زنگ سی‌شارپ – قسمت بیست و هفتم

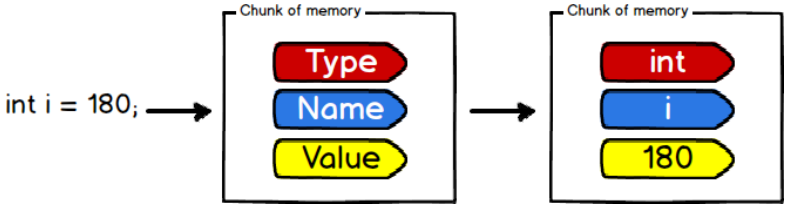
نوشته‌ی مسعود درویشیان

لینک مستقیم این مطلب در وب‌تارگت

در قسمت قبل با return کردن object و overloading آشنا شدید. در این قسمت از زنگ سی‌شارپ قصد داریم به مباحث مهم stack، heap، value types، reference types، boxing و unboxing بپردازیم و همچنین garbage collection، object initializers، optional arguments و named arguments را مورد بحث و بررسی قرار دهیم.

هنگامی‌که یک متغیر تعریف می‌کنید، دقیقاً چه اتفاقی می‌افتد؟

هنگامی‌که شما در اپلیکیشن‌های .NET یک متغیر تعریف می‌کنید، قسمتی از حافظه‌ی RAM برای این منظور اختصاص داده می‌شود. این قسمت از حافظه، شامل سه چیز است: نام متغیر، data type متغیر و مقدار متغیر.

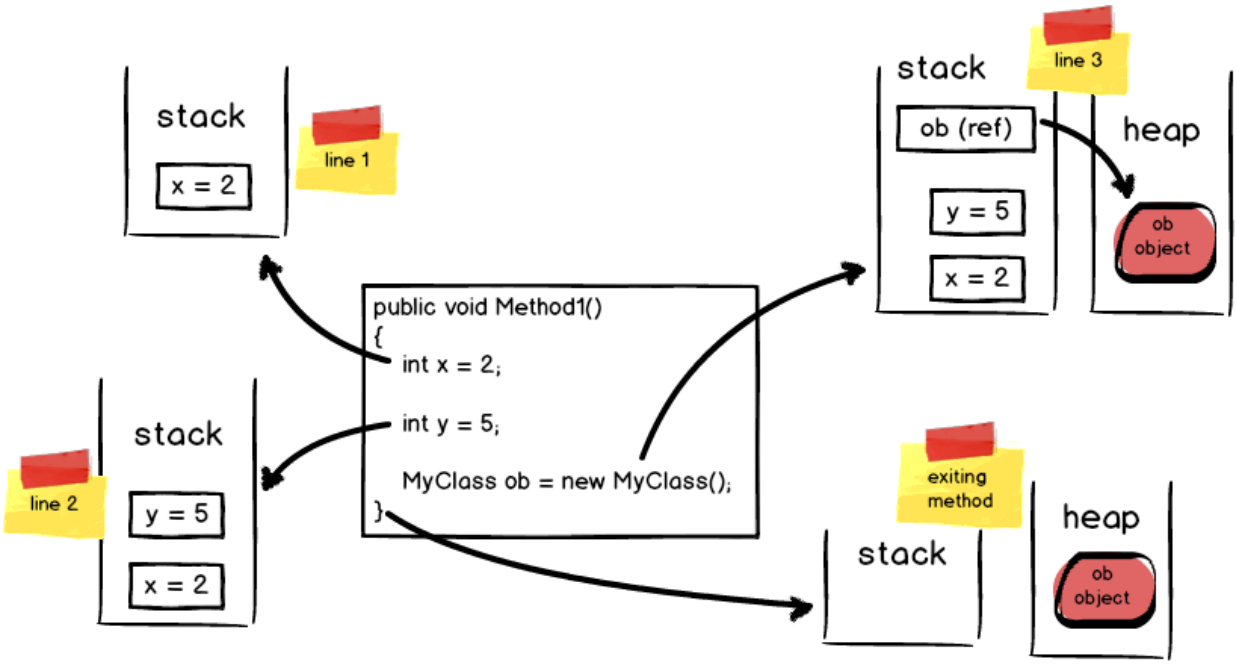


با توجه به data type، متغیر شما در قسمت‌های متفاوتی ذخیره می‌شود. دو نوع تخصیص حافظه وجود دارد که یکی stack memory و دیگری heap memory است. برای اینکه بهتر با stack و heap آشنا شوید به کد زیر و شرح آن توجه کنید:

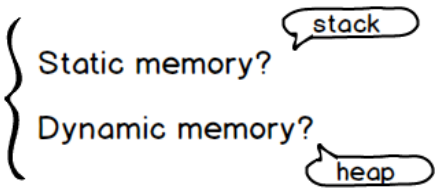
```
public void Method1()
{
    // line 1
    int x = 2;

    // line 2
    int y = 5;

    // line 3
    MyClass ob = new MyClass();
}
```

هنگامی‌که line 1 اجرا می‌شود، کامپایلر مقدار کمی از حافظه را در stack برای این منظور اختصاص می‌دهد. stack مسئول پیگیری حافظه‌ی مورد نیاز (در حال اجرا) در اپلیکیشن شما است. همان‌طور که پیش از این با نحوه‌ی ذخیره‌سازی اطلاعات در stack آشنا شدید، stack عملیات Last In First Out را اجرا می‌کند و هنگامی که line 2 اجرا می‌شود، متغیر y در بالای stack ذخیره خواهد شد. در line 3 ما یک شیء به‌وجود آورده‌ایم و در این‌جا اندکی داستان متفاوت می‌شود.

پس از این‌که line 3 اجرا شد، متغیر ob در stack ذخیره می‌شود و شیءای که ساخته شده در heap قرار می‌گیرد. نکته دقیقاً همین‌جاست که reference ها در stack ذخیره می‌شوند و عبارت MyClass ob حافظه را برای یک شیء از این کلاس اشغال نمی‌کند. این عبارت تنها متغیر ob را در stack قرار می‌دهد (و به آن مقدار null می‌دهد) و هنگامی‌که کلمه‌ی کلیدی new اجرا می‌شود، شیء این کلاس در heap ذخیره خواهد شد. در نهایت هنگامی‌که برنامه به انتهای متد می‌رسد، متغیرهایی که در stack بودند همه‌گی پاک می‌شوند. توجه کنید که پس از به پایان رسیدن متد چیزی از heap پاک نمی‌شود بلکه اشیای درون heap بعداً توسط garbage collector پاک خواهند شد. در مورد garbage collector در انتهای این مقاله صحبت خواهیم کرد.



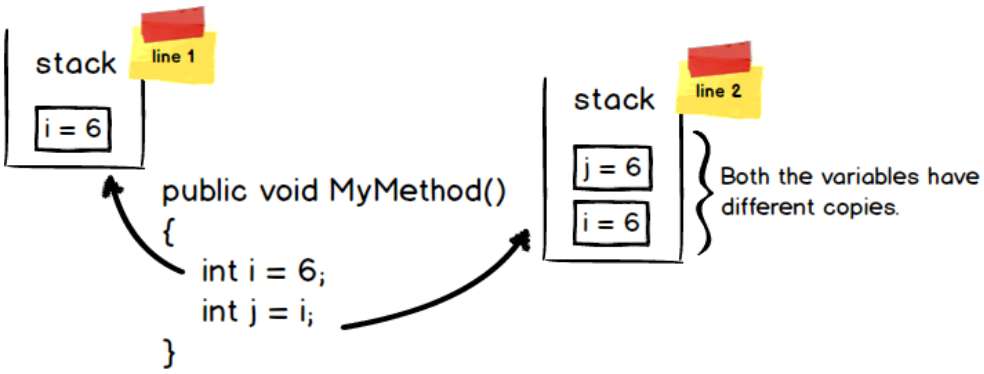
ممکن است برای‌تان سوال باشد که چرا stack و heap ؟ نمی‌شود همه در یک‌جا ذخیره شوند؟ اگر با دقت نگاه کنید می‌بینید که data type های اصلی (value types)، پیچیده و سنگین نیستند. آن‌ها مقادیر تکی مثل int i = 5 را نگه می‌دارند در حالی‌که object data types یا reference types پیچیده‌تر و سنگین‌تر هستند، آن‌ها به اشیای دیگری رجوع می‌کنند. به عبارت دیگر، آن‌ها به چندین مقدار رجوع می‌کنند (زیرا اشیاء می‌توانند شامل مقادیر زیادی از فیلد و متد و... باشند) که هرکدام از آن‌ها باید در حافظه ذخیره شده باشد. اشیاء به memory dynamic و data type های اصلی (value types) به static memory نیاز دارند. اگر اطلاعات شما نیازمند dynamic memory باشد، در heap ذخیره می‌شود، اگر نیازمند static memory باشد، در stack ذخیره خواهد شد.



Reference types و Value types

اکنون که با مفاهیم stack و heap آشنا شدید بهتر می‌توانید مفهوم value types و reference types را درک کنید. Value type ها تمام و کمال در stack ذخیره می‌شوند، یعنی هم مقدار و هم متغیر همه‌گی یک‌جا هستند اما در reference type متغیر در stack است درحالی‌که object در heap قرار می‌گیرد و متغیر و شیء به هم متصل می‌شوند (متغیر به شیء اشاره می‌کند).

در زیر، data type ای از جنس int داریم با اسم i که مقدارش به متغیری از نوع int با اسم j اختصاص داده می‌شود. این دو متغیر در stack ذخیره می‌شوند. هنگامی‌که مقدار i را به j اختصاص می‌دهیم، یک کپی (کاملاً جدا و مجزا) از مقدار i به j داده می‌شود و به عبارت دیگر هنگامی که یکی از آن‌ها را تغییر دهید، دیگری تغییر نمی‌یابد:



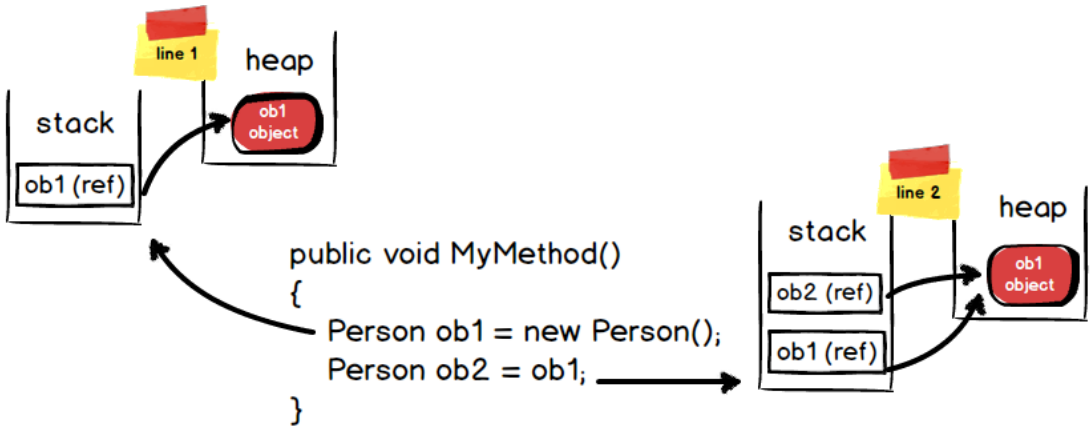
هنگامی‌که یک شیء می‌سازید و reference آن را با یک reference دیگر مساوی قرار می‌دهید، آن‌گاه هر دوی این reference ها به یک شیء رجوع می‌کنند و تغییر هر کدام از آن‌ها باعث تغییر شیء می‌شود زیرا هردو reference به یک شیء اشاره می‌کنند.

به مثال زیر توجه کنید:

```
using System;
class Person
{
    public string Name;
    public string Family;

    public void Show()
```

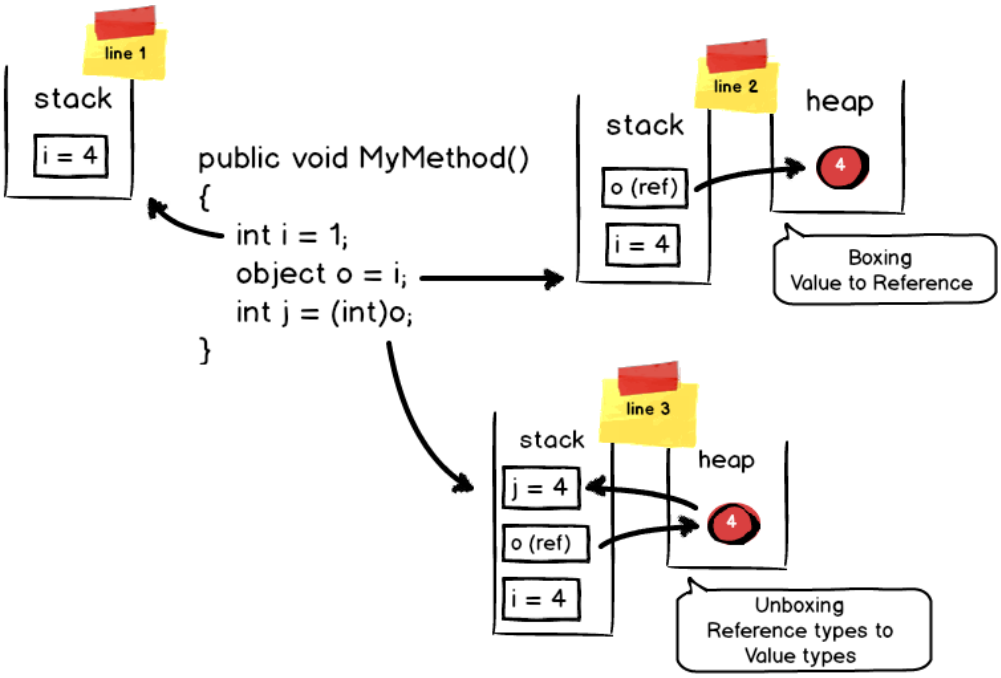
```
    {
        Console.WriteLine(Name + " " + Family);
    }
}
class Myclass
{
    static void Main()
    {
        Person ob1 = new Person();
        Person ob2 = ob1;

        ob1.Name = "Nicolas";
        ob1.Family = "Cage";

        Console.Write("ob1: ");
        ob1.Show();
        Console.Write("ob2: ");
        ob2.Show();

        Console.WriteLine();

        ob2.Name = "Ian";
        ob2.Family = "Somerhalder";

        Console.Write("ob1: ");
        ob1.Show();
        Console.Write("ob2: ");
        ob2.Show();
    }
}
```

همان‌طور که می‌بینید، ابتدا یک شیء ساخته و سپس reference دیگری تعریف کرده‌ایم و نهایتاً آن‌ها را مساوی هم قرار داده‌ایم. توجه کنید که برای ob2 شیء جدید تعریف نکرده‌ایم بلکه ob2 به همان شیءای رجوع می‌کند که ob1 به آن رجوع می‌کند. بنابراین تغییر هرکدام بر روی شیء تاثیر می‌گذارد. همان‌طور که می‌بینید، ob1.Name و ob2.Family در ابتدا برابر با Nicolas Cage است سپس با تغییر ob2.Name و ob2.Family به Ian Somerhalder مقادیر فیلدهای ob1 نیز تغییر خواهند کرد. به شکل زیر توجه کنید:



Boxing and Unboxing

به‌طور خلاصه، وقتی که یک مقدار value type را تبدیل به reference type می کنید، در واقع اطلاعات را از stack به heap می‌برید و هنگامی که یک مقدار reference type را تبدیل به value type می کنید، اطلاعات را از heap به stack می‌برید. این رفت و برگشت اطلاعات از stack به heap روی performance (کارایی، سرعت اجرا) برنامه تاثیر می گذارد. فرستادن اطلاعات از stack به heap در اصطلاح boxing و فرستادن اطلاعات از heap به stack در اصطلاح unboxing نامیده می‌شود.



استفاده از boxing و unboxing باعث افت performance می‌شود بنابراین تا آنجا که می‌توانید از انجام این کار پرهیز کنید و فقط در مواردی که واقعاً نیازمند این کار هستید و راه دیگری نیست، از آن استفاده کنید.

Garbage Collection

Garbage Collection نوعی مدیریت حافظه‌ی خودکار محسوب می‌شود. هربار که یک شیء می‌سازید، object شما در heap ذخیره می‌شود. تا زمانی که فضای کافی برای ذخیره‌ی این اشیاء داشته باشید می‌توانید شیء جدید بسازید اما همان‌طور که می‌دانید حافظه نامحدود نیست و ممکن است پر شود. بنابراین باید object های بی‌استفاده، از حافظه پاک شوند تا بتوان مجدداً اشیای دیگری را در حافظه ذخیره کرد. در بسیاری از زبان‌های برنامه‌نویسی برای آزاد کردن حافظه

از چیزهایی که در آن ذخیره شده، به‌صورت دستی و کدنویسی باید این‌کار انجام شود. مثلاً در C++ برای این منظور از delete operator استفاده می‌شود اما سی‌شارپ برای این منظور از راه حلی بهتر و ساده‌تر به اسم Garbage Collection استفاده می‌کند. Garbage Collection بدون اینکه برنامه‌نویس نیاز باشد کار خاصی انجام دهد به‌صورت خودکار، اشیایی که در heap قرار دارند و به هیچ reference ای وصل نیستند را پاک می‌کنند. اینکه دقیقاً چه زمانی این‌کار انجام می‌شود، مشخص نیست اما اگر می‌خواهید قبل از پاک شدن یک شیء توسط garbage collector کار خاصی را انجام دهید یا فقط از پاک شدن آن مطلع شوید از destructors استفاده می‌کنید. از destructor در سطوح حرفه‌ای برنامه‌نویسی استفاده می‌شود و دانستن آن چندان برای شما که اول راه هستید ضروری نیست اما اگر در این مورد کنجکاوید می‌توانید شخصاً در مورد آن تحقیق کنید.

Object Initializers

Object Initializers روشی دیگر برای ساخت شیء و مقدار دهی به field ها و property های (در مورد property بعداً بحث خواهیم کرد) کلاس است. با استفاده از object initializers، دیگر constructor کلاس را به روش معمول صدا نمی‌زنید بلکه اسم field ها و property ها را می‌نویسید و مستقیماً به آن‌ها مقدار می‌دهید. استفاده‌ی اصلی object initializers برای anonymous type های ساخته شده توسط LINQ است (در مورد LINQ و anonymous types بعداً صحبت خواهیم کرد) اما در حالت معمول نیز می‌توانند مورد استفاده قرار گیرند.

به مثال زیر توجه کنید:

```
using System;
class Human
{
    public string Name;
    public int Age;

    public void Show()
    {
        Console.WriteLine(Name + " " + Age);
    }
}
class ObjInitializersDemo
{
    static void Main()
    {
        Human Man = new Human { Name = "Paul", Age = 28 };
        Man.Show();
    }
}
```

همان‌طور که می‌بینید، Man.Name برابر با Paul و Man.Age را برابر با ۲۸ قرار داده‌ایم. نکته این‌جاست که از هیچ constructor ای استفاده نکرده‌ایم بلکه شیء Man توسط خط کد زیر تولید شده است:

```
Human Man = new Human { Name = "Paul", Age = 28 };
```

Optional Arguments

C# 4.0 ویژگی جدیدی به‌نام Optional Arguments دارد که باعث می‌شود برای فرستادن argument ها و دریافت پارامترها، روش دیگری نیز در دست‌تان باشد. همان‌طور که اسم این ویژگی جدید (argument های دلخواه) بیان‌کننده‌ی ماهیت آن است، با استفاده از optional arguments می‌توانید متدهایی تعریف کنید که از بین چندین پارامترش، بعضی از آن‌ها قابلیت این را داشته باشند که برای دریافت argument، اجباری نداشته باشند و اگر صلاح دانستید به آن‌ها argument دهید. استفاده از این ویژگی بسیار راحت است، کافی است هنگام تعریف پارامترها به آن‌ها یک مقدار پیش‌فرض بدهید.

به نمونه‌ی زیر توجه کنید:

```
public void OptArg(int a, int b = 2, int c = 3)
{
    Console.WriteLine("This is a, b, c: {0} {1} {2}", a, b, c);
}
```

در متد بالا، پارامتر b و c اختیاری هستند و به این طریق شما ویژگی optional argument را فعال کردید. توجه کنید که پارامتر a همان حالت معمول را دارد و اختیاری نیست و حتماً باید مقدار دهی شود.

به مثال زیر توجه کنید:

```
using System;
class OptionalArgs
{
    public void OptArg(int a, int b = 2, int c = 3)
    {
        Console.WriteLine("This is a, b, c: {0} {1} {2}", a, b, c);
    }
}
class OptionalArgsDemo
{
    static void Main()
    {
        OptionalArgs ob = new OptionalArgs();

        ob.OptArg(5);
        ob.OptArg(3, 9);
        ob.OptArg(4, 6, 8);
    }
```

```
}
```

در این مثال، متد OptArg() به سه طریق صدا زده شده است. ابتدا یک، سپس دو و در نهایت سه argument دریافت کرده است. این امکان وجود ندارد که این متد را بدون هیچ argument ای اجرا کنید چراکه پارامتر a اختیاری نیست و مقداردهی به آن اجباری است. آیا استفاده از این روش شبیه به method overloading نیست؟ بله، شما با این کار به یک متد به سه طریق مقدار داده‌اید که به method overloading شباهت دارد اما این روش‌ها جایگزینی برای هم نیستند بلکه در بعضی موارد برای راحتی برنامه‌نویس استفاده می‌شود و در برخی موارد برای خط کد کمتر ممکن است از این روش هم بتوانید بهره‌مند شوید. توجه کنید که اگر به پارامترهای دلخواه هیچ مقداری ندهید، مقدار پیش‌فرض آن‌ها در نظر گرفته می‌شود. همچنین پارامترهای که اجباری هستند باید پیش از پارامترهای اختیاری قرار بگیرند. برای نمونه، خط کد زیر نادرست است:

```
public void OptArg(int b = 2, int c = 3, int a) // Error!

// Or
public void OptArg(int b = 2, int a, int c = 3) // Error!
```

به‌دلیل اینکه پارامتر a اجباری است باید پیش از پارمترهای اختیاری قرار بگیرد. از optional arguments نیز می‌توانید در constructor، indexer و delegate نیز استفاده کنید (indexer و delegate در مقالات آینده مورد بحث قرار می‌گیرند).

Named Arguments

یکی دیگر از ویژگی‌های جدیدی که به C# 4.0 افزوده شده، named argument است. همان‌طور که می‌دانید، هنگامی‌که argument هایی را به متد می‌فرستید، ترتیب این argument ها باید مطابق با ترتیب پارامترهایی باشد که در متد تعریف شده‌اند. با استفاده از named arguments می‌توانید این محدودیت و اجبار را بردارید. استفاده از این ویژگی نیز بسیار ساده است، کافی‌ست نام پارامتری که argument قرار است به آن داده شود را در هنگام ارسال argument مشخص کنید و بعد از این‌کار، دیگر ترتیب argument ها اهمیتی ندارد.

به مثال زیر توجه کنید:

```
using System;
class NamedArgsDemo
{
    static int Div(int firstParam, int secondParam)
    {
        return firstParam / secondParam;
    }
    static void Main()
    {
```

```
        int result;

        // Call by use of normal way (positional arguments).
        result = Div(10, 5);
        Console.WriteLine(result);

        // Call by use of named arguments.
        result = Div(firstParam: 10, secondParam: 5);
        Console.WriteLine(result);

        // Order dosn't matter with a named argument.
        result = Div(secondParam:5, firstParam: 10);
        Console.WriteLine(result);
    }
}
```

همان‌طور که می‌بینید متد Div() در هر سه باری که فراخوانی شده، نتیجه‌ی یکسانی را تولید کرده است. ابتدا از این متد به‌صورت معمول استفاده کردیم و سپس در فراخوانی بعدی، نام پارامترها را نیز مشخص کرده‌ایم (در اینجا از ویژگی named arguments استفاده شد) و در نهایت همان‌طور که می‌بینید، ترتیب را به‌هم زدیم و جای argument ها را عوض کردیم اما نتیجه تغییر نکرده است.

همچنین می‌توانید named arguments را با حالت معمول (positional arguments) ادغام کنید به شرطی‌که همه‌ی positional arguments را پیش از named arguments قرار دهید:

```
Div(10, secondParam: 5);
```

از named arguments و optional arguments همچنین می‌توانید در constructor، indexer و delegate نیز استفاده کنید. (indexer و delegate جزء مباحث آینده هستند.)