More About C# Types

Topics Covered

- The Size of Objects
- The Call Stack
- Stack Frames
- The Stack
- The Heap
- The new Operator
- Object Variables
- The Lifetime of Objects
- Garbage Collection
- Value Types and Reference Types
- Structs and Classes
- The Garbage Collector

Other Built-In Types

If we revisit a slide from the previous deck, we see an anomaly.

All of the built-in types have a known size except for two types:

- string
- object

What is different about these types that makes their exact size variable?

C# Alias	.NET Class Name	Type	Size (bits)	Range (values)
char	Char	A single Unicode character	16	Unicode symbols used in text
bool	Boolean	Logical Boolean type	8	true or false
object	Object	Base type of all other types		
string	String	A sequence of characters		
DateTime	DateTime	Represents date and time	64	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999

Strings

To answer this question, lets think about strings. These are all strings:

```
string empty = "";
string a = "a";
string alphabet = "abcdefghijklmnopqrstuvwxyz";
Console.WriteLine(empty.Length);    // 0
Console.WriteLine(a.Length);        // 1
Console.WriteLine(alphabet.Length); // 26
```

Here we declare and initialize three string variables, each different.

String objects have a property, Length, which tells us how many characters are in the string. These strings have lengths of 0, 1, and 26.

The maximum length of a string is 2 gigabytes. That's a lot of memory!

Objects

Here are some objects, each quite different:

```
object o = new object();
object dt = new DateTime(2018, 9, 11);
object q27 = new string('q', 27);
object rand = new Random();
object memStream = new MemoryStream();
object client = new HttpClient();
object md5 = MD5.Create();
```

Objects are created from class definitions using the **new** operator. **new** allocates memory.

Objects manage information. That information can change as you call its methods and set its properties. It maintains state.

The size of an object depends on its data, and that can change as you use the object.

Memory: The Stack and The Heap

Programs use memory in two different ways.

When we declare local variables of primitive types that are of fixed size, that memory is allocated from the stack.

To create strings and objects, memory is allocated from the heap.

First, let's look at the stack. To understand the stack, we need to look at what happens when you call a method.

The Call Stack

Given the code shown to the right, imagine that our Main method calls MethodZero. MethodZero calls MethodOne, which calls MethodTwo, which calls MethodThree.

If we set a break point in MethodThree then start the debugger, we can inspect the call stack when the debugger breaks:

Call Stack

Name	Langu
DataTypes.exe!DataTypes.Program.MethodThree() Line 92	C#
DataTypes.exe!DataTypes.Program.MethodTwo() Line 86	C#
DataTypes.exe!DataTypes.Program.MethodOne() Line 80	C#
DataTypes.exe!DataTypes.Program.MethodZero() Line 98	C#
DataTypes.exe!DataTypes.Program.Main(string[] args) Line 17	C#

Call Stack Breakpoints Exception Settings Command Window Immediate Window Output

```
static void MethodOne()
{
    // declare local variables and do stuff
    MethodTwo();
}

static void MethodTwo()
{
    // declare local variables and do stuff
    MethodThree();
}

static void MethodThree()
{
    // declare local variables and do stuff
    Console.WriteLine("Leaving MethodThree");
}

static void MethodZero()
{
    // declare local variables and do stuff
    MethodOne();
}
```

Stack Frames

Each time we call a method, we require memory in which to store local variables.

This memory is isolated from the memory used by other methods in the call stack.

When code in a method is executing, it cannot “see” variables local to the methods below it in the call stack unless those variables are passed to it as parameters.

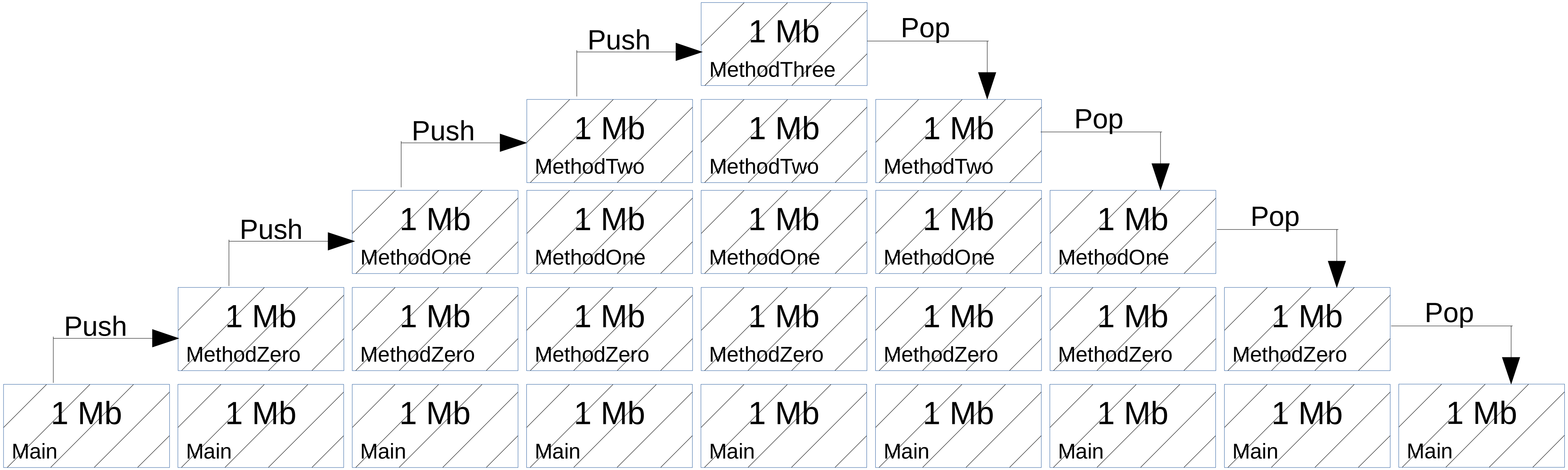
These memory structures are called *stack frames*.

Each time a method is called, a new stack frame is created and *pushed* onto a structure called the stack.

When a method returns, its stack frame is *popped* from the stack.

The Stack

The default size of a stack frame in a .NET executable is 1 megabyte (1 Mb).



The Heap

The heap is a pool of memory managed by the running process.

A process can have many stack frames. The exact number changes as the depth of the call stack changes.

A process has only one heap.

The heap can grow and shrink as an application creates new objects, uses them, and lets go of them.

The heap is managed by the Garbage Collector.

One good description of the stack and heap can be found here.

Local Variables for objects

When we create an object using the **new** operator, the operator returns a value that we assign to a local variable:

```
Random rand = new Random();
double d = rand.NextDouble();
```

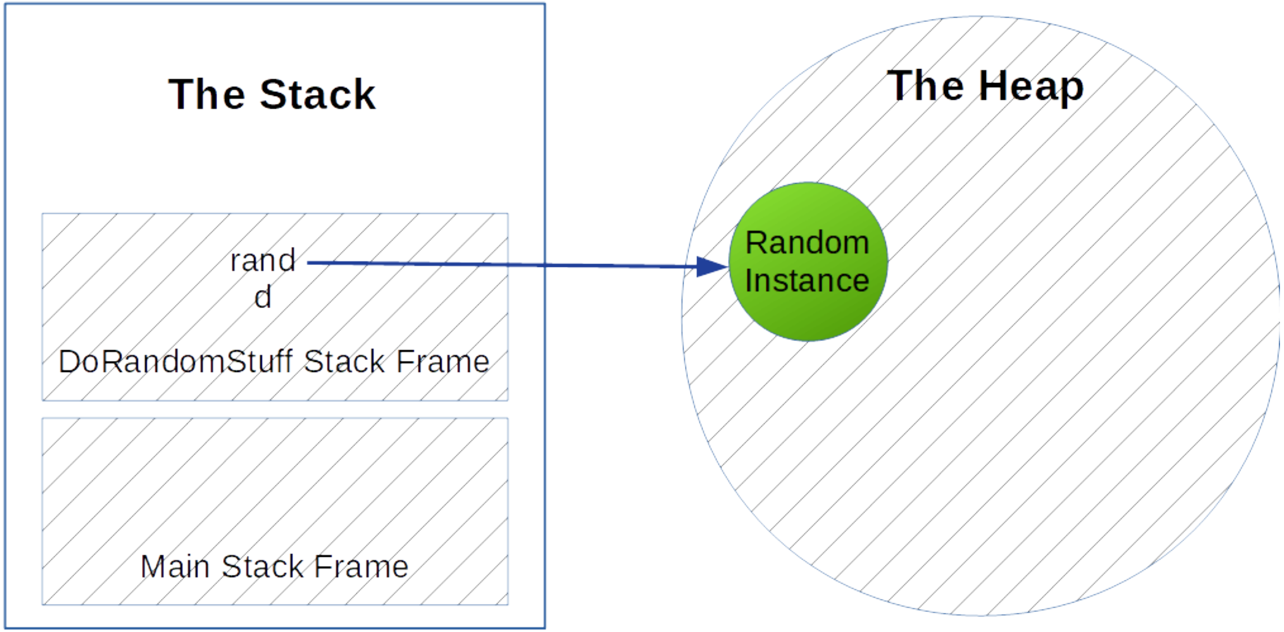
Under the hood, the local variable *rand* is not the actual Random object – it is pointer to a location in the heap where the object was created.

One of the objectives of C# was to relieve programmers of the complications of using pointers. Thus, the language allows us to use *rand* as if it were the actual object to which the variable points.

Note that rand itself, the local variable which is just a pointer, lives on the stack.

Lifetime of Objects

```
static void DoRandomStuff()
{
    Random rand = new Random();
    double d = rand.NextDouble();
    // do more stuff with rand.
}
```

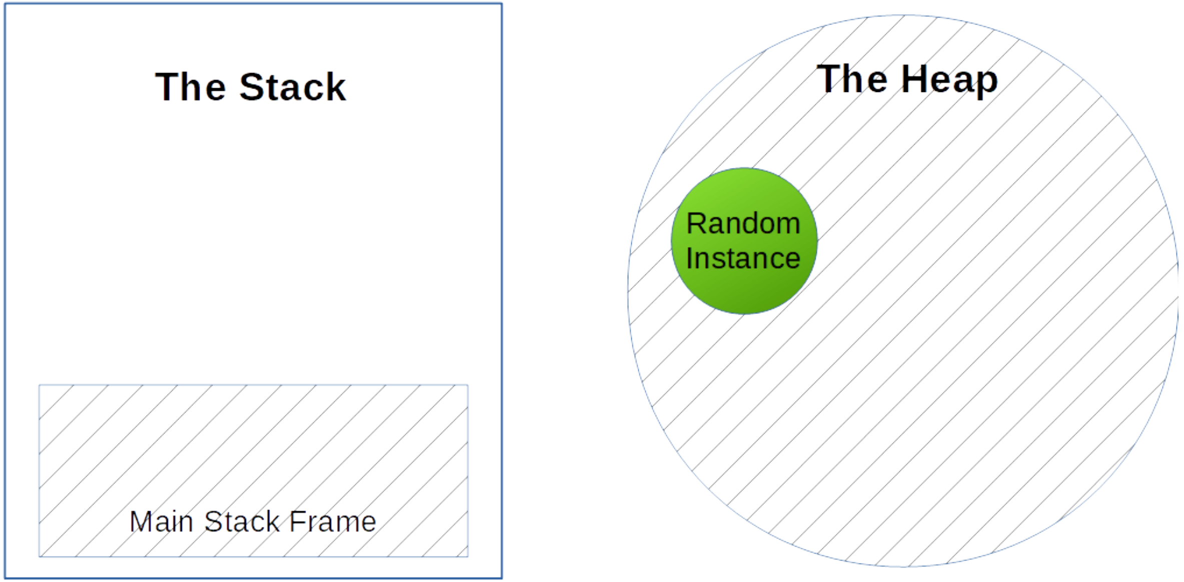


Garbage Collection

After DoRandomStuff returns, its stack frame is popped from the stack. The pointer to the Random instance no longer exists, though the object still exists on the heap.

The Garbage Collector monitors objects in the heap and detects those which have no remaining variables pointing them.

Unreferenced objects in the heap are removed so that the memory can be reclaimed.



Value Types and Reference Types

These primitive types that are defined entirely within the stack memory are called *value types*.

The types that are allocated on the heap are called *references types*, because our local variables reference, or point to them.

All value types are structs. All reference types are classes.

We can observe this by examining their definitions: double, HttpClient, DateTime, MemoryStream, etc.

Structs and classes are very similar – they can define methods, properties, fields and events.

Structs do no support inheritance.

Arguments “by value” and “by reference”

When value type variables are passed as arguments into a function, they are passed “by value” - a copy of the value is allocated on the stack of the called function. Changes made to the copy will not persist after the method call returns.

When reference type variables are passed as arguments into a function, they are passed as a pointer to a location in heap memory. Changes made to this object will persist in the original object after the method call returns.

```
    int year = 1987;
    Building townHall = new Building(year, "wood", "A-frame");
    Update(year - 12, townHall);
    Console.WriteLine(year);  // 1987
    Console.WriteLine(townHall.YearBuilt); // 1975
}

static void Update(int yearBuilt, Building b)
{
    b.YearBuilt = yearBuilt;  // change retained
    yearBuilt = 3173; // assignment has no consequence
}
```

The Garbage Collector

We rarely need to concern ourselves with the Garbage Collector – it runs in the background - but there are scenarios where it may make sense to use it.

If your program has just completed an operation that you know has generated a large number of objects which are no longer needed, you can force a cycle of garbage collection by calling GC.Collect().

The GC class also exposes numerous methods that allow us to inspect the state of the heap.

Topics Covered

- The Size of Objects
- The Call Stack
- Stack Frames
- The Stack
- The Heap
- The new Operator
- Object Variables
- The Lifetime of Objects
- Garbage Collection
- Value Types and Reference Types
- Structs and Classes
- The Garbage Collector