**面向对象编程：类和对象**

**7.2 面向对象的基本概念**

面向对象的基本概念主要有类、封装、接口和对象

1. 类（Class）

面向对象编程也采用了类的概念，它把事物编写成一个个“类”。在类中，用数据表

示事物的属性，用函数实现事物的行为，数据和事物是一个统一的整体，这就使编程方式和人的思维方式保持一致，极大的降低了思维难度。

1. 封装（Encapsulation）

面向对象技术把事物的属性和行为的实现细节封装在类中，形成一个可以重复使用的“零件”，便于重复使用而不需要知道实现细节。

对象构造器

一般情况下我们先创建对象，然后逐个初始化对象的变量。

Cat doraemon = new Cat();

doraemon.name = "Doraemon";

doraemon.age = 8;

C#3.0 中加入新特性——对象构造器，使得对象的初始化工作变得格外简单，我们可以采用类似于数组初始化的方式来初始化类的对象。比如：

Cat doraemon = new Cat { name = "Doraemon", age = 8 };

**7.5 属性**

原则上我们可以随时访问类的公有成员，但这种不加限制的访问可能会造成问题，比如

catkitty = newcat();

kitty.age = -10;

C#设计了一种特殊的语法——属性（Property）。在属性中，定义了 get 和 set 两个访问器，get 访问器用来读取变量的值，set 访问器用来设置变量的值，合法性检查的代码定义在两个访问器中。需要提醒的是 set 访问器没有声明显式参数，但它有一个名为 value 的隐式参数。属性的运行方式和方法相似，因此属性可以看作特殊的方法，但属性的使用方式和变量完全相同。**无论何时使用属性，都会在后台隐式地调用 get 访问器或 set 访问器，并执行访问器中的代码**。

每个属性背后都对应着一个变量，我们一般让属性和它所对应的变量同名，只是将首

字母大写，以示区别。比如变量 age 相对应属性 Age。当然理论上我们可以给属性起一个

毫不相干的名字，但这样做显然不利于阅读理解。

自动属性

有时候属性很简单，get 和 set 里面只有取值和赋值，没有其它逻辑代码，这种属性可以通过自动属性来快速定义。

**public int Name { get; set;}**

自动属性包含两层含义，先创建一个私有变量，然后创建该私有变量的属性

**7.6 构造函数**

1．默认构造函数

当创建对象时，系统会调用默认的构造函数，用默认值初始化所有成员变量。比如整型变量初始化为 0，实型变量初始化为 0.0，布尔型变量初始化为 false，字符串变量初始化为null（空），等等。

构造函数总是存在的，即使没有显式定义，编译器也会为类分配一个默认构造函数。

3．无参数构造函数

当我们自定义了构造函数后，默认构造函数就失效了，要想继续使用无参数的构造函

数，必须显式定义。 当自定义了无参数的构造函数后，系统会用自定义的构造函数代替默认的构造函数，用自定义的构造函数中的代码初始化变量。

**7.7 析构函数和垃圾回收**

由于 C#设计了非常完善的垃圾回收机制，一般也不用向析构函数里添加代码。析构函数通常用来释放对象使用的非托管资源。在大型的程序中，有时有些对象虽然不再使用了，

但离作用域结束还有相当长的时间，在这期间，对象仍然占用内存，浪费资源。总之在 C#

中删除对象的工作是由垃圾回收器负责完成，析构函数通常用来释放对象使用的非托管资

源。需要指出的是垃圾回收器的运行时间具有不确定性，我们无法预计垃圾回收器什么时

候运行，因此也无法预测什么时候会删除无用对象，有可能立即删除，也可能要过一段时

间才删除。我们可以通过调用 System.GC.Collect()方法强迫垃圾回收器运行（会对性能产

生一定影响），但即使调用了该方法，我们也无法预知垃圾回收器具体在何时运行。

**7.8 实例演练：T im e类**

**8.2 静态成员**

8.2.1 静态变量

class wolf

{

Public static int wolfcount = 0;

Public wolf() { wolfcount++; }

~wolf() { wolfcount--; }

Public void act(){

if (wolfcount> 5) {

*Console*.*WriteLine*("wolf count = {0},attack lion!", wolfcount); }

}

publicvoidsam(){

wolf[] wolves = newwolf[5];

for (inti = 0; i<wolves.*Length*; i++){

wolves[i] = newwolf();

}// 此处 wolfcount==5

}

}

**8.3常量**

8.3.1 const常量

const 常量只能在声明的时候初始化，不能在其它地方赋值，运行过程中它的值保持不变。特别需要注意的是，类的const常量成员是隐式静态的。比较奇怪的是，虽然const常量默认是静态的，但不能用static关键字显式声明。

8.3.2 readonly常量

const 常量是隐式静态的，为同一个类的全部对象所共有，即同一个类的所有对象具有

相同的值。readonly 常量，每个对象可以有不同的值。

**8.4 重载（O verload）**

重载运算符由关键字 operator 声明，必须定义为静态。

class Complex

{

public static Complex operator +(Complex z1, Complex z2) { return new Complex(); }

}

编译器是如何识别这些重载的运算符呢？对了，和方法重载类似，重载运算符的调用

原则也是参数的“最佳匹配”，即系统是根据左右两个操作数的类型选择调用哪个版本的

运算符。

**8.6 索引**

索引可以让我们像数组那样访问类的数据成员。下面我们定义一个“立方体类”，它有长、宽、高三个属性。

索引的定义方式如下：

public double this[int index]

索引的函数体与属性类似，也是用 get 和 set 访问器。 get 访问器用于获取成员变量的值， set 访问器用于为成员变量赋值。

class Cube

{

private double length;

private double width;

private double height;

public double this[int index]

{

get

{

switch (index)

{

case 0:

return length;

case 1:return width;

case 2:return height;

default:

return 0;

}

}

set

{

switch (index)

{

case 0:

length = value;

break;

case 1:

width = value;

break;

case 2:

height = value;

break;

default:

throw new *IndexOutOfRangeException*("下标出界！");

}

}

}

}

索引的使用方法和数组完全一样，如果我们创建了一个名为 box 的 Cube 对象，就可

以用 box[0] ， box[1] ， box[2] 分别表示立方体的长、宽、高了。

在数组中，下标只能为整数，在索引中，有了更灵活的选择，既可以为 int 型，也可

以为 double 、 string 等类型。

**8.7 值类型和引用类型**

void test(){

Cat kitty; kitty.strname = "cat third"; // 使用了未经赋值的局部变量

}

为什么会出现这样的结果呢？原来语句“ Cat kitty ”只是声明了一个对象的名称，仅仅在栈上创建了一个引用符而已，并没有真正的在堆中创建对象，对象的各个成员是不存

在的，引用符中存储的是空地址。只有使用 new 运算符后，对象才真正在堆中创建，才能

使用它的成员。

当对象没有实例化时，引用符的值为 null （空），当对象已经实例化时，引用符的值就是该对象在堆中的地址。一般情况下，我们没有必要区分引用符和对象，比如我们提到 kitty ，就是指 kitty 所指向的对象。但一个引用符先后指向不同对象时，就有必要把引用符和对象区分清楚。

void test()

{

cat someone; //声明类型为 Cat 的引用符

cat kitty = new cat("Kitty", 3); //创建对象 kitty

cat garefield = new cat("Garefield ", 5); //创建对象 garfield

//让引用符 someone 指向对象 kitty

someone = kitty;

*Console*.*WriteLine*("嗨，我是{0},我有{1}岁了。", someone.name, someone.age);

//让引用符 someone 指向对象 Garifiel

someone = garefield;

*Console*.*WriteLine*("嗨，我是{0},我有{1}岁了。", someone.name, someone.age);

}

**8.9 对象的相等**

**ReferenceEquals()** 方法是一个静态方法，用来测试两个引用符是否指向同一个对象

**实例版** **Equals()** 方法是一个虚方法，只进行引用比较。但我们一般会在派生类中重写该方法以实现值比较，如果两个对象是值类型，则判断两个值是否相等；如果两个对象是引用型，则判断两个对象的数据成员是否相等。

**静态版 Equals()** 方法的功能与实例版基本一样，实际上静态版 Equals() 方法通过调用实例版方法进行比较，只是在调用“ objA.Equals(objB) ”前，会先检查两个参数是否均为空引用，如果均为空引用，返回 true ，如果一个为空，一个不为空，返回 false 。

**相等运算符==**。默认状态下，若两个对象为值类型，相等运算符“ == ”比

较两个对象的值；若两个对象为引用类型，相等运算符“ == ”比较两个引用符。但我们可

以通过重载运算符的方法加以改变。

**8.10 以对象为元素的数组**

声明以对象为元素的数组的方式和普通数组相同：

Cat[] cats = new Cat[5];

但这只是声明了一组“引用符”而已，并没有真正创建对象。下面的语句才具体创建对象，并让每个引用符指向一个对象。

for (int i = 0; i < cats.*Length*; i++)

{

cats[i] = new Cat("cat" + i, i);

}

**8.11 匿名类型**

匿名类型只能由一个或多个公共只读属性组成，不能包含其它种类的类成员。

var city = new { *Name* = "Beijing", ZipCode = 100000 };

*Console*.*WriteLine*("Name={0}\nZipCode={1}", city.*Name*, city.ZipCode);

**8.12 扩展方法**

很多时候我们需要在已经编译好的类中添加新的功能，比如在 Double 类中添加一个

计算次方的方法 Power() 。你可能会想到去创建一个派生类，但使用起来往往不太方便。

现在好了， C#3.0 为提供了一个非常漂亮的解决方案——扩展方法。

namespace ExtensionMethods

{

public static class MyExtensions

{

//扩展方法：求实数的次方

public static double Power(this double x, int exp)

{

double result = 1;

for (int i = 1; i <= exp; i++)

{

result \*= x;

}

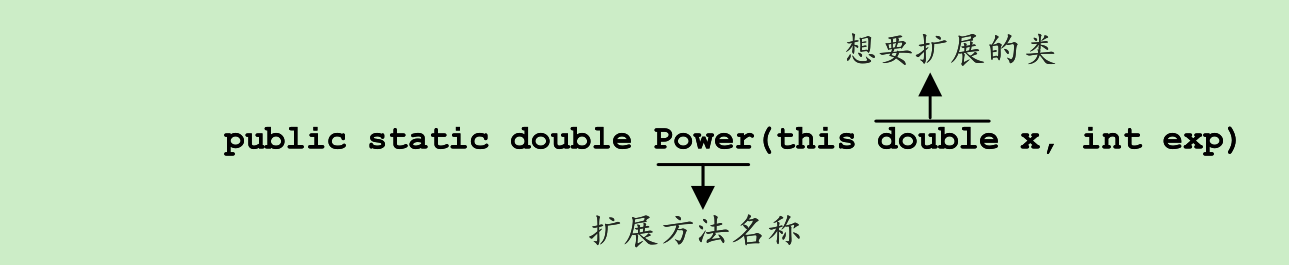
return result;

}

}

}

声明扩展方法的语法如下：



扩展方法的第一个参数的类型就是该方法想要扩展的类，必须用 this 关键字修饰。扩

展方法的是静态方法，但调用它的时候只能用对象来调用。

//调用扩展方法

using *System*;

using ExtensionMethods; //注意引用扩展方法所在的命名空间

namespace HelloTomorrow

{

class Program

{

static void Main(string[] args)

{

double x = 2.478;

double y = x.Power(3);

*Console*.*WriteLine*("{0}的3次方为{1}。", x, y);

}

}

}