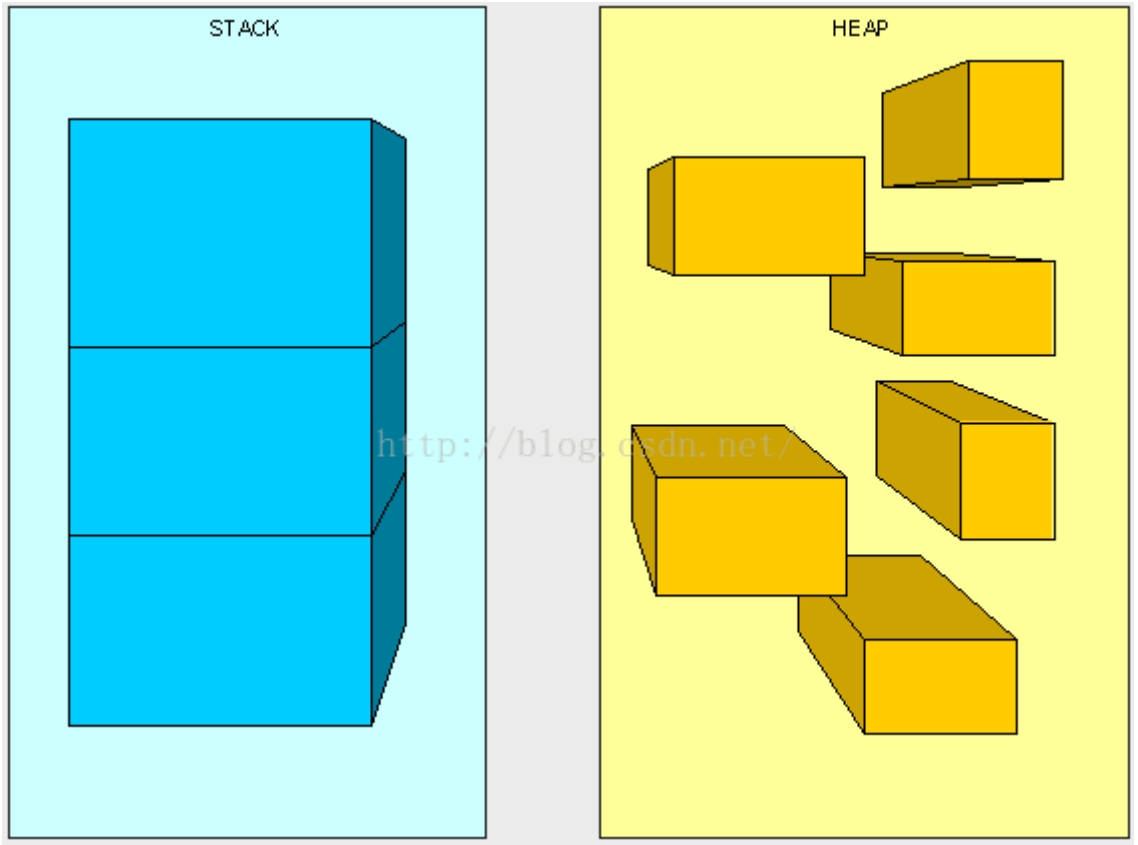
尽管在.NET framework下我们并不需要担心内存管理和垃圾回收(GarbageCollection)，但是我们还是应该了解它们，以优化我们的应用程序。同时，还需要具备一些基础的内存管理工作机制的知识，这样能够有助于解释我们日常程序编写中的变量的行为。在本文中我将讲解栈和堆的基本知识，变量类型以及为什么一些变量能够按照它们自己的方式工作。

在.NET framework环境下，当我们的代码执行时，内存中有两个地方用来存储这些代码。假如你不曾了解，那就让我来给你介绍栈(Stack)和堆(Heap)。栈和堆都用来帮助我们运行代码的，它们驻留在机器内存中，且包含所有代码执行所需要的信息。

* 栈vs堆：有什么不同？

栈负责保存我们的代码执行（或调用）路径,而堆则负责保存对象（或者说数据，接下来将谈到很多关于堆的问题）的路径。

可以将栈想象成一堆从顶向下堆叠的盒子。当每调用一次方法时，我们将应用程序中所要发生的事情记录在栈顶的一个盒子中,而我们每次只能够使用栈顶的那个盒子。当我们栈顶的盒子被使用完之后，或者说方法执行完毕之后，我们将抛开这个盒子然后继续使用栈顶上的新盒子。堆的工作原理比较相似，但大多数时候堆用作保存信息而非保存执行路径，因此堆能够在任意时间被访问。与栈相比堆没有任何访问限制，堆就像床上的旧衣服，我们并没有花时间去整理，那是因为可以随时找到一件我们需要的衣服，而栈就像储物柜里堆叠的鞋盒，我们只能从最顶层的盒子开始取，直到发现那只合适的。



以上图片并不是内存中真实的表现形式，但能够帮助我们区分栈和堆。

栈是自行维护的，也就是说内存自动维护栈，当栈顶的盒子不再被使用，它将被抛出。相反的，堆需要考虑垃圾回收，垃圾回收用于保持堆的整洁性，没有人愿意看到周围都是赃衣服，那简直太臭了！

* 栈和堆里有些什么？

当我们的代码执行的时候，栈和堆中主要放置了四种类型的数据：值类型(Value Type)，引用类型(Reference Type)，指针(Pointer)，指令(Instruction)。

1.值类型：

在C#中，所有被声明为以下类型的事物被称为值类型：

bool

byte

char

decimal

double

enum

float

int

long

sbyte

short

struct

uint

ulong

ushort

2.引用类型：

所有的被声明为以下类型的事物被称为引用类型：

class

interface

delegate

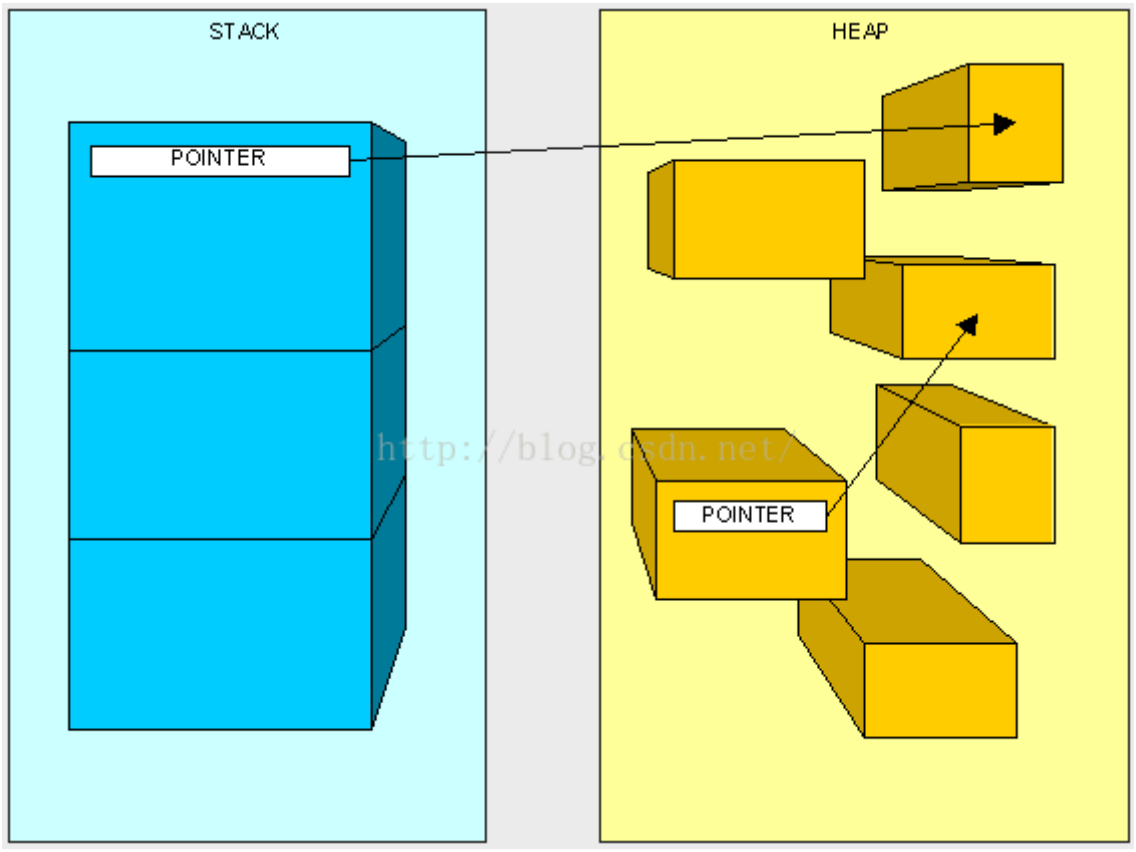
object

string

3.指针：

在内存管理方案中放置的第三种类型是类型引用，引用通常就是一个指针。我们不会显示的使用指针，它们由公共语言运行时（CLR）来管理。指针（或引用）是不同于引用类型的，

是因为当我们说某个事物是一个引用类型时就意味着我们是通过指针来访问它的。指针是一块内存空间，而它指向另一个内存空间。就像栈和堆一样，指针也同样要占用内存空间，但它的值是一个内存地址或者为空。



4.指令：

在后面的文章中你会看到指令是如何工作的...

* 如何决定放哪儿?

这里有一条黄金规则:

1. 引用类型总是放在堆中。(够简单的吧?)
2. 值类型和指针总是放在它们被声明的地方。(这条稍微复杂点，需要知道栈是如何工作的，然后才能断定是在哪儿被声明的。)

就像我们先前提到的，栈是负责保存我们的代码执行(或调用)时的路径。当我们的代码开始调用一个方法时,将放置一段编码指令(在方法中)到栈上,紧接着放置方法的参数，然后代码执行到方法中的被“压栈”至栈顶的变量位置。通过以下例子很容易理解...

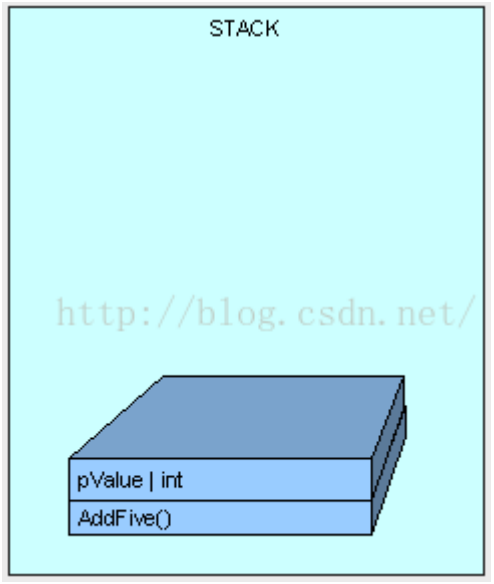
下面是一个方法（Method）:

```
public int AddFive(int pValue)
{
        int result;
        result = pValue + 5;
```

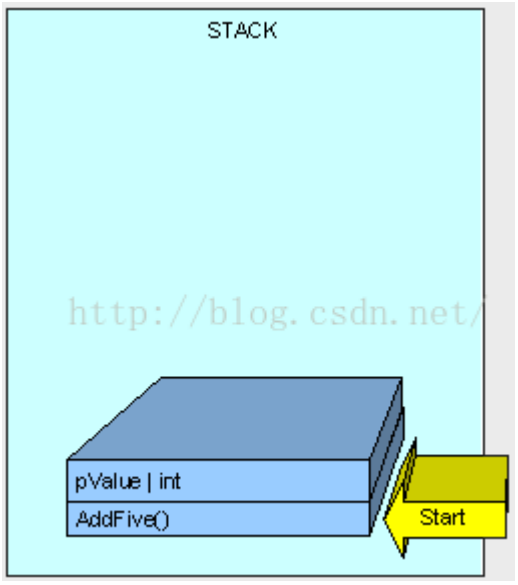
```
        return result;
    }
```

现在就来看看在栈顶发生了些什么，记住我们所观察的栈顶下实际已经压入了许多别的内容。

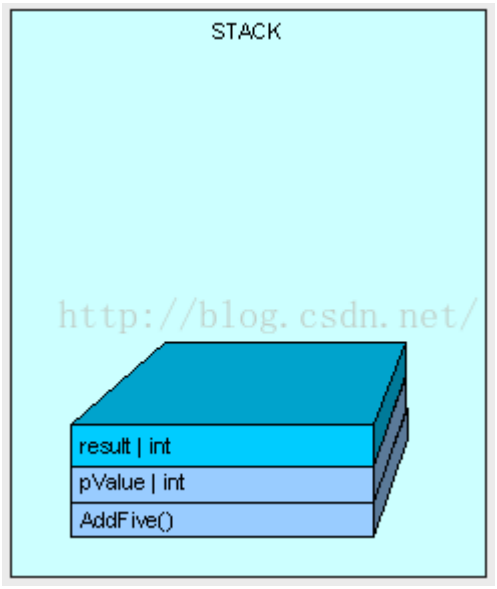
首先方法（只包含需要执行的逻辑字节，即执行该方法的指令，而非方法体内的数据）入栈，紧接着是方法的参数入栈。（我们将在后面讨论更多的参数传递）



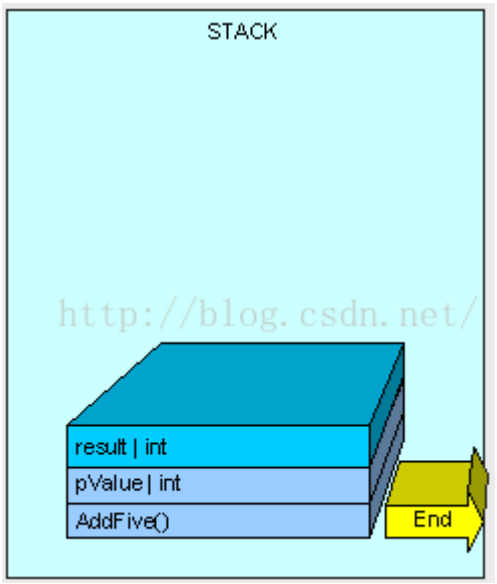
接着，控制（即执行方法的线程）被传递到堆栈中AddFive()的指令上，



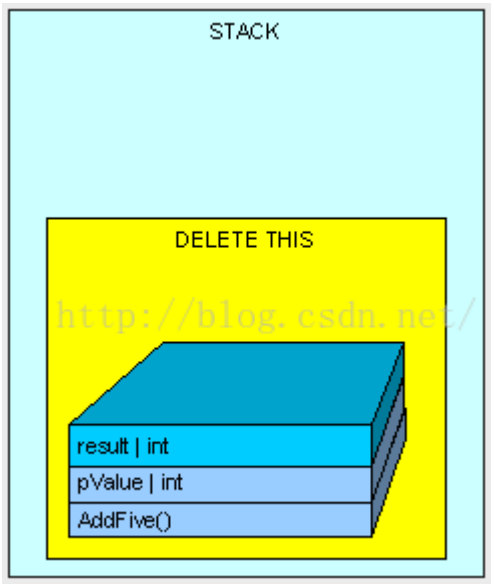
当方法执行时，我们需要在栈上为“result”变量分配一些内存，



Themethod finishes execution and our result is returned.
方法执行完成，然后方法的结果被返回。



通过将栈指针指向AddFive()方法曾使用的可用的内存地址，所有在栈上的该方法所使用内存都被清空，且程序将自动回到栈上最初的方法调用的位置(在本例中不会看到)。



在这个例子中，我们的"result"变量是被放置在栈上的，事实上，当值类型数据在方法体中被声明时，它们都是被放置在栈上的。
值类型数据有时也被放置在堆上。记住这条规则--值类型总是放在它们被声明的地方。好的，如果一个值类型数据在方法体外被声明，且存在于一个引用类型中，那么它将被堆中的引用类型所取代。

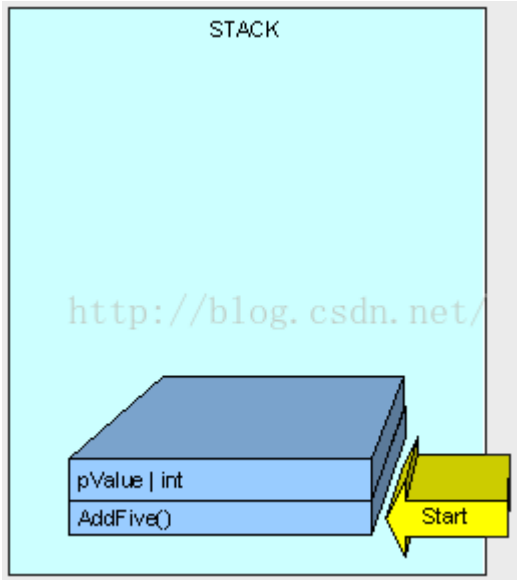
来看另一个例子：
假如我们有这样一个MyInt类（它是引用类型因为它是一个类类型）：

```
public class MyInt
{
        publicint MyValue;
 }
```

然后执行下面的方法:

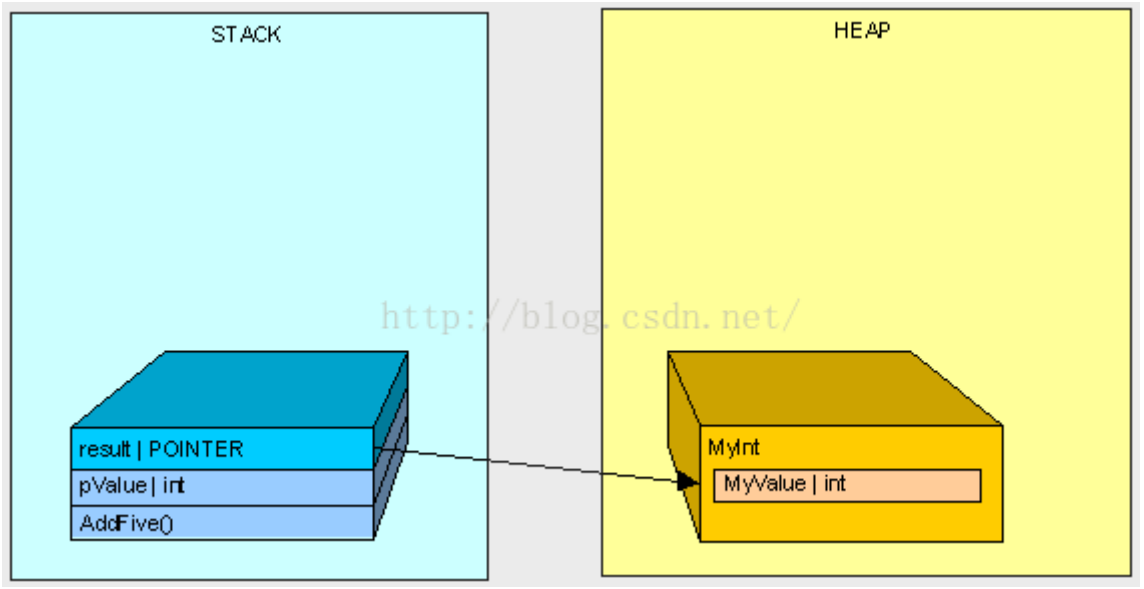
```
public MyInt AddFive(int pValue)
 {
            MyInt result = new MyInt();
            result.MyValue = pValue + 5;
            return result;
 }
```

就像前面提到的，方法及方法的参数被放置到栈上，接下来，控制被传递到堆栈中AddFive()的指令上。

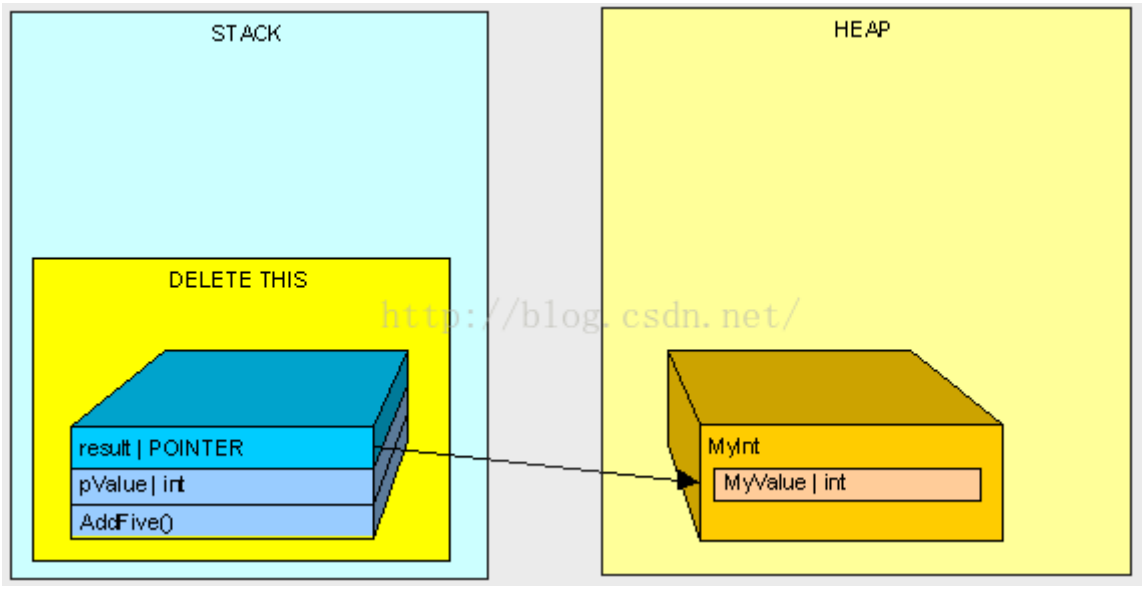


接着会出现一些有趣的现象...

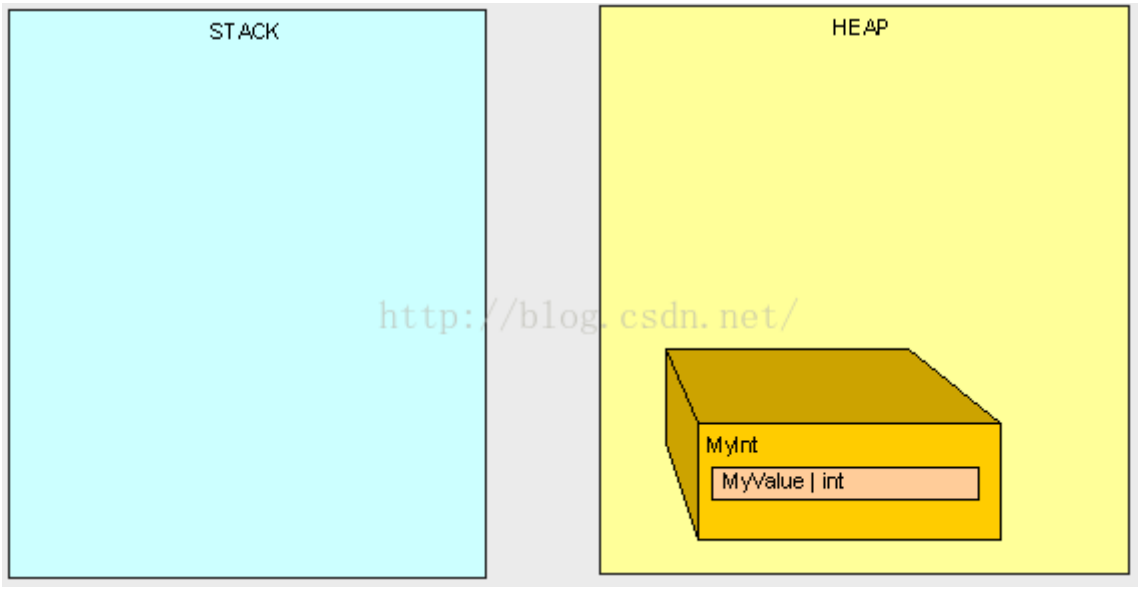
因为"MyInt"是一个引用类型,它将被放置在堆上,同时在栈上生成一个指向这个堆的指针引用。



在AddFive()方法被执行之后，我们将清空...



我们将剩下孤独的MyInt对象在堆中（栈中将不会存在任何指向MyInt对象的指针!）



这就是垃圾回收器（后简称GC）起作用的地方。当我们的程序达到了一个特定的内存阀值，我们需要更多的堆空间的时候，GC开始起作用。GC将停止所有正在运行的线程，找出在堆中存在的所有不再被主程序访问的对象，并删除它们。然后GC会重新组织堆中所有剩下的对象来节省空间，并调整栈和堆中所有与这些对象相关的指针。你肯定会想到这个过程非常耗费性能，所以这时你就会知道为什么我们需要如此重视栈和堆里有些什么，特别是在需要编写高性能的代码时。

Ok...这太棒了，当它是如何影响我的？

Goodquestion.

当我们使用引用类型时，我们实际是在处理该类型的指针，而非该类型本身。当我们使用值类型时，我们是在使用值类型本身。听起来很迷糊吧？

同样，例子是最好的描述。

假如我们执行以下的方法：

```
public int ReturnValue()
 {
            int x = new int();
            x = 3;
            int y = new int();
            y = x;
            y = 4;
            return x;
 }
```

我们将得到值3，很简单，对吧？

假如我们首先使用MyInt类

```
public class MyInt
 {
            public int MyValue;
 }
```

接着执行以下的方法：

```
public int ReturnValue2()
 {
            MyInt x = new MyInt();
            x.MyValue = 3;
            MyInt y = new MyInt();
            y =x;
            y.MyValue =4;
            return x.MyValue;
 }
```

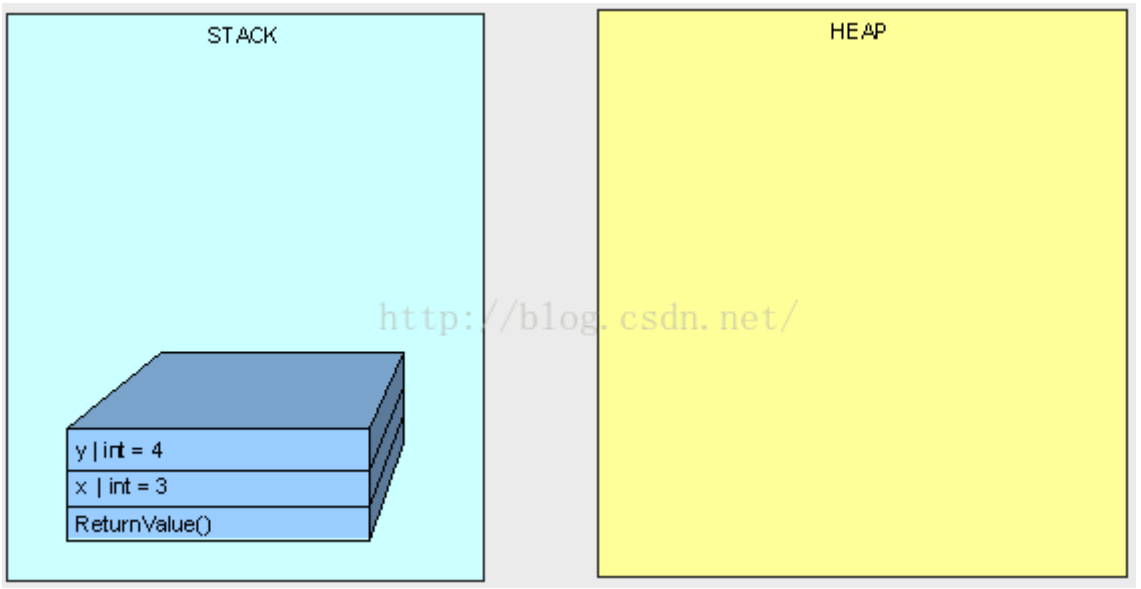
我们将得到什么？...　　4!

为什么？...　x.MyValue怎么会变成4了呢？...　看看我们所做的然后就知道是怎么回事了：

在第一例子中，一切都像计划的那样进行着：

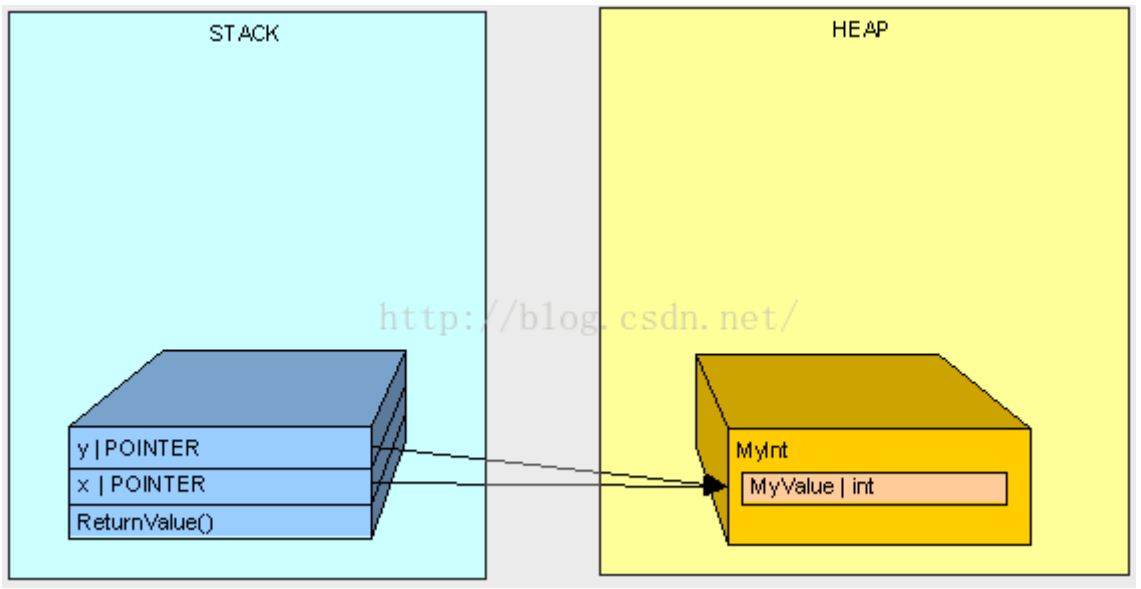
```
public int ReturnValue()
 {
            int x = 3;
            int y = x;
            y = 4;
```

```
        return x;
    }
```



在第二个例子中，我们没有得到"3"是因为变量"x"和"y"都同时指向了堆中相同的对象。

```
    public intReturnValue2()
    {
        MyInt x;
        x.MyValue = 3;
        MyInt y;
        y =x;
        y.MyValue = 4;
        return x.MyValue;
    }
```



希望以上内容能够使你对C#中的值类型和引用类型的基本区别有一个更好的认识，并且对指针及指针是何时被使用的有一定的基本了解。在系列的下一个部分，我们将深入内存管理并专门讨论方法参数。