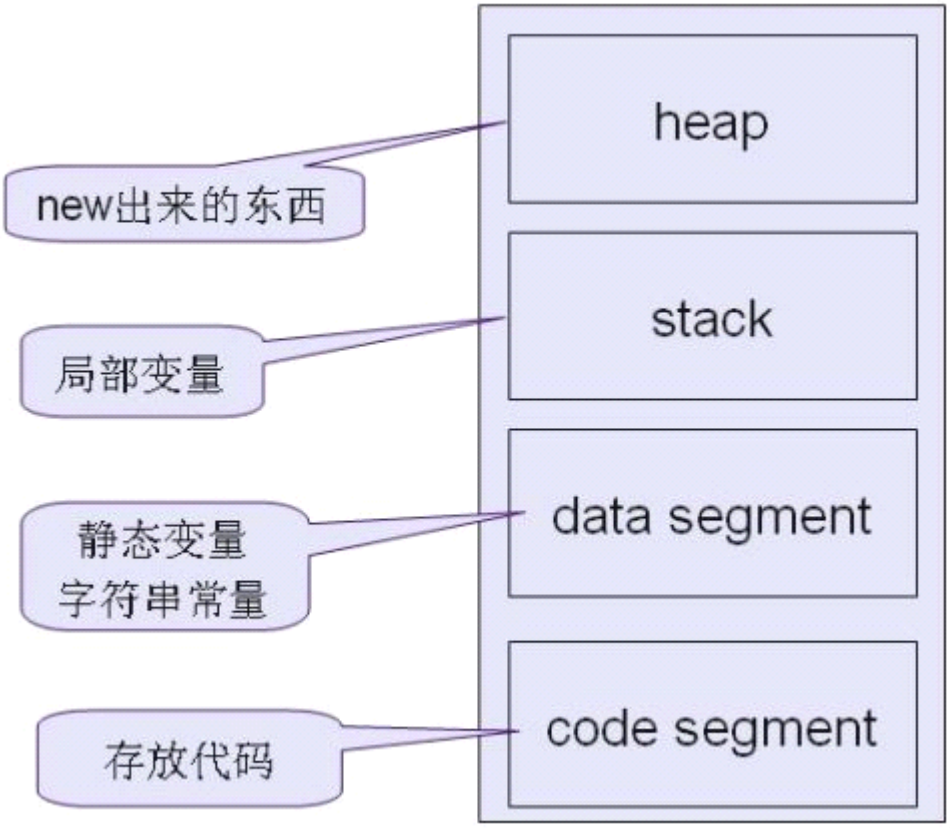
这个例子要演示哪些属性或方法不能被继承。本例中定义了一个 Father 类，其中包括 private 的 money 属性，public 的 car 属性和 private 的 work() 方法。

然后定义一个 Son 类继承自 Father 类，在 Son 类中新增加一个 bike 属性，同样也有一个 work() 方法。在 Son 的构造函数中打印出 bike，car 属性或调用 work() 方法都没问题。但是不能打印出 money 属性，因为它是 Father 类私有的，回想一下第三节所讲的“访问控制”查一下那个表，可以看到 private 的成员只能在类内访问，不能被继承。那么父类中还有一个 work() 方法也是 private 的，因此也不会被继承，所以在子类中再定义一个 work() 方法也不会有冲突，也就谈不上重写。关于重写，后面还有相关的例子。

学习 OOP 编程时，如果可以理解程序执行时内存的结构，那将对我们学习 OOP 编程有莫大好处。下面就来了解一下内存的结构以及各部分的作用。

4.4 内存分析

根据不同的操作系统内存的结构可能有所差异，但是通常在程序执行中会把内存分为四部分：



（1）heap 堆：存放 new 出来的对象；

（2）stack 栈：局部变量；

（3）data segment 数据段：静态变量和字符串常量；

（4）code segment 代码段：存放代码。

后面会在一些例子中利用内存图来帮助理解。先看下面一个例子 this 和 super。

4.5 this 和 super 关键字

4.5.1 TestSuper.as —— this 和 super 以及 override

```
package {
 public class TestSuper {
  public function TestSuper() {
   new Extender();
  }
 }
}

class Base {
 public var name:String;
```

```
 public var age:uint;

 //function Base(){
  // 即使不写构造函数，系统也会自动添加的无参构造函数
 //}

 public function work():void {
  trace("Writing!");
 }
}

class Extender extends Base {
 //public var name:String; // name 属性已从父类继承，不能重复定义

 function Extender() {
  // super(); // 即使不写 super()，系统也会添加的无参的 super();
  super.work();
  this.work();
 }

 override public function work():void {
  trace("Programming!");
 }
}
```

这个例子演示了 this 与 super 关键字的使用，以及重写（override）的概念。

首先，定义 Base 类代表基类，它被 Extender 类继承。Base 类中定义两个 public 的属性 age 和 name，以及一个公有的方法 work()。Extender 类继承了 Base 类的这些属性和方法，但是如果要想在 Extender 类中再重复定义 name 或 age 属性就不行了，因为它已经继承了这两个属性，不能再重复定义。但是方法可以被重新定义，前提是要在方法前面加上 override 关键字，显示地说明要重写（覆盖）基类的 work() 方法。

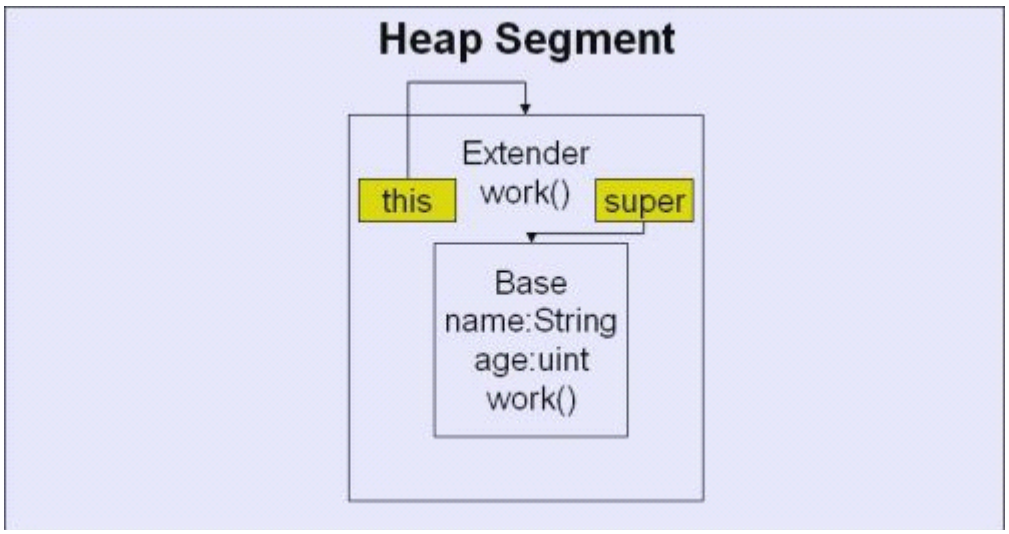
这一样来就有两个 work() 方法了如何区分它们呢？我们可以在方法名前面显示地加上 super 或 this 关键字，见 Extender 的构造函数。super.work() 表示调用父类的 work() 方法，this.work() 表示调用该类中的 work() 方法。

override 关键字表示对父类方法的重写，override 是利用继承实现多态的必要手段。

何时使用重写呢？当我们对父类的方法不满意、或者同样的一个方法对于两个类来说实现的功能不相同时，就要考虑重写，把父类的方法冲掉，让它改头换面。

下面我们看一下内存图中，this 和 super 各代表着什么。

4.5.2 this 和 super 的内存分析



测试类中有一条语句 new Extender()。new 出来的对象放在堆内存（Heap Segment）中。在对象内部隐藏着两个对象：this 和 super。

this 持有当前对象的引用；super 持有父类的引用。

上例中我们有两种调用的方法：super.work() 和 this.work()。如果不加修饰，只写 work()，那么系统默认调用的是 this.work()。

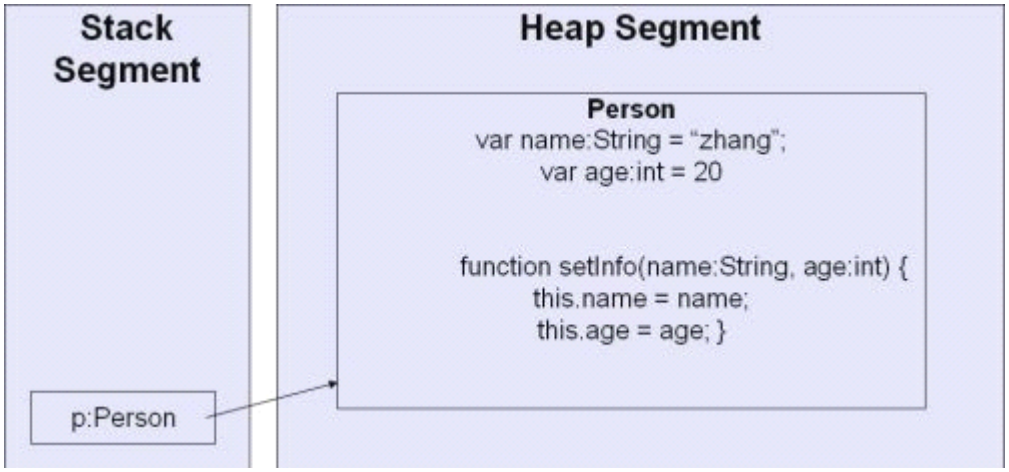
既然系统默认调用的就是 this.work()，那还要 this 有什么用？比如，当局部变量与成员变量同名时，如果要在属于这个局部变量的上下文中引用成员变量，那么就要显示地调用“this.成员变量”名来指定引用的是成员变量而非局部变量。请看下面一个例子。

4.5.3 TestThis.as —— 用 this 区分局部变量与成员变量

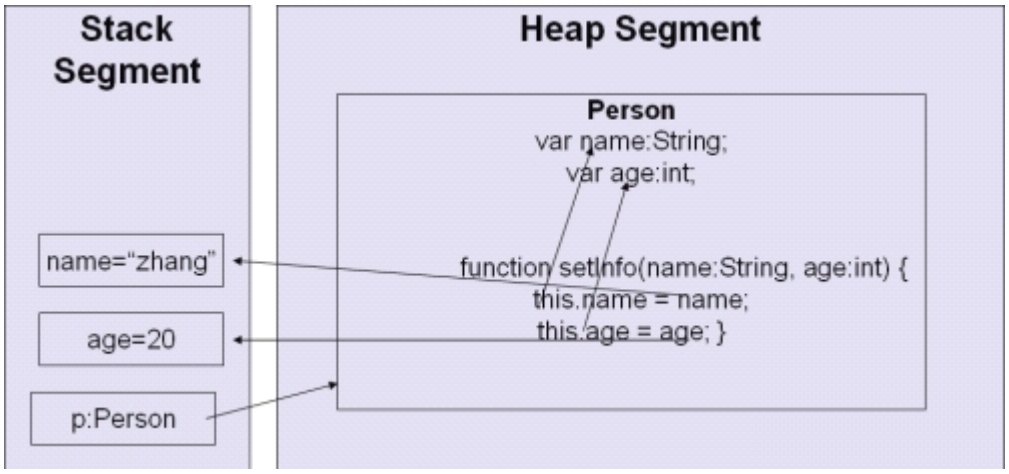
```
package {
 public class TestThis {
  public function TestThis() {
   var p:Person = new Person();
   p.setInfo("ZhangSan", 20);
  }
 }
}

class Person {
 private var name:String;
 private var age:int;

 public function setInfo(name:String, age:int) {
  this.name = name;
```

```
    this.age = age;
  }
}
```

本例中就是用 this.name 指定是成员变量 name，而非传进来的局部变量的 name。age 也是如此。下面请看内存分析。



传进去的两个变量 name 和 age 属于临时变量存放在栈内存（Stack Segment），setInfo() 方法中的 this.name 和 this.age 则指的是该对象的成员变量 name 和 age。由于 name 和 age 都是基元数据类型，因此直接传值，把 "zhang" 和 20 赋给了 Person 的成员变量。最后在 setInfo() 执行完成后，为该方法分配的临时变量全部释放，赋值工作完成。



4.6 初始化顺序

下面 Think in Java 中的一段演示代码，见 TestSandwich.as：

```
package {
 public class TestSandwich {
```

```
  public function TestSandwich() {
    new Sandwich();
  }
 }
}

class Water {
 //static var w = trace("static water");
 function Water() {
  trace("Water");
 }
}

class Meal {
 //static var w = trace("static meal");
 function Meal() {
  trace("Meal");
 }
}

class Bread {
 function Bread() {
  trace("Bread");
 }
}

class Cheese {
 function Cheese() {
  trace("Cheese");
 }
}

class Lettuce{
 function Lettuce() {
  trace("Lettuce");
 }
}

class Lunch extends Meal {
 function Lunch() {
  trace("Lunch");
 }
}
```

```
class PortableLunch extends Lunch {
 //static var w = trace("static lunck");
 function PortableLunch() {
  trace("PortableLunch");
 }
}

class Sandwich extends PortableLunch {
 var bread:Bread = new Bread();
 var cheese:Cheese = new Cheese();
 var lettuce:Lettuce = new Lettuce();

 //static var good = trace("static sandwich");

 function Sandwich() {
  trace("Sandwich");
 }
}
```

测试类很简单只有一句：new Sandwich()。构造出 Sandwich 类一个实例。

Sandwich 类继承了 PortableLunch 这个类。现在有一个问题，是先有子类还是先有父类？是先有父亲后有儿子，还是先有儿子后有父亲？肯定是先有父亲。那么怎么有的父亲？需要先构造出来。怎么构造？调用构造函数！

因此，我们说在构造子类之前，要先将它的父类构造出来，如果父类还有父类，就要先把父类的父类构造出来。在这段程序中每个类在构造出来后都会打印出该类的类名。下面请看执行结果：

Bread

Cheese

Lettuce

Meal

Lunch

PortableLunch

Sandwich

我们看到，最先打印出来的是 Bread，Cheese，Lettuce。这是 Sandwich 类的三个成员变量，可见在调用构造函数之前，要新将该类的成员变量构造出来，然后再去构造这个类本身。

前面提到，要构造这个类就先要构造它的父类，Sandwich 的父类是 PortableLunch，而 PortableLunch 还有父类叫 Lunch，而 Lunch 还有父类叫 Meal，到了 Meal 就终止了。这时将执行 Meal 的构造函数，有了 Meal 之后就可以构造 Lunch 了，有了 Lunch，PortableLunch 就可以构造，有了 PortableLunch 我们的 Sandwich 才被构造出来。

现在我们得到的结论就是：类的成员变量先被初始化，然后才是构造函数。

下面，请大家把代码中注释掉的部分全部打开。现在，又新加入了一些 static 的成员变量，我们来实验一下静态的成员变量是何时被调用的，执行结果如下：

static water

static meal

static lunck

static sandwich

Bread

Cheese

Lettuce

Meal

Lunch

PortableLunch

Sandwich

我们看到，所有静态的成员都先于非静态成员变量被构造出来！最上面有一个 Water 类，虽然没有地方会用到它，但是它也被打印出来了。结论是：当类被加载时静态的属性和方法就会被初始化。注意，什么叫类被加载时？这是指类被加载到内存里面。可见，我们整个这个 as 文件中的所有类都被加载到了内存中了，只要这个类被读入到内存中，那么它的所有静态成员就会被初始化。

最终的结论 —— 初始化顺序：

（1）当类被加载时该类的静态的属性和方法就会被初始化

（2）然后初始化成员变量

（3）最后构造出这个类本身

OK，既然这里提到了静态成员，下面我们就来了解一下它。

4.7 静态属性与方法

4.7.1 static 的概念

静态属性与方法属于这个类，而不属于该类的实例。

静态属性只生成一份。同类对象的静态属性值都相同。改变一个类的静态属性值会影响该类所有的对象。静态属性可以节省内存，方便调用。一个方法即使不声明为静态的实际上也只生成一份，因为除了方法所处理的数据不同外，方法本身都相同的。

静态属性和静态方法中不能存在非静态的属性和方法，或者说 static 的属性和方法中不能出现 this。

4.7.2 TestStatic.as —— static 属于这个类，不属于该类实例

```
package {
 public class TestStatic {
  public function TestStatic() {
   var b:Ball = new Ball();

   // 静态属性或方法属于这个类，而不属于该类对象，只能用类名引用
   // b.sMethod();
   // trace(b.color);
```

```
    Ball.sMethod();
    trace(Ball.color);
   }
  }
}

class Ball {
 public var xpos:Number = 300;

 public static var color:uint = 0xff0000;

 public function changeColor():void {
  color = 0;
 }

 public static function sMethod():void {
  trace("I'm a static Method");
  //trace(xpos); // 静态方法中不能调用非静态成员变量
  //changeColor(); // 静态方法中不能调用非静态成员函数
 }
}
```

在 Ball 这个类中有一个静态属性 color，一个静态方法 sMethod()。注意这条原则：静态属性与方法属于这个类，而不属于该类的实例。在 sMethod 中不能出现非静态的成员变量或方法。反之可以，在非静态的成员变量或方法中可以调用静态的成员变量或方法。

注意，在测试类 TestStatic 中，要访问静态的成员变量或方法只能通过“类名.方法（或属性）”的形式去调用，而不能通过“实例名.方法（或属性）”的形式调用。

由于静态属性只生成一份，所有该类对象都共享这一个，因此可以节省一部分内存，并且可以一改全改。既然大家都用一份属性，那就不能存在差别化了，这也是声明静态属性的一个原则，当所有对象都共用一个相同的属性时，可考虑将其声明为静态属性。而静态方法带来的好处就是方便引用。

下面，我们看看在设计模式中是如何使用 static 的，有请单例模式。

4.7.4 单例模式（Singleton Pattern）

单例模式是指“只能有一个该类的对象存在”。

单例模式有什么用处？例如系统的缓存、注册表、日志、显卡驱动、回收站等等都是独一无二的，都只有一个。如果造出了多个实例，就会导致系统出问题。这时就需要用到单例模式，原则就是该类的对象只能有一个。

首先，如果一个类的构造函数是公开的，那么就有可能被其它地方 new 出来，首先将构造函数变为私有的。遗憾的是 ActionScript 3 中，构造函数只能是 public 的，不能为 private。以下是 AS 3 版本的单例模式：

```
package {
    public class Singleton {
        static private var instance:Singleton;
        public function Singleton(singletonEnforcer:SingletonEnforcer) {}
        public static function getInstance():Singleton {
            if (instance == null) {
                instance = new Singleton(new SingletonEnforcer());
            }
            return instance;
        }
    }
}
class SingletonEnforcer { }
```

这里用 SingletonEnforcer 类作为 Singleton 类构造函数的参数，由于这两个类放在一个 as 文件中，instance 这个类只能被 Singleton 访问到，保证其它类得不到 SingletonEnforcer 的实例，用这种方法达到私有构造函数的作用。

这就是单例模式，一个最简单的设计模式！单例模式还可分为饿汉式 / 懒汉式。上面演示的是懒汉式单例模式，它在第一次调用 getInstance() 方法时才 new 出这个对象来。没人调用它的话，它永远不会主动去做，是不是很懒？！另一种是饿汉式的，instance 一上来就 new 出来该类的实例。

3天学透 Actionscript（第二天）

5. 多态（Polymorphism）

5.1 多态的概念

面向对象的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

多态的作用：消除类型之间的耦合关系。

现实中，关于多态的例子不胜枚举。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

下面是多态存在的三个必要条件，要求大家做梦时都能背出来！

5.2 多态存在的三个必要条件

一、要有继承；

二、要有重写；

三、父类引用指向子类对象。

5.3 TestPolymoph.as —— 多态的应用，体会多态带来的好处

```
package {
    public class TestPolymoph {
        public function TestPolymoph() {
            var cat:Cat = new Cat("MiMi");
            var lily:Lady = new Lady(cat);

            // var dog:Dog = new Dog("DouDou");
            // var lucy:Lady = new Lady(dog);

            lady.myPetEnjoy();
        }
    }
}

class Animal {
    private var name:String;
    function Animal(name:String) {
        this.name = name;
    }
    public function enjoy():void {
        trace("call...");
    }
}

class Cat extends Animal {
    function Cat(name:String) {
        super(name);
    }
    override public function enjoy():void {
        trace("Miao Miao...");
    }
```

```
}

class Dog extends Animal {
    function Dog(name:String) {
        super(name);
    }
    override public function enjoy():void {
        trace("Wang Wang...");
    }
}

// 假设又添加了一个新的类 Bird
class Bird extends Animal {
    function Bird(name:String) {
        super(name);
    }
    override public function enjoy():void {
        trace("JiJi ZhaZha");
    }
}

class Lady {
    private var pet:Animal;

    function Lady(pet:Animal) {
        this.pet = pet;
    }

    public function myPetEnjoy():void {
        // 试想如果没有多态
        //if (pet is Cat) { Cat.enjoy() }
        //if (pet is Dog) { Dog.enjoy() }
        //if (pet is Bird) { Bird.enjoy() }
        pet.enjoy();
    }
}
```

首先，定义 Animal 类包括：一个 name 属性（动物的名字），一个 enjoy() 方法（小动物玩儿高兴了就会叫）。接下来，定义 Cat，Dog 类它们都继承了 Animal 这个类，通过在构造函数中调用父类的构造函数可以设置 name 这个属性。猫应该是“喵喵”叫的，因此对于父类的 enjoy() 方法进行重写（override），打印出的叫声为 “Miao Maio…”。Dog 也是如此，重写 enjoy 方法，叫声为 “Wang Wang…”。

再定义一个 Lady 类，设置一个情节：假设这个 Lady 是一个小女孩儿，她可以去养一只宠物，这个小动物可能是 Cat，Dog，或是 Animal 的子类。在 Lady 类中设计一个成员变量 pet，存放着宠物的引用。

具体是哪类动物不清楚，但肯定是 Animal 的子类，因此 pet 的类型为 Animal，即 pet:Animal。注意这是父类引用，用它来指向子类对象。

最后在 Lady 类里面有一个成员函数 myPetEnjoy()，这个方法中只有一句 pet.enjoy()，调用 pet 的 enjoy() 方法。

现在来看测试类。new 出来一只 Cat，new 出来一个 Lady，将 Cat 的对象传给 Lady。现在 Lady 中的成员变量应该是 pet:Animal = new Cat("MiMi")。下面，调用 lady.myPetEnjoy() 方法，实际就是在调用 pet.enjoy()，打印出 Miao Miao。pet 的类型明明是 Animal，但被调的方法却是 Cat 的 enjoy()，而非 Animal 的 enjoy()，这就叫动态绑定——"在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法"。

想象一下，如果没有多态的话，myPetEnjoy() 中方法可能要做这样的一些判断：

```
if (pet is Cat) { new Cat("c").enjoy() }
if (pet is Dog) { new Dog("d").enjoy() }
```

判断如果 pet 是 Cat 类型的话调用 new Cat().enjoy()，如果是 Dog 的话调用 new Dog().enjoy()。假设有一天我要传入一个 Bird，那还得手动加上：

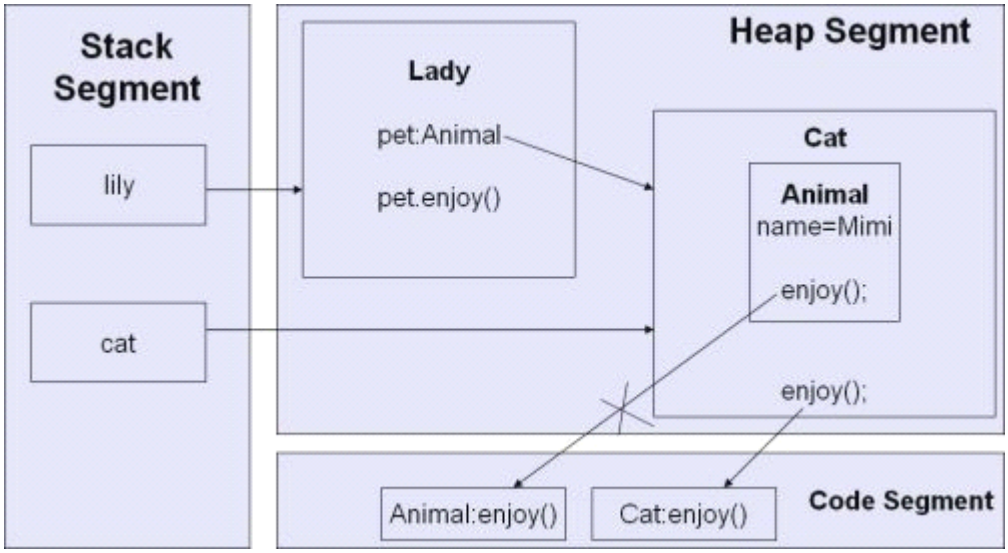
```
if (pet is Bird) { new Bird ("b").enjoy() }
```

新加入什么类型的都要重新修改这个方法，这样的程序可扩展性差。但是现在我们运用了多态，可以随意地加入任何类型的对象，只要是 Animal 的子类就可以。例如，var lily:Lady = new Lady(new Bird("dudu"))，直接添加进去就可以了，不需要修改其它任何地方。这样就大大提升的代码的可扩展性，通过这个例子好好体会一下多态带来的好处。

最后再补充一点，在使用父类引用指向子类对象时，父类型的对象只能调用是在父类中定义的，如果子类有新的方法，对于父类来说是看不到的。拿我们这个例子来说，如果 Animal 类不变，在 Cat 和 Dog 中都新定义出一个 run() 方法，这个方法是父类中没有的。那么这时要使用父类型的对象去调用子类新添加的方法就不行了。

下面看一个这个例子的内存图。

5.4 Test Polymoph 内存分析



在内存中，一个个方法就是一段段代码，因此它们被存放在代码段中。上例中的 pet 是 Animal 类型的成员变量，但是它指向的是一个 Cat 类型的具体对象，同时 Cat 又是它的子类，并且重写了 enjoy() 方法，满足了多态存在的三个必要条件。那么当调用 pet.enjoy() 的时候，调用的就是实际对象 Cat 的 enjoy() 方法，而非引用类型 Animal 的 enjoy() 方法。

5.5 多态的好处

多态提升了代码的可扩展性，我们可以在少量修改甚至不修改原有代码的基础上，轻松加入新的功能，使代码更加健壮，易于维护。

在设计模式中对于多态的应用比比皆是，面向对象设计（OOD）中有一个最根本的原则叫做“开放 - 关闭”原则（Open-Closed Principle OCP），意思是指对添加开放，对修改关闭。看看上面的例子，运用了多态以后我们要添加一个 Bird 只需要再写一个 Bird 类，让它继承自 Animal，然后 new 出来一个对象把它传给 lily 即可。

我们所做的就是添加新的类，而对原来的结构没有做任何的修改，这样代码的可扩展性就非常好了！因为我们遵循了“开放-关闭”原则 —— 添加而不是修改。

前面这个例子中还有一个地方需要说明，Animal 这个类，实际上应该定义为一个抽象类，里面的 enjoy() 方法，事实上不需要实现，也没法实现。想一想，Animal 的叫声？！你能想象出 Animal 是怎么叫的吗？显然，这个方法应该定义为一个抽象方法，留给它的子类去实现，它自己不需要实现，那么一旦这个类中有一个方法抽象的，那么这个类就应该定义为抽象类。但是很遗憾 AS 3 不支持抽象类，因为它没有 abstract 关键字。但是抽象类也是一个比较重要的概念，因此下面还要给大家补充一下。

5.6 抽象类的概念

一个类如果只声明方法而没有方法的实现，则称为抽象类。

含有抽象方法的类必须被声明为抽象类，抽象类必须被继承，抽象方法必须被重写。如果重写不了，应该声明自己为抽象。

抽象类不能被实例化。

抽象方法只需声明，而不需实现。

ActionScript 3.0 不支持抽象类（abstract），以后肯定会支持的，相信我，那只是时间问题。因此这里只介绍一下抽象类的概念。

5.7 对象转型（Casting）

一个基类类型变量可以“指向”其子类的对象。

一个基类的引用不可以访问其子类对象新增加的成员（属性和方法）。

可以使用“变量 is 类名”来判断该引用型变量所“指向”的对象是否属于该类或该类的子类。

子类的对象可以当作基类的对象来使用称作向上转型（upcasting），反之称为向下转型（downcasting）。

每说到转型，就不得不提到“里氏代换原则（LSP）”。里氏代换原则说，任何基类可以出现的地方，子类一定可以出现。里氏代换原则是对“开放—关闭”原则的补充。

里氏代换原则准确的描述：在一个程序中，将所有类型为 A 的对象都转型为 B 的对象，而程序的行为没有变化，那么类型 B 是类型 A 的子类型。

比如，假设有两个类：Base 和 Extender，其中 Extender 是 Base 的子类。如果一个方法可以接受基

类对象 b 的话: method(b:Base) 那么它必然可以接受一个子类对象 e，即有 method(e)。注意，里氏代换原则反过来不能成立。使用子类对象的地方，不一定能替换成父类对象。

向上转型是安全的，可以放心去做。但是在做向下转型，并且对象的具体类型不明确时通常需要用 instanceof 判断类型。下面看一个例子 TestPolymoph.as：

```
package {
    public class TestCast {
        public function TestCast() {
            // -------------- UpCasting --------------
            var cat:Cat = new Cat();
            var dog:Dog = new Dog();
            var animal:Animal = Animal(cat);
            animal.call();
            animal.sleep();
            //animal.eat(); // 不能调用父类中没有定义的方法

            // ------------- DownCasting -------------
            if (animal is Cat) {
                cat = Cat(animal);
                cat.eat();
            } else if (animal is Dog) {
                dog = Dog(animal);
                dog.eat();
            }
        }
    }
}

class Animal {
    public function call():void{};
    public function sleep():void{};
}
class Cat extends Animal {
    override public function call():void {
        trace("Cat Call");
    }
    override public function sleep():void {
        trace("Cat Sleep");
    }
    public function eat():void {
        trace("Cat Eat");
    }
}
```

```
class Dog extends Animal {
    override public function call():void {
        trace("Dog Call");
    }
    override public function sleep():void {
        trace("Dog Sleep");
    }
    public function eat():void {
        trace("Dog Eat");
    }
}
```

首先创建 Animal 类，定义两个方法 call() 和 sleep()，它的子类 Cat 和 Dog 分别重写了这两个方法，并且都扩展了出了一个新的方法 eat()。

来看测试类，new 出来一个 cat，再将它向上转型 animal:Animal = Animal(cat)。由于向上转型是安全的，所以这样做没有问题，但是当它转型成了父类对象后，就不能再调用 eat() 方法了，因为在父类中只有 call() 和 sleep() 方法，父类对象不能调用子类扩展出的新方法。

接下来一段代码是在进行向下转型，animal 这个对象可以是一个放一个 dog 也可以放一个 cat，当这两种情况都有可能时，进行向下转型就要判断一下当然对象到底是哪个类型的，使用“is”进行判断，看看该对象是不是一个 Cat 或 Dog，如果是 Cat 就将它向下转型为一个 Cat，这样就可以安全地调用 Cat 的 eat() 方法了。

最后再举一个现实中的例子 TestEventCast.as ：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    public class TestEventCast extends Sprite {
        public function TestEventCast() {
            var ball:Sprite = new Sprite();
            ball.graphics.beginFill(0xff0000);
            ball.graphics.drawCircle(0,0,50);
            ball.graphics.endFill();
            ball.y = 150;
            ball.x = 150;
            addChild(ball);
            ball.addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(evt:Event):void {
          // evt.target 是 Object 类型的，需要转型成为实际类型才能使用 x 属性
            var ball:Sprite = Sprite(evt.target);
            ball.x += 5;
        }
```

```
    }
}
```

构造函数中创建一个 Sprite 类的对象，并在里面绘制一个圆，加入 ENTER_FRAME 侦听，在 onEnterFrame 函数中，var ball:Sprite = Sprite(evt.target) 这里我们必须做向上转型，如果不做的话系统会报错，为什么呢？

查看一下帮助文档，Event 类 target 属性的实现：public function get target():Object。这是一个只读属性，它返回的是一个 Object 类型的对象。由于 AS 3 是单根继承的，因此任何一个对象都可以向上转型成 Object 类型的。因此每次要拿到这个 evt.target 的时候都要将它向下转型成为该对象的实际类型才能放心使用。

6. 接口（Interface）

6.1 接口的概念

每次说到接口，我都会想到现在很流行的一句话 —— “三流的企业卖产品，二流的企业卖服务，一流的企业卖标准”。接口就是在“卖标准”。

接口是方法声明的集合，让不相关的对象能够彼此通信。接口是实现“多继承”的一种手段。因此这一节非常重要。

接口仅包含一组方法声明，没有具体的代码实现。实现接口的类必须按照接口的定义实现这些方法，因此，实现同一个接口的类都具有这个接口的特征。

接口与类的区别：接口中只能定义成员方法，不能定义成员变量。接口中的方法都是抽象方法（没有具体实现）。

6.2 依赖倒转原则（Dependence Inversion Principle）

如果说“开放—关闭”原则是面对对象设计的目标，那么依赖倒转原则就是这个面向对象设计的主要机制。

依赖倒转原则讲的是：要依赖于抽象，不要依赖于具体。

依赖倒转原则的另一种表述是：要针对接口编程，不要针对实现编程。针对接口编程意思就是说，应当使用接口或抽象类来编程。它强调一个系统内实体间关系的灵活性。如果设计者要遵守“开放—关闭”原则，那么依赖倒转原则便是达到此要求的途径，它是面向对象设计的核心原则，设计模式的研究和应用均以该原则为指导原则。

6.3 实现接口的原则

在实现接口的类中，实现的方法必须（选自帮助文档）：

（1）使用 public 访问控制标识符。

（2）使用与接口方法相同的名称。

（3）拥有相同数量的参数，每一个参数的数据类型都要与接口方法参数的数据类型相匹配。

（4）使用相同的返回类型。

6.4 TestInterFaceAccess.as —— 实现多个接口

```
package {
    public class TestInterFaceAccess {
        public function TestInterFaceAccess() {
            var duck:Duck = new Duck();
            duck.run(56);
            duck.fly();
        }
    }
}

interface Runnable {
    function run(meter:uint):void;
}

interface Flyable {
    function fly():void;
}

class Duck implements Runnable, Flyable {
    public function run(meter:uint):void {
        trace("I can run " + meter + " meters");
    }

    public function fly():void {
        trace("I can fly");
    }
}
```

这个例子很简单，首先定义两个接口 Runnable 和 Flyable。Runnable 中定义了一个抽象的 run() 方法，Flyable 中定义了一个抽象的 fly() 方法。我们知道，接口是在定义标准，它自己不需要实现，具体的实现交给实现该接口的类去完成。

Duck 类实现（implements）了 Runnable 和 Flyable，因此它必需去实现这两个接口中定义的所有方法。并且方法名，参数类型，返回值类型要与接口中定义的完全一致，权限修饰符必需是 public。大家可以试一试其它的访问权限，例如，private，internal 看看能不能测试通过。结论是不能，请查看 6.3 节实现接口的原则。

其实这些结论大家通过动手实验就能得出结论。比如说如果实现了该接口的类的方法权限不是 public 或者方法返回值、参数类型、参数个数与接口中定义的不同，是否可以测试通过呢？如果在定义接口时在 function 前面加入了访问权限修饰符，可以不可以以呢？类似这些问题不需要查书或去问别人，自己动手做实验是最快最高效的学习方法，编译器会告诉你，行还是不行，直接问它就可以了！以上做法都行不通。

为了更好地保证接口的实现不出差错，通常最保险的做法就将该方法复制（ctrl + c）过来，并在前面加上 public，再去实现。

6.5 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。（就像人拥有一项本领）。

通过接口可以指明多个类需要实现的方法。（描述这项本领的共同接口）。

通过接口可以了解对象的交互界面，而不需要了解对象所对应的类。

我们通常所说的“继承”广义来讲，它不仅是指 extends ，还包括 implements，回想 5.2 节中说到多态存在的三个必要条件，这里面所说的就广义的继承。说得更明确一点就是：要有继承（或实现相同接口），要有重写（或实现接口），父类引用指向子类对象（或接口类型指向实现类的对象）。下面来看最后一个知识点，使用接口实现多态。

6.6 TestInterFacePoly.as —— 接口实现多态

```
package {
    public class TestInterFacePoly {
        public function TestInterFacePoly() {
            var cat:Cat = new Cat();
            var duck:Duck = new Duck();

            var racing:Racing = new Racing(cat);
            racing.go();
        }
    }
}

interface Runnable {
    function run():void;
}

interface Swimmable {
    function swim():void;
}

interface Flyable {
    function fly():void;
}

class Cat implements Runnable, Swimmable {
    public function run():void {
        trace("Cat run");
    }
```

```
    public function swim():void {
        trace("Cat swim");
    }

    public function climb():void {
        trace("Cat Climb");
    }
}

class Duck implements Runnable, Flyable {
    public function run():void {
        trace("Duck run");
    }

    public function fly():void {
        trace("Duck fly");
    }
}

class Racing {
    var runner:Runnable;
    public function Racing(r:Runnable) {
        runner = r;
    }
    public function go():void {
        runner.run();
    }
}
```

使用接口实现多态，和前面通过继承实现多态几乎是相同的，只不过这次是把父类引用改成了接口类型的引用。实现了同一接口的类的对象表示它们都具有这一项相同的能力，当我们只关心某些这项能力，而并不关心具体对象的类型时，使用多态可以更好地保证代码的灵活性，这就是向上抽象的作用。下面来解释一下这个例子。

首先，定义三个接口 Runnable（会跑的），Swimmable（会游的），Flyable（会飞的）。让 Cat 类实现 Runnable，Swimmable，让 Duck 类实现 Runnable，Flyable。实现了某个接口就代表拥有了某项（或几项）技能。Cat 类中除了实现这两个接口之外，它还有自己的 climb() 方法。

在测试类中，创建一个 Cat 类的对象 cat，一个 Dog 类的对象 dog。

接下来，加入一个 Racing（跑步比赛）类的实例，将 cat 传进去，赋给 runner 变量（Runnable 接口类型的引用），Racing 类的 go() 方法中调用了实现了 Runnable 接口的对象，并调用它的 run() 方法，这里就有了多态，动态绑定到实际对象的 run() 方法。

成员变量 runner 是 Runnable 接口类型的，说明我们只关心它是能跑的对象（拥有 run() 方法），具体它是怎么跑的我不管，反正你实现了 Runnable 接口，就肯定有 run() 方法，我只要你的 run() 方法，

其它的我不管。

注意，在 Runnable 中只定义 run() 方法，对于 runner 来说只能看到 Runnable 接口里定义的方法，在此接口以外的方法一律看不到，如果要让 runner 调用 Cat 对象的 climb() 或 swim() 方法是行不通的，与 5.3 节最后一段说明的道理是一样的。

接口在设计模式中应用广泛，下面请出策略模式。

6.7 策略模式（Strategy Pattern）

同样，不直接给出最终的答案，先看下面这个例子：

```
package {
    public class TestStrategy {
        public function TestStrategy() {
            var rabbit:Rabbit = new Rabbit();
            rabbit.run();
            rabbit.jump();
        }
    }
}

interface Runnable {
    function run():void;
}

interface Jumpable {
    function jump():void;
}

class Rabbit implements Runnable, Jumpable {
    public function run():void {
        trace("I can run fast");
    }
    public function jump():void {
        trace("I can jump 5m");
    }
}
```

这个例子很简单，让 Rabbit 实现 Runnable, Jumpable 接口，让它能跑能跳。

现在如果要让 Rabbit 跳不起来，那么就要修改它的 jump() 方法，打印出 “I can't jump”。如果要让它能跑 1000 m，并且还能跨栏，那么还要修改 run() 方法的实现。还记得 OO 设计的最根本原则吗？“开放—关闭”原则 —— 对添加开放，对修改关闭。下面来看看策略模式是怎样做到“开放—关闭”原则的。以下是 TestStrategy.as：

```
interface Runnable {
    function run():void;
```

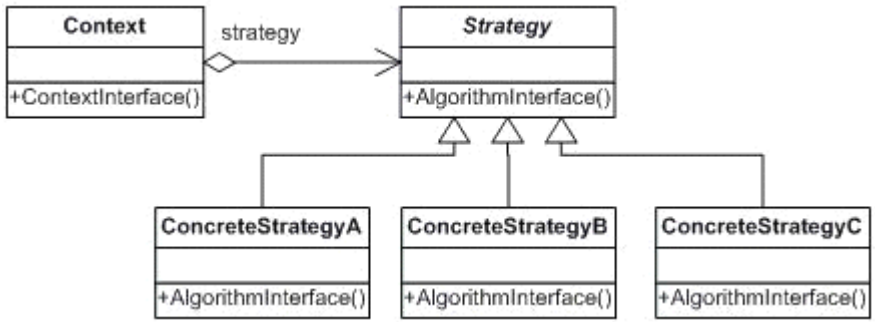
```
}

interface Jumpable {
    function jump():void;
}

class FastRun implements Runnable {
    public function run():void {
        trace("I can run fast");
    }
}

class JumpHigh implements Jumpable {
    public function jump():void {
        trace("I can jump 5m");
    }
}

class JumpNoWay implements Jumpable {
    public function jump():void {
        trace("I can't jump");
    }
}

class Rabbit {
    var runBehavior:Runnable = new FastRun();
    var jumpBehavior:Jumpable = new JumpHigh(); // new JumpNoWay();

    public function run() {
        runBehavior.run();
    }
    public function jump() {
        jumpBehavior.jump();
    }
}
```

现在 Rabbit 中加入了两个成员变量 runBehavior，jumpBehavior 分别是 Runnable 和 Jumpable 类型的引用，又是父类引用（接口）指向子类对象。Rabbit 的 run 和 jump 直接调用了 runBehavior.run() 和 jumpBehavior.jump()。而 runBehavior，jumpBehavior 指向的是两个实现了 Runnable 和 Jumpable 接口的类 FastRun 类和 JumpHigh 类。而在这两个类中分别实现了 Runnable 和 Jumpable 接口，run() 和 jump() 的具体实现被放到 FastRun 和 JumpHigh 这两个类中去了。

这样做有什么好处呢？首先，如果将来的策略发生了变化让兔子跳不起来，那么只需要添加一个新的类（策略）：JumpNoWay 同样让它实现 Jumpable 接口，jump 方法中打印出 "I can't jump"，然后将 Rabbit

类中的 new JumpHigh() 改为 new JumpNoWay() 即可，这样就实现了“添加而不是修改”的原则，我们只添加了一个新的策略（类），对原来的策略没有任何修改，最后只是替换了一个策略而以（当然这种修改是必要的）。另一个好处是，将来如果要修改 JumpHigh 的算法，让它可以跳 150 米，那么直接去修改 JumpHigh 里的 jump() 就可以，而不会影响到 Rabbit，从而降低了耦合度，这是封装算法所带来的好处。

这一切的灵活性都是多态所带来了，因此在很多的设计模式中都会用到多态，为的就是降低耦合度，增加程序的灵活性以及提高扩展性。以下是该模式的 UML 类图：



策略模式（Strategy Pattern）属于对象行为型模式，体现了两个非常基本的面向对象设计的基本原则：封装变化的概念；编程中使用接口，而不是对接口实现。策略模式的定义如下：

定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。每一个算法封装到具有共同接口的独立的类中，策略模式使这些算法在客户端调用它们的时候能够互不影响地变化。

策略模式使开发人员能够开发出由许多可替换的部分组成的软件，并且各个部分之间是弱连接的关系。弱连接的特性使软件具有更强的可扩展性，易于维护；

策略模式中有三个对象：

（1）环境对象：该类中实现了对抽象策略中定义的接口或者抽象类的引用。

（2）抽象策略对象：它可由接口或抽象类来实现。

（3）具体策略对象：它封装了实现不同功能的不同算法。

7.浅谈设计模式

前面我们已经介绍了两个设计模式以及一些面向对象设计（OOD）的原则，那么到底还有多少种原则呢？下面我们一起简单地了解一下：

1. 单一职责原则：一个类，最好只做一件事，只有一个引起它变化得原因。
2. 开放封闭原则：软件实体应当对修改关闭，对扩展开放。
3. 依赖倒置原则：依赖于抽象，而不要依赖于具体，因为抽象相对稳定。
4. 接口隔离原则：尽量应用专门的接口，而不是单一得总接口，接口应该面向用户，将依赖建立在最小得接口上。
5. 里氏替换原则：子类必须能够替换其基类。
6. 合成/聚合复用原则：在新对象中聚合已有对象，使之成为新对象的成员，从而通过操作这些对象达到复用得目的。合成方式较继承方式耦合更松散，所以应该少继承，多聚合。
7. 迪米特法则（又叫最少知识原则）：软件实体应该尽可能少的和其他软件实体发生相互作用。

设计原则是基本的工具，应用这些规则可使代码更加灵活、更容易维护，更容易扩展。基本原则：封装变化；面向接口变成而不是实现；优先使用组合而非继承。

对于开发人员而言，学习使用设计模式是很有必要的，而且越早学越好。尽早地了解它，能让我们脑子里先有这个概念，在日后写程序时或许就能联系上某个模式，可以对比一下人家的设计比我的设计强在哪儿，从而形成条件反射。

我在论坛上看到有人说“设计模式不用学，自己平时写写程序就会了”，楼下居然还有人跟帖表示赞同。设计模式最早由 GoF 提出，集结了这四位专家多年的心血才提炼出了 30 多种设计模式，难道说光凭你一个人的力量能够超越这些位专家经过多少年得到的成就？

作为一名知识的传播者，我认为讲授知识点是必要的，但更重要的是讲授学习的方法。比讲授学习方法更重要的是传授正确思考问题的方法。

3天学透 Actionscript（第三天）

8.面向对象程序设计

8.1 类和对象的概念

类：类是用来创建同一类型的对象的“模板”，在一个类中定义了该类对象所应具有的成员变量以及方法。

对象：对象是类的实例。

8.2 类之间的关系

系统中的类有那些关系：依赖、关联（聚合、合成）、泛化、实现。

1.依赖：对于外部类或对象的引用；

5.关联：关联暗示两个类在概念上位于相同的级别；

6.聚合：表示一种“拥有”关系，是两个类之间一种整体 / 局部的关系；

7.合成：表示一种更强“拥有”关系，就像人和腿的关系一样。组合而成的新对象对组成部分的内容分配和释放有绝对责任；

8.泛化：表现为继承 extends；

9.实现：表现为实现 implements。

8.3 面向对象程序设计（OOP）

在面向对象出现以前，结构化程序设计是程序设计的主流，结构化程序设计又称为面向过程的程序设计。这种设计方法开发的软件稳定性、可修改性和可重用性都比较差。

与过程相比对象是稳定的。面向对象的软件系统是由对象组成的，复杂的对象是由比较简单的对象组合而成的。也就是说，面向对象方法学使用对象分解取代了传统的功能分解。

面向对象的精髓在于考虑问题的思路是从现实世界人类思维习惯出发的，只要领会了这一点，就领会了面向对象的思维方法。万事万物皆为对象，大至日月星辰，小至沙粒微尘，都是对象。对象包容了一切事物，不仅仅是那些看得见摸得着的是实体，如：地球、汽车、树叶，还包括那些客观存在的事物，如：社会、互联网、朋友圈子等等，包罗万象。

以开车为例，用面向过程的思想去考虑，那么你先得知道怎么启动，怎么踩油门，怎么挂档。这些应该是司机的活，你要把这些步骤都实现出来。如果用面向对象的思想，把自己看成领导，只需要下达命令，告诉它你要去哪里就行了（例如，调用 drive() 方法），具体怎么开，怎么踩油门，怎么挂档，不需要我们去管。

那么 dirve() 这个方法放到车里是否合适呢，是不是应该放到“司机”类更合理呢？封装是很灵活的，没有对与错之分，只有好与更好，需要具体问题具体分析。因为 dirve() 方法要用到油门和车档，而这些东西都在车里面，因此如果将它封装到车这个类里面可能更好些。

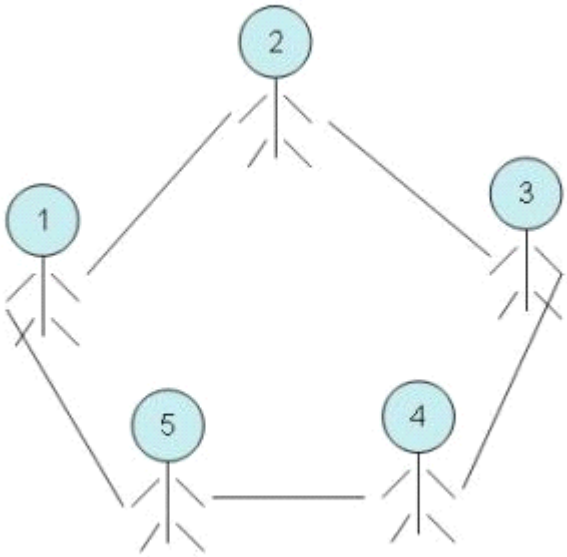
下面我们通过对比面向过程和面向对象的设计方式体会什么才是面向对象的思维。

8.4 出圈游戏 —— 面向过程 VS 面向对象

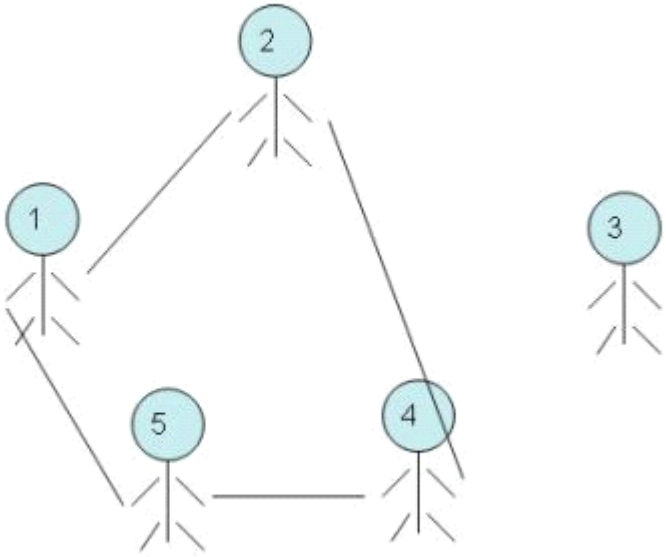
8.4.1 游戏规则

假设有 5 个小孩儿手拉手围成一圈。从第一个小孩儿开始以顺时针方向依次报数 —— “1，2，3”，报 3 的人出列，第四个人从 1 开始重新报数，报到 3 时再出列。如此下去，直到所有人全部出列为止，要求按照出列的顺序输出他们的序号。

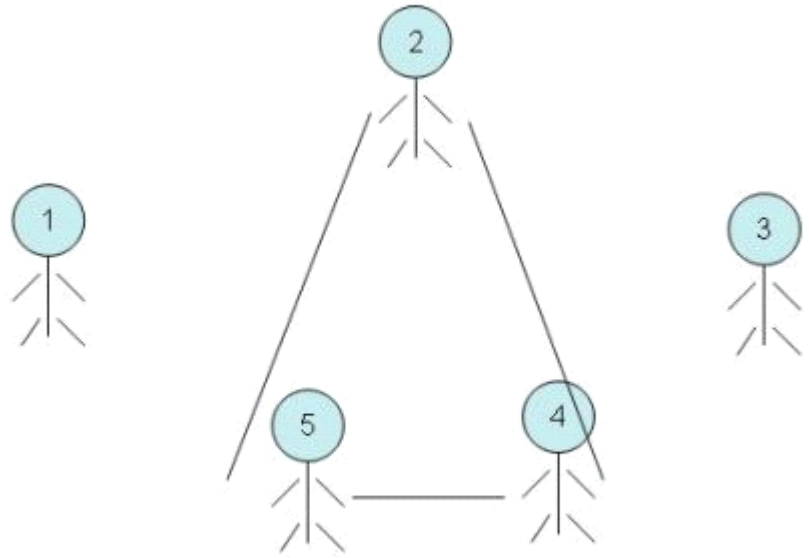
下面来看图理解，首先有 5 个小孩围成一个圈：



图一、 5 个小孩儿围成一圈



图二、 数到 3 的小孩儿退出去



图三、再从 2 号开始数三个人， 5 号退出，然后是 2 号，最后是 4 号。

最终输出的顺序应该是 3、1、5、2、4。

8.4.2 出圈游戏 —— 面向过程（circgame/ CircleGame1.as）

下面用面向过程的思想写这个程序，通过读注释先来看一下这个程序：

```
// 有 5 个小孩儿围成的圈
var array:Array = new Array(5);
for (var i = 0; i < array.length; i++) {
    // 如果元素值为 true 表示他在圈内，如果是 false 表示不在圈内
    array[i] = true;
}

// 圈内还剩多少人，最开始人都在，等于 array.length
var leftCount:int = array.length;
// 当前所报的数，初始为 0
var countNum:int = 0;
// 圈子的数组下标，表示当前指向的是谁
```

```
var index:int = 0;

while(leftCount > 0) {
    if (array[index] == true) {
        // 如果当前这个人在圈内则报数
        countNum++;
        if (countNum == 3) {
            // 如果所报的数是 3 则出列，剩余人数减1，并且下一次从新开始报数
            trace("out " + (index + 1));
            array[index] = false;
            leftCount--;
            countNum = 0;
        }
    }

    // 数组下标增加
    index++;
    if (index == array.length) {
        // 如果下标是最后一个位则归 0，因为这个圈是圆的
        index = 0;
    }
}
```

用 array 数组代表这个围成的圈，开始让圈数组中的每个元素都为 true，表示它们都在圈内，如果设为 false 则表示不在圈内，后面报数的时候就不予考虑了。

接下来定义三个变量分别表示圈内还剩多少人，所报的数是多少和数组下标。

下面 while 循环开始，只有圈内还有人（leftCount > 0）就进行循环，首先判断当前 index 所指的元素是否为真，如果是则报数加 1，再判断是不是加到 3 了，如果是则打印出当前的数组下标，再将该元素设为 false，剩余人数减1，下一次从新开始报数。

最后让数组下标加 1，当指到最后时，将数组下标置为 0，因为这是一个圈，要用循环的数组来表示。

8.4.3 出圈游戏 —— 面向对象（cirgame/ CircleGame2.as）

回顾上一个例子，在面向过程的程序中，明明是围成的一个圈儿，却要看成是一个数组；明明是一个个小孩儿却要看成是数组的一个个元素。这不就是为了让计算机看懂吗？但是，面向对象是更加接近人类的思维模式，我们在现实中看到的就是一个个小孩儿，怎么能说是数组？那么这一个个小孩儿就是一个个对象，他们都是 Kid。围成的这个圈，就是一个 KidCircle。很自然吧，比大自然还自然！下面来体会面向对象的设计思想：

```
package cirgame {
    public class CircleGame2 {
        public function CircleGame2() {
            var kc:KidCircle = new KidCircle(5);
            var countNum:int = 0;
```

```
            var k:Kid = kc.head;

            while (kc.count > 0) {
                countNum++;
                if (countNum == 3) {
                    trace(k.id + 1);
                    kc.remove(k);
                    countNum = 0;
                }
                k = k.right;
            }

        }
    }
}

// 每个 Kid 都有自己的 id，并且左右手还拉着其它的 Kid
class Kid {
    var id:uint;
    var left:Kid;
    var right:Kid;
}

// 这个圈子里可以加入或移除一些 Kid
class KidCircle {
    var count:uint = 0;
    var head:Kid;
    var rear:Kid;

    function KidCircle(n:uint) {
        for (var i = 0; i < n; i++) {
            var kid:Kid = new Kid();
            add(kid);
        }
    }

    function add(kid:Kid):void {
        kid.id = count;
        if (count == 0) {
            head = kid;
            rear = kid;
            kid.left = kid;
```

```
            kid.right = kid;
        } else {
            rear.right = kid;
            kid.left = rear;
            kid.right = head;
            head.left = kid;
            rear = kid;
        }
        count++;
    }

    function remove(kid:Kid):void {
        if (count <= 0) {
            return;
        } else if (count == 1) {
            head = rear = null;
        } else {
            kid.left.right = kid.right;
            kid.right.left = kid.left;
            if (kid == head) {
                head = kid.right;
            } else if (kid == rear) {
                rear = kid.left;
            }
        }
        count--;
    }
}
```

这段程序中设计了两个类，代表两类客观事物 —— 小孩（Kid）和圈子（KidCircle）。从 8.4.1 的图中可以确切地看到。Kid 有三个属性：id 号、左手和右手，左手拉着一个 Kid，右手拉着一个 Kid，因此 left 和 right 存放两个 Kid 的引用。

下面是 KidCircle 类，代表围成的圈子，这个圈子可以加入或移除一些 Kid，因此有 add 和 remove 两个方法。head 和 rear 两个成员变量用于指向队首和队尾的两个 Kid，因为些添加的 Kid 要放在队尾（rear）的后面，因为这是一个圈子，所以还需要让队尾的小孩儿拉住队首（head）的小孩儿，因此需要保存这两个成员变量。

最后是测试类，主要的逻辑和前一个例子比较像，这里就不多解释了。

通过这个例子大家可以看出，我们无形之间就完成了一个数据结构 —— 双向循环链表。而前面面向过程的例子，实际上就是一个顺序的存储结构 —— 线性表。

后面，我会带大家写一个贪吃蛇的游戏，目的是学习面向对象编程的思想（并非该游戏本身），在贪吃蛇中我们就会运用到类似于单向链表的结构，如果双向链表掌握了，那么单向一定没问题。

通过本节请大家认真体会面向对象的设计思想。一次学会，终身受用。