C# Heap(ing) Vs Stack(ing) in .NET

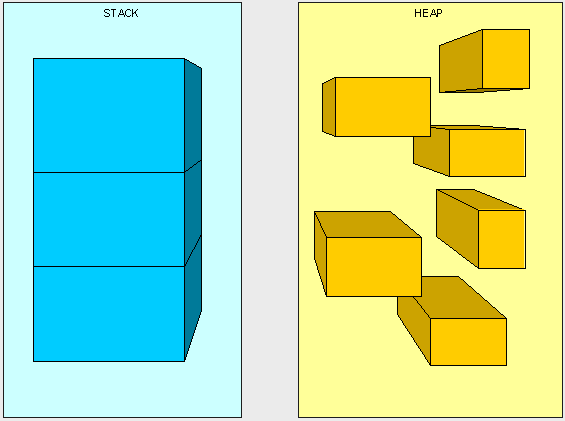
Even though with the .NET framework we don't have to actively worry about memory management and garbage collection (GC), we still have to keep memory management and GC in mind in order to optimize the performance of our applications. Also, having a basic understanding of how memory management works will help explain the behavior of the variables we work with in every program we write.  In this article I'll cover the basics of the Stack and Heap, types of variables and why some variables work as they do.

There are two places the .NET framework stores items in memory as your code executes.  If you are not yet familiar with them, let me introduce you to the Stack and the Heap. Both the Stack and Heap help us run our code. They reside in the operating memory on our machine and contain the pieces of information we need to make it all happen.

**Stack vs. Heap: What's the difference?**

The Stack is more or less responsible for keeping track of what's executing in our code (or what's been "called").  The Heap is more or less responsible for keeping track of our objects (our data, well... most of it; we'll get to that later).

Think of the Stack as a series of boxes stacked one on top of the next.  We keep track of what's going on in our application by stacking another box on top every time we call a method (called a Frame).  We can only use what's in the top box on the Stack.  When we're done with the top box (the method is done executing) we throw it away and proceed to use the stuff in the previous box on the top of the Stack. The Heap is similar except that its purpose is to hold information (not keep track of execution most of the time) so anything in our Heap can be accessed at  any time.  With the Heap, there are no constraints as to what can be accessed like in the Stack.  The Heap is like the heap of clean laundry on our bed that we have not taken the time to put away yet; we can grab what we need quickly. The Stack is like the Stack of shoe boxes in the closet where we have to take off the top one to get to the one underneath it.



The picture above, while not really a true representation of what's happening in memory, helps us distinguish a Stack from a Heap.  
   
The Stack is self-maintaining, meaning that it basically takes care of its own memory management.  When the top box is no longer used, it's thrown out.  The Heap, on the other hand, must worry about Garbage collection (GC), which deals with how to keep the Heap clean (no one wants dirty laundry laying around, it stinks!).

**What goes on the Stack and Heap?**

We have four main types of things we'll be putting in the Stack and Heap as our code is executing: Value Types, Reference Types, Pointers, and Instructions.

**Value Types:**

In C#, all the "things" declared with the following list of type declarations are Value types (because they are from System.ValueType):

* bool
* byte
* char
* decimal
* double
* enum
* float
* int
* long
* sbyte
* short
* struct
* uint
* ulong
* ushort

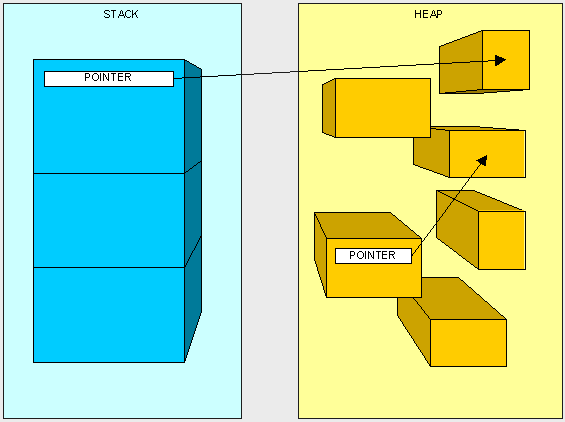
**Reference Types:**

All the "things" declared with the types in this list are Reference types (and inherit from System.Object, except, of course, for object which is the System.Object object):

* class
* interface
* delegate
* object
* string

**Pointers:**

The third type of "thing" to be put in our memory management scheme is a Reference to a Type. A Reference is often referred to as a Pointer. We don't explicitly use Pointers, they are managed by the Common Language Runtime (CLR). A Pointer (or Reference) is different than a Reference Type in that when we say something is a Reference Type, it means we access it through a Pointer. A Pointer is a chunk of space in memory that points to another space in memory.  A Pointer takes up space just like any other thing that we're putting in the Stack and Heap and its value is either a memory address or null.



**Instructions:**

You'll see how the  "Instructions" work later in this article...

**How is it decided what goes where? (Huh?)**

Ok, one last thing and we'll get to the fun stuff.

Here are our two golden rules:

1. A Reference Type always goes on the Heap; easy enough, right?
2. Value Types and Pointers always go where they were declared. This is a little more complex and needs a bit more understanding of how the Stack works to figure out where "things" are declared.

The Stack, as we mentioned earlier, is responsible for keeping track of where each thread is during the execution of our code (or what's been called).  You can think of it as a thread "state" and each thread has its own Stack.  When our code makes a call to execute a method the thread starts executing the instructions that have been JIT compiled and live on the method table, it also puts  the method's parameters on the thread Stack. Then, as we go through the code and run into variables within the method, they are placed on top of the Stack. This will be easiest to understand with an example.

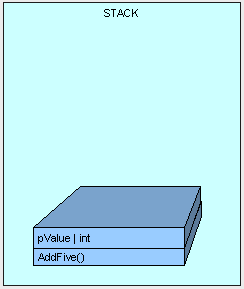
Take the following method:

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

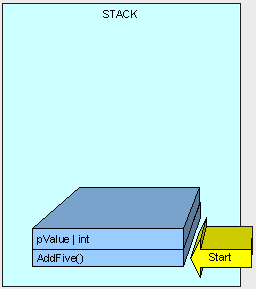
Here's what happens at the very top of the Stack.  Keep in mind that what we are looking at is placed on top of many other items already living in the Stack:

Once we start executing the method, the method's parameters are placed on the Stack (we'll talk more about passing parameters later).

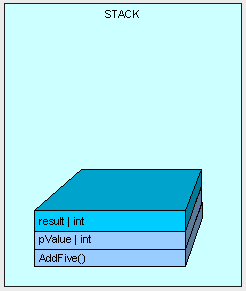
NOTE : the method does not live on the stack and is illustrated just for reference.



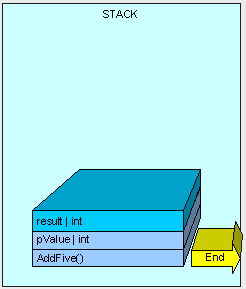
Next, control (the thread executing the method) is passed to the instructions to the AddFive() method which lives in our type's method table, a JIT compilation is performed if this is the first time we are hitting the method.



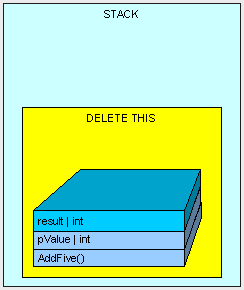
As the method executes, we need some memory for the "result" variable and it is allocated on the Stack.



The method finishes execution and our result is returned.



And all memory allocated on the Stack is cleaned up by moving a pointer to the available memory address where AddFive() started and we go down to the previous method on the stack (not seen here).



In this example, our "result" variable is placed on the stack.  As a matter of fact, every time a Value Type is declared within the body of a method, it will be placed on the stack.

Now, Value Types are also sometimes placed on the Heap.  Remember the rule, Value Types always go where they were declared?  Well, if a Value Type is declared outside of a method, but inside a Reference Type then it will be placed within the Reference Type on the Heap.

Here's another example.

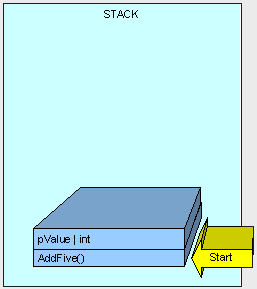
If we have the following MyInt class (which is a Reference Type because it is a class):

          public class MyInt  
          {            
             public int MyValue;  
          }

and the following method is executing:

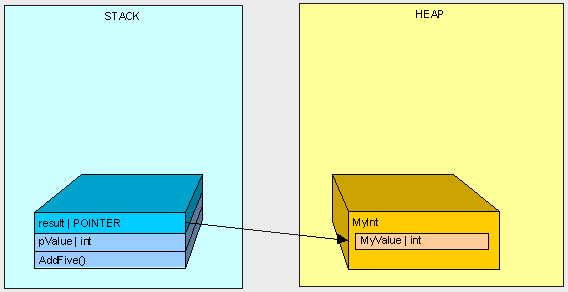
          public MyInt AddFive(int pValue)  
          {  
                MyInt result = new MyInt();  
                result.MyValue = pValue + 5;  
                return result;  
          }

Then just as before, the thread starts executing the method and its parameters are placed on sthe thread's stack.

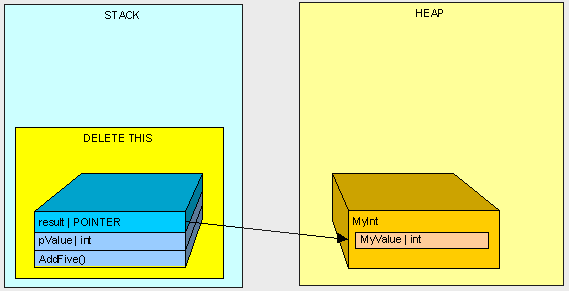


Now is when it gets interesting.

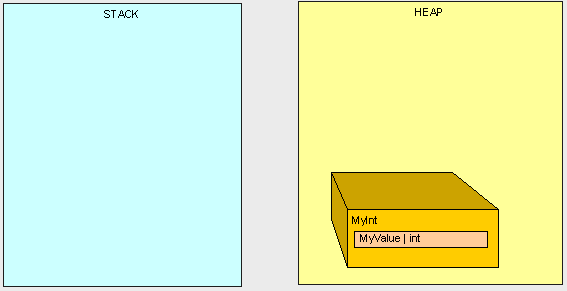
Because MyInt is a Reference Type, it is placed on the Heap and referenced by a Pointer on the Stack.



After AddFive() is finished executing (like in the first example), and we are cleaning up...



we're left with an orphaned MyInt in the Heap (there is no longer anyone in the Stack standing around pointing to MyInt)!



This is where the Garbage Collection (GC) comes into play.  Once our program reaches a certain memory threshold and we need more Heap space, our GC will kick off.  The GC will stop all running threads (a FULL STOP), find all objects in the Heap that are not being accessed by the main program and delete them.  The GC will then reorganize all the objects left in the Heap to make space and adjust all the Pointers to these objects in both the Stack and the Heap.  As you can imagine, this can be quite expensive in terms of performance, so now you can see why it can be important to pay attention to what's in the Stack and Heap when trying to write high-performance code.

Ok, that's great, but how does it really affect me?

Good question.

When we are using Reference Types, we're dealing with Pointers to the type, not the thing itself.  When we're using Value Types, we're using the thing itself.  Clear as mud, right?

Again, this is best described by example.

If we execute the following method:

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

We'll get the value 3.  Simple enough, right?

However, if we are using the MyInt class from before:

     public class MyInt  
          {  
                public int MyValue;  
          }

and we are executing the following method:

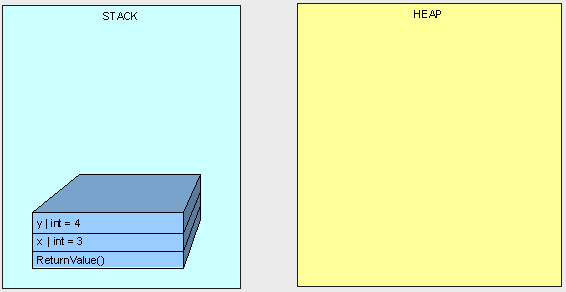
          public int ReturnValue2()  
          {  
                MyInt x = new MyInt();  
                x.MyValue = 3;  
                MyInt y = new MyInt();  
                y = x;                   
                y.MyValue = 4;                
                return x.MyValue;  
          }

What do we get?    4!

Why?...  How does x.MyValue get to be 4? Take a look at what we're doing and see if it makes sense:

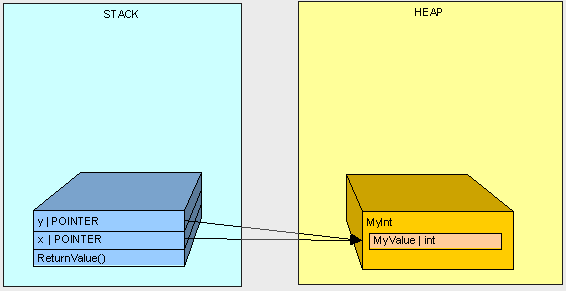
In the first example everything goes as planned:

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



In the next example, we don't get "3" because both variables "x" and "y" point to the same object in the Heap.

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



Hopefully this gives you a better understanding of a basic difference between Value Type and Reference Type variables in C# and a basic understanding of what a Pointer is and when it is used.  In the next part of this series, we'll get further into memory management and specifically talk about method parameters.

Even though with the .NET framework we don't have to actively worry about memory management and garbage collection (GC), we still have to keep memory management and GC in mind in order to optimize the performance of our applications. Also, having a basic understanding of how memory management works will help explain the behavior of the variables we work with in every program we write. In this article I'll cover some of the behaviors we need to be aware of when passing parameters to methods.  
  
In [Part I](http://www.c-sharpcorner.com/UploadFile/rmcochran/csharp_memory01122006130034PM/csharp_memory.aspx?ArticleID=9adb0e3c-b3f6-40b5-98b5-413b6d348b91&PagePath=/UploadFile/rmcochran/csharp_memory01122006130034PM/csharp_memory.aspx) we covered the basics of the Heap and Stack functionality and where Variable Types and Reference Types are allocated as our program executes. We also covered the basic idea of what a Pointer is.

**Parameters, the Big Picture.**

Here's the detailed view of what happens as our code executes. We covered the basics of what happens when we make a method call in Part I. Let's get into more detail...

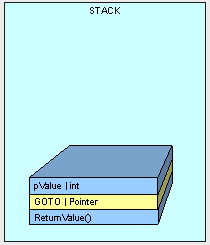
When we make a method call here's what happens:

1. Space is allocated for information needed for the execution of our method on the stack (called a Stack Frame). This includes the calling address (a pointer) which is basically a GOTO instruction so when the thread finishes running our method it knows where to go back to in order to continue execution.
2. Our method parameters are copied over. This is what we want to look at more closely.
3. Control is passed to the JIT'ted method and the thread starts executing code. Hence, we have another method represented by a stack frame on the "call stack".

The code:

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

Will make the stack look like this:



NOTE : the method does not live on the stack, and is illustrated here just for reference as the beginnnig of the stack frame.  
   
As discussed in Part I, Parameter placement on the stack will be handled differently depending on whether it is a value type or a reference type. A value types is copied over and the reference of a reference type is copied over.ed over.

**Passing Value Types.**

Here's the catch with value types...

First, when we are passing a value types, space is allocated and the value in our type is copied to the new space on the stack. Look at the following method:

     class Class1

     {

          public void Go()

          {

              int x = 5;

              AddFive(x);

              Console.WriteLine(x.ToString());

          }

          public int AddFive(int pValue)

          {

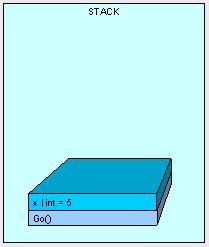
              pValue += 5;

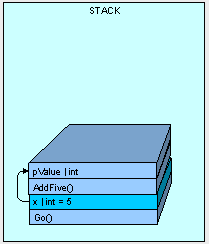
              return pValue;

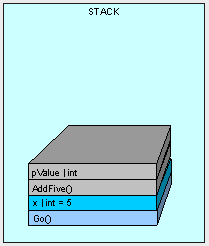
          }

     }

As the method executes, space for "x" is placed on the stack with a value of 5.

  
   
Next, AddFive() is placed on the stack with space for it's parameters and the value is copied, bit by bit from x.

  
   
When AddFive() has finished execution, the thread is passed back to Go() and because AddFive() has completed, pValue is essentially "removed":

  
   
So it makes sense that the output from our code is "5", right? The point is that any value type parameters passed into a method are carbon copies and we count on the original variable's value to be preserved.

One thing to keep in mind is that if we have a very large value type (such as a big struct) and pass it to the stack, it can get very expensive in terms of space and processor cycles to copy it over each time. The stack does not have infinite space and just like filling a glass of water from the tap, it can overflow. A struct is a value type that can get pretty big and we have to be aware of how we are handling it.

Here's a pretty big struct:

           public struct MyStruct

           {

               long a, b, c, d, e, f, g, h, i, j, k, l, m;

           }

Take a look at what happens when we execute Go() and get to the DoSomething() method below:

          public void Go()

          {

             MyStruct x = new MyStruct();

             DoSomething(x);

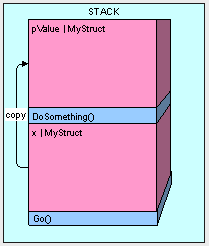
          }

           public void DoSomething(MyStruct pValue)

           {

                    // DO SOMETHING HERE....

           }



This can be really inefficient. Imaging if we passed the MyStruct a couple thousand times and you can understand how it could really bog things down.

So how do we get around this problem? By passing a reference to the original value type as follows:

          public void Go()

          {

             MyStruct x = new MyStruct();

             DoSomething(ref x);

          }

           public struct MyStruct

           {

               long a, b, c, d, e, f, g, h, i, j, k, l, m;

           }

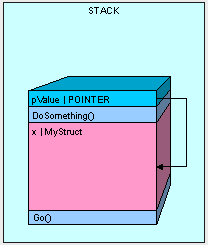
           public void DoSomething(ref MyStruct pValue)

           {

                    // DO SOMETHING HERE....

           }

This way we end up with more memory efficient allocation of our objects in memory.

  
   
The only thing we have to watch out for when passing our value type by reference is that we have access to the value type's value. Whatever is changed in pValue is changed in x. Using the code below, our results are going to be "12345" because the pValue.a actually is looking at the memory space where our original x variable was declared.

          public void Go()

          {

             MyStruct x = new MyStruct();

             x.a = 5;

             DoSomething(ref x);

             Console.WriteLine(x.a.ToString());

          }

          public void DoSomething(ref MyStruct pValue)

          {

                   pValue.a = 12345;

          }

**Passing Reference Types.**

Passing parameters that are reference types is similar to passing value types by reference as in the previous example.

If we are using the value type

           public class MyInt

           {

               public int MyValue;

           }

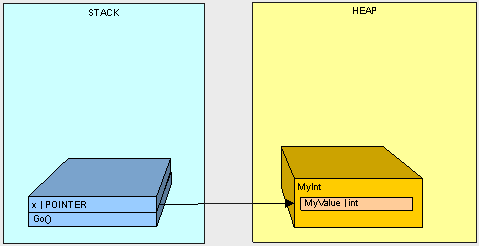
And call the Go() method, the MyInt ends up on the heap because it is a reference type:

          public void Go()

          {

             MyInt x = new MyInt();

          }



If we execute Go() as in the following code ...

          public void Go()

          {

             MyInt x = new MyInt();

             x.MyValue = 2;

             DoSomething(x);

             Console.WriteLine(x.MyValue.ToString());

          }

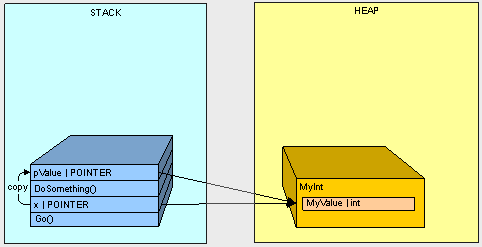
           public void DoSomething(MyInt pValue)

           {

               pValue.MyValue = 12345;

           }

Here's what happens...



1. Starting with the call to Go() the variable x goes on the stack.
2. Starting with the call to DoSomething() the parameter pValue goes on the stack.
3. The value of x (the address of MyInt on the stack) is copied to pValue

So it makes sense that when we change the MyValue property of the MyInt object in the heap using pValue and we later refer to the object on the heap using x, we get the value "12345".

So here's where it gets interesting. What happens when we pass a reference type by reference?

Check it out. If we have a Thing class and Animal and Vegetables are both things:

           public class Thing

           {

           }

           public class Animal:Thing

           {

               public int Weight;

           }

           public class Vegetable:Thing

           {

               public int Length;

           }

And we execute the Go() method below:

          public void Go()

          {

             Thing x = new Animal();

             Switcharoo(ref x);

              Console.WriteLine(

                "x is Animal    :   "

                + (x is Animal).ToString());

              Console.WriteLine(

                  "x is Vegetable :   "

                  + (x is Vegetable).ToString());

          }

           public void Switcharoo(ref Thing pValue)

           {

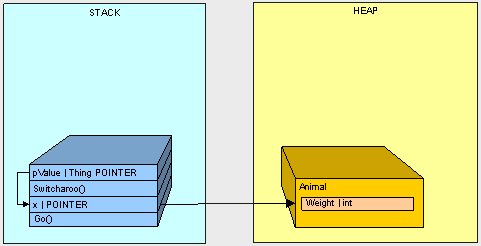
               pValue = new Vegetable();

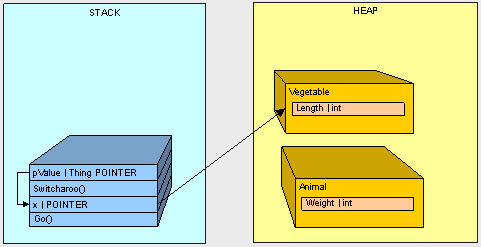
           }

Our variable x is turned into a Vegetable.

x is Animal    :   False  
x is Vegetable :   True

Let's take a look at what's happening:



1. Starting with the Go() method call, the x pointer goes on the stack
2. The Animal goes on the hea
3. Starting with the call to Switcharoo() method, the pValue goes on the stack and points to x  
     
   
4. The Vegetable goes on the heapthe heap
5. The value of x is changed through pValue to the address of the Vegetable

If we don't pass the Thing by ref, we'll keep the Animal and get the opposite results from our code.

If the above code doesn't make sense, check out my article on types of Reference variables to get a better understanding of how variables work with reference types.

**In Conclusion.**

We've looked at how parameter passing is handled in memory and now know what to look out for. In the next part of this series, we'll take a look at what happens to reference variables that live in the stack and how to overcome some of the issues we'll have when copying objects.

**A Copy Is Not A Copy.**

To clearly define the problem, let's examine what happens when there is a value type on the heap versus having a reference type on the heap. First we'll look at the value type. Take the following class and struct. We have a Dude class which contains a Name element and two Shoe(s). We have a CopyDude() method to make it easier to make new Dudes.

           public struct Shoe{

               public string Color;

           }

           public class Dude

           {

                public string Name;

                public Shoe RightShoe;

                public Shoe LeftShoe;

                public Dude CopyDude()

                {

                    Dude newPerson = new Dude();

                     newPerson.Name = Name;

                     newPerson.LeftShoe = LeftShoe;

                     newPerson.RightShoe = RightShoe;

                     return newPerson;

                }

                public override string ToString()

                {

                     return (Name + " : Dude!, I have a " + RightShoe.Color  +

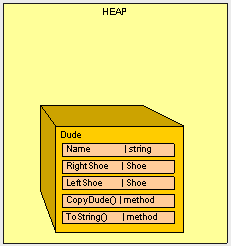
                         " shoe on my right foot, and a " +

                          LeftShoe.Color + " on my left foot.");

                }

           }

Our Dude class is a variable type and because the Shoe struct is a member element of the class they both end up on the heap.



When we run the following method:

           public static void Main()

           {

               Class1 pgm = new Class1();

                  Dude Bill = new Dude();

                  Bill.Name = "Bill";

                  Bill.LeftShoe = new Shoe();

                  Bill.RightShoe = new Shoe();

                  Bill.LeftShoe.Color = Bill.RightShoe.Color = "Blue";

                  Dude Ted =  Bill.CopyDude();

                  Ted.Name = "Ted";

                  Ted.LeftShoe.Color = Ted.RightShoe.Color = "Red";

                  Console.WriteLine(Bill.ToString());

                  Console.WriteLine(Ted.ToString());

           }

We get the expected output:

Bill : Dude!, I have a Blue shoe on my right foot, and a Blue on my left foot.  
Ted : Dude!, I have a Red shoe on my right foot, and a Red on my left foot.

What happens if we make the Shoe a reference type?  Herein lies the problem. If we change the Shoe to a reference type as follows:

           public class Shoe{

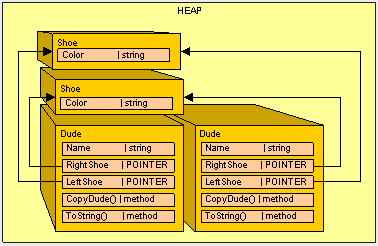
               public string Color;

           }

and run the exact same code in Main(), look how our input changes:

Bill : Dude!, I have a Red shoe on my right foot, and a Red on my left foot  
Ted : Dude!, I have a Red shoe on my right foot, and a Red on my left foot

The Red shoe is on the other foot. This is clearly an error. Do you see why it's happening? Here's what we end up with in the heap.



Because we now are using Shoe as a reference type instead of a value type and when the contents of a reference type are copied only the pointer is copied (not the actual object being pointed to), we have to do some extra work to make our Shoe reference type behave more like a value type.

Luckily, we have an interface that will help us out: ICloneable. This interface is basically a contract that all Dudes will agree to and defines how a reference type is duplicated in order to avoid our "shoe sharing" error. All of our classes that need to be "cloned" should use the ICloneable interface, including the Shoe class.

ICloneable consists of one method: Clone()

                  public object Clone()

                  {

                  }

Here's how we'll implement it in the Shoe class:

           public class Shoe : ICloneable

             {

                  public string Color;

                  #region ICloneable Members

                  public object Clone()

                  {

                      Shoe newShoe = new Shoe();

                      newShoe.Color = Color.Clone() as string;

                      return newShoe;

                  }

                  #endregion

             }

Inside the Cone() method, we just make a new Shoe, clone all the reference types and copy all the value types and return the new object. You probably noticed that the string class already implements ICloneable so we can call Color.Clone(). Because Clone() returns a reference to an object, we have to "retype" the reference before we can set the Color of the shoe.

Next, in our CopyDude() method we need to clone the shoes instead of copying them

                public Dude CopyDude()

                {

                    Dude newPerson = new Dude();

                     newPerson.Name = Name;

                     newPerson.LeftShoe = LeftShoe.Clone() as Shoe;

                     newPerson.RightShoe = RightShoe.Clone() as Shoe;

                     return newPerson;

                }

Now, when we run main:

           public static void Main()

           {

               Class1 pgm = new Class1();

                  Dude Bill = new Dude();

                  Bill.Name = "Bill";

                  Bill.LeftShoe = new Shoe();

                  Bill.RightShoe = new Shoe();

                  Bill.LeftShoe.Color = Bill.RightShoe.Color = "Blue";

                  Dude Ted =  Bill.CopyDude();

                  Ted.Name = "Ted";

                  Ted.LeftShoe.Color = Ted.RightShoe.Color = "Red";

                  Console.WriteLine(Bill.ToString());

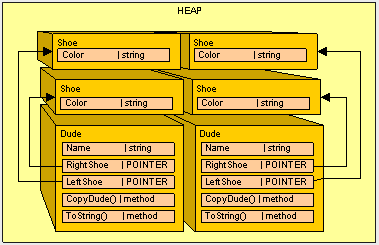
                  Console.WriteLine(Ted.ToString());

           }

We get:

Bill : Dude!, I have a Blue shoe on my right foot, and a Blue on my left foot  
Ted : Dude!, I have a Red shoe on my right foot, and a Red on my left foot

Which is what we want.



**Wrapping Things Up.**

So as a general practice, we want to always clone reference types and copy value types. (It will reduce the amount of aspirin you will have to purchase to manage the headaches you get debugging these kinds of errors.)

So in the spirit of headache reduction, let's take it one step further and clean up the Dude class to implement ICloneable instead of using the CopyDude() method.

           public class Dude: ICloneable

           {

                public string Name;

                public Shoe RightShoe;

                public Shoe LeftShoe;

                public override string ToString()

                {

                     return (Name + " : Dude!, I have a " + RightShoe.Color  +

                         " shoe on my right foot, and a " +

                          LeftShoe.Color + " on my left foot.");

                    }

                  #region ICloneable Members

                  public object Clone()

                  {

                       Dude newPerson = new Dude();

                       newPerson.Name = Name.Clone() as string;

                       newPerson.LeftShoe = LeftShoe.Clone() as Shoe;

                       newPerson.RightShoe = RightShoe.Clone() as Shoe;

                       return newPerson;

                  }

                  #endregion

             }

And we'll change the Main() method to use Dude.Clone()

           public static void Main()

           {

               Class1 pgm = new Class1();

                  Dude Bill = new Dude();

                  Bill.Name = "Bill";

                  Bill.LeftShoe = new Shoe();

                  Bill.RightShoe = new Shoe();

                  Bill.LeftShoe.Color = Bill.RightShoe.Color = "Blue";

                  Dude Ted =  Bill.Clone() as Dude;

                  Ted.Name = "Ted";

                  Ted.LeftShoe.Color = Ted.RightShoe.Color = "Red";

                  Console.WriteLine(Bill.ToString());

                  Console.WriteLine(Ted.ToString());

           }

And our final output is:

Bill : Dude!, I have a Blue shoe on my right foot, and a Blue on my left foot.  
Ted : Dude!, I have a Red shoe on my right foot, and a Red on my left foot.

So all is well.

Something interesting to note is that the assignment operator (the "=" sign) for the System.String class actually clones the string so you don't have to worry about duplicate references. However you do have to watch our for memory bloating. If you look back at the diagrams, because the string is a reference type it really should be a pointer to another object in the heap, but for simplicity's sake, it's shown as a value type.

**In Conclusion.**

As a general practice, if we plan on ever copying of our objects, we should implement (and use) ICloneable.  This enables our reference types to somewhat mimic the behavior of a value type. As you can see, it is very important to keep track of what type of variable we are dealing with because of differences in how the memory is allocated for value types and reference types.

In the next article, we'll look at a way to reduce our code "footprint" in memory.

<http://www.c-sharpcorner.com/article/C-Sharp-heaping-vs-stacking-in-net-part-iii/>

**Graphing**

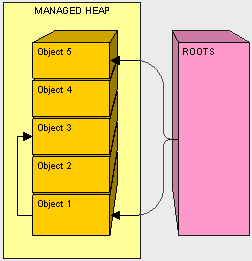
Let's look at this from the GC's point of view. If we are responsible for "taking out the trash" we need a plan to do this effectively. Obviously, we need to determine what is garbage and what is not (this might be a bit painful for the pack-rats out there).

In order to determine what needs to be kept, we'll first make the assumption that everything not being used is trash (those piles of old papers in the corner, the box of junk in the attic, everything in the closets, etc.)  Imagine we live with our two good friends: Joseph Ivan Thomas (JIT) and Cindy Lorraine Richmond (CLR). Joe and Cindy keep track of what they are using and give us a list of things they need to keep. We'll call the initial list our "root" list because we are using it as a starting point.  We'll be keeping a master list to graph where everything is in the house that we want to keep. Anything that is needed to make things on our list work will be added to the graph (if we're keeping the TV we don't throw out the remote control for the TV, so it will be added to the list. If we're keeping the computer the keyboard and monitor will be added to the "keep" list).

This is how the GC determines what to keep as well. It receives a list of "root" object references to keep from just-in-time (JIT) compiler and common language runtime (CLR) (Remember Joe and Claire?) and then recursively searches object references to build a graph of what should be kept.

Roots consist of:

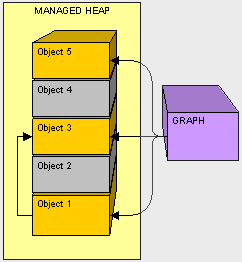
* Global/Static pointers. One way to make sure our objects are not garbage collected by keeping a reference to them in a static variable.
* Pointers on the stack. We don't want to throw away what our application's threads still need in order to execute.
* CPU register pointers. Anything in the managed heap that is pointed to by a memory address in the CPU should be preserved (don't throw it out).



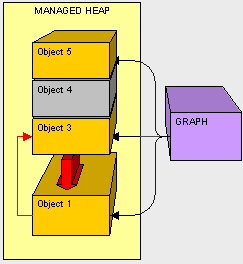
In the above diagram, objects 1, 3, and 5 in our managed heap are referenced from a root 1 and 5 are directly referenced and 3 is found during the recursive search.  If we go back to our analogy and object 1 is our television, object 3 could be our remote control. After all objects are graphed we are ready to move on to the next step, compacting.

**Compacting**

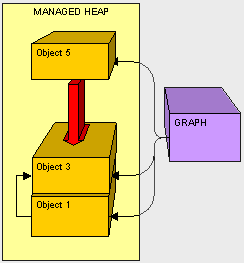
Now that we have graphed what objects we will keep, we can just move the "keeper objects" around to pack things up.



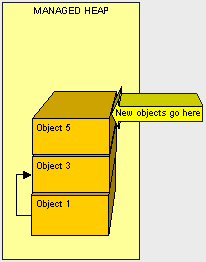
Fortunately, in our house we don't need to clean out the space before we put something else there. Since Object 2 is not needed, as the GC we'll move Object 3 down and fix the pointer in Object 1.



Next, as the GC, we'll copy Object 5 down



Now that everything is cleaned up we just need to write a sticky note and put it on the top of our compacted heap to let Claire know where to put new objects.



Knowing the nitty-gritty of CG helps in understanding that moving objects around can be very taxing. As you can see, it makes sense that if we can reduce the size of what we have to move, we'll improve the whole GC process because there will be less to copy.

**What about things outside the managed heap?**

As the person responsible for garbage collection, one problem we run into in cleaning house is how to handle objects in the car. When cleaning, we need to clean everything up. What if the laptop is in the house and the batteries are in the car?

There are situations where the GC needs to execute code to clean up non-managed resources such as files, database connections, network connections, etc. One possible way to handle this is through a finalizer.

class Sample

{

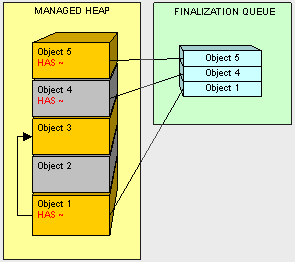
          ~Sample()

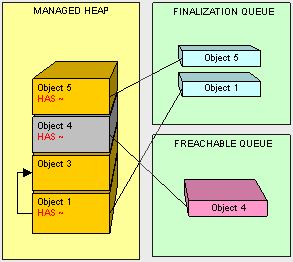
          {

                    // FINALIZER: CLEAN UP HERE

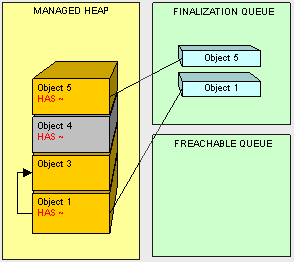
          }

}

During object creation, all objects with a finalizer are added to a finalization queue. Let's say objects 1, 4, and 5 have finalizers and are on the finalization queue.  Let's look at what happens when objects 2 and 4 are no longer referenced by the application and ready for garbage collection.  
   


Object 2 is treated in the usual fashion. However, when we get to object 4, the GC sees that it is on the finalization queue and instead of reclaiming the memory object 4 owns, object 4 is moved and it's finalizer is added to a special queue named freachable.   
   


There is a dedicated thread for executing freachable queue items. Once the finalizer is executed by this thread on Object 4, it is removed from the freachable queue. Then and only then is Objet 4 ready for collection.



So Object 4 lives on until the next round of GC.

Because adding a finalizer to our classes creates additional work for GC it can be very expensive and adversely affect the performance garbage collection and thus our program. Only use finalizers when you are absolutely sure you need them.

A better practice is to be sure to clean up non-managed resources. As you can imagine, it is preferable to explicitly close connections and use the IDisposable interface for cleaning up instead of a finalizer where possible.

**IDisposaible**

Classes that implement IDisposable perform clean-up in the Dispose() method (which is the only signature of the interface). So if we have a ResouceUser class instead of using a finalizer as follows:

public class ResourceUser

{

          ~ResourceUser() // THIS IS A FINALIZER

          {

                    // DO CLEANUP HERE

          }

}

We can use IDisposable as a better way to implement the same functionality:

public class ResourceUser : IDisposable

{

          #region IDisposable Members

          public void Dispose()

          {

                    // CLEAN UP HERE!!!

          }

          #endregion

}

IDisposable in integrated with the using keyword. At the end of the using block Dispose() is called on the object declared in using(). The object should not be referenced after the using block because it should be essentially considered "gone" and ready to be cleaned up by the GC.

public static void DoSomething()

{

ResourceUser rec = new ResourceUser();

using (rec)

{

                // DO SOMETHING

} // DISPOSE CALLED HERE

            // DON'T ACCESS rec HERE

}

I like putting the declaration for the object in the using block because it makes more sense visabally and rec is no longer available outside of the scope of the using block. Whis this pattern is more in line with the intention of the IDisposible interface, it is not required.

public static void DoSomething()

{

using (ResourceUser rec = new ResourceUser())

{

                // DO SOMETHING

} // DISPOSE CALLED HERE

}

By using using() with classes that implement IDisposible we can perform our cleanup without putting additional overhead on the GC by forcing it to finalize our objects.

**Static Variables: Watch Out!**

class Counter

{

          private static int s\_Number = 0;

          public static int GetNextNumber()

          {

                    int newNumber = s\_Number;

                    // DO SOME STUFF

                    s\_Number = newNumber + 1;

                    return newNumber;

          }

}

If two threads call GetNextNumber() at the same time and both are assigned the same value for newNumber before s\_Num}

If two threads call GetNextNumber() at the same time and both are assigned the same value for newNumber before s\_Number is incremented they will return the same result!word is one way to ensure only one thread can access a block of code at a time. As a best practice, you should lock as little code as possible because threads have to wait in a queue to execute the code in the lock()  block and it can be inefficient.

class Counter

{

          private static int s\_Number = 0;

          public static int GetNextNumber()

          {

                    lock (typeof(Counter))

                    {

                             int newNumber = s\_Number;

                             // DO SOME STUFF

                             newNumber += 1;

                             s\_Number = newNumber;

                             return newNumber;

                    }

          }

}

**Static Variables: Watch Out... Number 2!**

The next thing we have to watch out for objects referenced by static variables.  Remember, how anything that is referenced by a "root" is not cleaned up. Here's one of the ugliest examples I can come up with:

class Olympics

{

          public static Collection<Runner> TryoutRunners;

}

class Runner

{

          private string \_fileName;

          private FileStream \_fStream;

          public void GetStats()

          {

                    FileInfo fInfo = new FileInfo(\_fileName);

                    \_fStream = \_fileName.OpenRead();

          }

}

Because the Runner Collection is static for the Olympics class, not only will objects in the collection will not be released for garbage collection (they are all indirectly referenced through a root), but as you probably noticed, every time we  run GetStats() the stream is opened to the file. Because it is not closed and never released by GC this code is effectively a disaster waiting to happen. Imagine we have 100,000 runners trying out for the Olympics.  We would end up with that many non-collectable objects each with an open resource.  Ouch! Talk about poor performance!

**Singleton**

One trick to keep things light is to keep only one instance of a class in memory at all times. To do this we can use the GOF Singleton Pattern.

One trick to keep things light is to keep only one instance of a utility class in memory at all times. One easy way to  do this we can use the GOF Singleton Pattern. Singletons should be used with caution because they are really "global variables" and cause us much headached and "strange" behavior in multi-threaded applications where different threads could be altering the state of the object.  If we are using the singleton pattern (or any global variable) we should be able to justify it (in other words... don't do it without a good reason).

public">{

          private static Earth \_instance = new Earth();

          private Earth() { }

          public static Earth GetInstance() { return \_instance; }

}

We have a private constructor so only Earth can execute it's constructor and make an Earth. We have a static instance of Earth and a static method to get the instance. This particular implementation is thread safe because the CLR ensures thread safe creation of static variables. This is the most elegant way I have found to implement the singleton pattern in C#.

**In Conclusion...**

So to wrap up, some things we can do to improve GC performance are:

1. Clean up. Don't leave resources open!  Be sure to close all connections that are opened and clean up all non-managed objects as soon as possible. As a general rule when using non-managed objects, instantiate as late as possible and clean up as soon as possible.
2. Don't overdo references.  Be reasonable when using references objects.  Remember, if our object is alive, all of it's referenced objects will not be collected (and so on, and so on). When we are done with something referenced by class, we can remove it by either setting the reference to null.  One trick I like to do is setting unused references to a custom light weight NullObject to avoid getting null reference exceptions. The fewer references laying about when the GC kicks off, the less pressure the mapping process will be.
3. Easy does it with finalizers. Finalizers are expensive during GC we should ONLY use them if we can justify it. If we can use IDisposible instead of a finalizer, it will be more efficient because our object can be cleaned up in one GC pass instead of two.
4. Keep objects and their children together. It is easier on the GC to copy large chunks of memory together instead of having to essentially de-fragment the heap at each pass, so when we declare a object composed of many other objects, we should instantiate them as closely together as possible.

Next time we'll look even more closely at the GC process and look into ways to check under the hood as your program executes to discover problems that may need to be cleaned up.

<http://www.c-sharpcorner.com/article/C-Sharp-heaping-vs-stacking-in-net-part-iv/>