C# Review Guide

# C# Datatypes

* Primitive or simple Types – data types that can be created with a literal (All numeric types, bool, char, and strings are primitive types). All other types can parse a string to create an instance of that type but cannot take a literal.

int x = 5;

* Type suffixes – When using a literal, the compiler looks at the value of that literal and assumes the datatype based on that value.

int x = 5;

decimal x = 5.00;

In the above example, the compiler has determined quite obviously that that 5 is an integer and so that line compiles just fine. But the value *5.00* could either be a float, a double or a decimal. This is one of the rare instances where the compiler just picks one and assumes. It determines 5.00 is a double precision float and will therefore show a compile error since float is not automatically convertible to type decimal.

Type suffixes can be used to tell the compiler that a literal should be considered a certain type.

decimal x = 5.00M;

The list of suffixes is:

|  |  |
| --- | --- |
| Suffix | Data Type |
| F | float |
| D | double |
| M | Decimal |
| L | Long |
| U | Unsigned Int |
| UL | Unsigned Long |

* The list of intrinsic datatypes in C# is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| data type | clr type | size | smallest value | largest value |
| Signed Integrals | | | | |
| sbyte | System.SByte | 8 | -128 | 127 |
| short | System.Int16 | 16 | -32,768 | 32,768 |
| int | System.Int32 | 32 | -2,147,483,648 | 2,147,483,647 |
| long | System.Int64 | 64 | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| Unsigned Integrals | | | | |
| byte | System.Byte | 8 | 0 | 255 |
| ushort | System.UInt16 | 16 | 0 | 65,535 |
| uint | System.UInt32 | 32 | 0 | 4,294,967,295 |
| ulong | System.UInt64 | 64 | 0 | 18,446,744,073,709,551,615 |
| Floating Point | | | | |
| float | System.Single | 32 | -3.402823e38 | 3.402823e38 |
| double | System.Double | 64 | -1.79769313486232e308 | 1.79769313486232e308 |
| Other Types | | | | |
| char | System.Char | 16 | 1 unicode symbol |  |
| bool | System.Boolean | 8 | true | false |
| string | System.String | ? | a sequence of characters |  |

# Variable declaration

* C# is a type-safe language – Each variable must be declared having a specific type. Once it is declared as that type, it stays that type for the entire lifetime of that variable.

{datatype} {identifier} [ = {initial value}];

* C# is a case sensitive language – The following code creates two different variables. They are **not** the same.

int x = 5;

int X = 5;

* Implicitly-typed variables - These variables are declared using the ***var*** keyword and are required to have an assignment. The compiler looks at the assignment and discerns the type on the fly. These are still strongly-typed variables, it merely means the type of discovered and set by the compiler.

var X = 5;

Employee emp = new Employee();

* Variables, Class, Struct, Namespace, Method and Enum names should be Pascal-cased (each word-part is capitalized)

namespace ConsoleApp26

{

public enum MyEnum { }

class Program

{

static void Main(string[] args)

{

int EmployeeAge;

}

}

public struct MyStruct { }

}

* Method parameters should be camel-cased

private static void AddOrUpdateWord(string word, int id, HashSet<SearchTerm> terms)

# Variable Scope

Variable scope deal with the visibility of a variable within the context of your code. Variable scope has absolutely nothing to do with the lifetime of the variable, only the ability of our code to be able to see and access the data. There are basically three types of variable scope, but they all simply refer to the curly braces that they are defined within.

* Block scope – any variable declared within a statement block is scoped/visible only within the block that it was declared in.

if (DateFromDatabase.HasValue)

{

DateTime theDate = DateTime.Now;

}

// theDate not visible here.

* Method Scope – any variable declared within a method is scoped/visible only within the method that it was declared in (from curly-brace to curly-brace)

public void SomeMethod(int param1)

{

// This variable visible only within this method

DateTime theDate = DateTime.Now;

}

* Class Scope – any variable declared within a class is scoped/visible everywhere within the class that it was declared in. We are not talking public vs private here because that deals with access. This is merely stating that the visibility of this variable is from curly brace to curly brace.

public class MyClass

{

public int \_InternalTemp;

}

# Operators

The list of operators is:

|  |  |
| --- | --- |
| ++, -- | unary postfix or prefix operators |
| <, > | less than and greater than |
| <=, >= | less than or equal to and greater than or equal to |
| != | not equal to |
| == | is equal to (test) |
| is | returns true if left side is a type of the right side |
| as | casting operator |
| & | logical and |
| && | conditional and |
| | | logical or |
| || | conditional or |
| ^ | exclusive or |
| ?? | null coalescing (returns left side unless it is null) |
| ?: | conditional operator - like IIF() in TSQL or VB |
| = | assignment operator |
| +=,-=, \*=, /=, %=, ^=, >>=, <<= | compound assignment operators |
| >> << | bit shifting operators |
| +, -, \*, /, \, % | arithmetic operators |
| => | lambda operator |

# Converting between data types

* ToString method – Every single object automatically has a method called ToString. By default, this method merely returns the fully qualified name of the object (which is not usually helpful). However, most of the intrinsic C# data types have a useful implementation of this method which successfully converts it to a string.

int a = 456;

string a\_AsString = a.ToString();

* Casting
* Implicit Casting – In scenarios where you are converting from a datatype designed to hold a smaller numeric value into a datatype designed to hold much larger numbers, converting is implicitly done by the compiler. No casting needed.

int b = int.MaxValue;

long c = b; // no casting required

In the first line of code above, an integer is set to the absolute maximum value it will hold.

The second line of code creates a variable designed to hold a number twice that size and the value of b is copied into it. We know this will fit so the compiler handles that conversion for you.

* Explicit Casting – In scenarios where you are converting from a datatype designed to hold much larger number into a datatype designed to hold smaller numbers, the compiler requires casting.

// explicit casting - This would most assuredly fit in an int

long d = int.MaxValue;

// but the implicit cast doesn't work - explicit casting is required

int e = (int)d;

* Casting Operator – In scenarios that use a reference type, you may use the Casting Operator (as) to accomplish the same thing. However, casting still works just fine

object f = "Hello there";

string g = f as string; // casting operator

string h = (string)f; // normal c type casting

* Convert Class – This class contains many different methods for converting from one datatype to another

string age = "49";

int age\_AsInt = Convert.ToInt32(age);

* Parse Method – Many datatypes in C# contain a static method called Parse which takes a string as input and converts the string to the appropriate type.

string age2 = "49";

int AlsoAge\_AsInt = int.Parse(age2);

string today = "02/31/2017";

DateTime today\_As\_Date = DateTime.Parse(today);

* TryParse Method – Many datatypes in C# contain a static method called TryParse which gracefully tries to convert a string to the appropriate type. TryParse returns true or false indicating if it was successful or not….as opposed to returning the converted value. Instead, it relies on an out parameter to return the converted value.

string age3 = "Noneya";

int Age\_As\_Int\_Again;

if (int.TryParse(age3, out Age\_As\_Int\_Again))

{

// We'll enter here if the convert was successful

}

# Nullable Types

Many situations, especially when dealing with data from a database, require us to be able to assign a value of null to a variable. In C#, null is a reference and references cannot refer to value types by default. Therefore value types cannot be null! So to get around this in C#, we create a proxy type which wraps itself around our data and provides the functionality we need. Nullable types will have two properties: ***HasValue*** which indicates if the underlying value is null or not, and ***Value*** which contains the value if it is not null. If ***HasValue*** is false, don’t look at the ***Value*** property, as you will get an *InvalidCast* exception. If ***HasValue*** is true, you can safely access the ***Value*** property.

DateTime? DateFromDatabase = null;

if (DateFromDatabase.HasValue)

{

DateTime theDate = DateFromDatabase.Value;

// or

DateTime theDatetoo = (DateTime)DateFromDatabase;

}

# Arrays

In C#, there are 3 types of arrays. Arrays are zero based.

* Single dimensional arrays

// Single Dimensional Arrays

// Creation syntax

int[] Myintegers = new int[5]; // 5 items total

Myintegers[0] = 1; // initialize the first item to the value of 1

// Using an array initializer

int[] Myintegers2 = { 1, 2, 3, 4, 5 };

* Multidimensional arrays

// Multidimensional Arrays

int[,] TwoDimensions = new int[24, 36];

int[,,] ThreeDimensions = new int[24, 36, 24];

int[,,,] DoPeopleSeriouslyDoThis = new int[24, 36, 24, 36];

// Using an array initializer

int[,] KindaYucky = { { 1, 2, 3, 4, 5 }, { 1, 2, 3, 4, 5 }, { 1, 2, 3, 4, 5 } };

* Jagged arrays – simplest way to think of it is an array of arrays

// Jagged Arrays

int[][] jagged = new int[5][];

jagged[0] = new int[5];

* Other Array Information

// Find out how many elements in an array

var ArrayLength = Myintegers.Length;

// Find the number of dimensions

var NumElements = Myintegers.Rank;

// Sorting an array

Array.Sort(Myintegers);

// To Resize an array with a potential Gottcha!!!!

int[] AnotherVariable = Myintegers2;

Array.Resize(ref Myintegers2, 6);

Myintegers2[5] = 6;

// Resize creates another array.. Myintegers is now pointing somewhere else

// Compiler see AnotherVariable is still pointing to old memory to leaves it there

// Generates an IndexOutofBounds Error because AnotherVariable is not resized

var JustChecking = AnotherVariable[5];

# Decision constructs

* If statement – the if statement can be used to allow a certain number of lines of code to only execute if a condition is true. I want to print to the console “x is 5!” only if x is actually 5

var x = 5;

if (x ==5)

{

Console.WriteLine("x is 5!");

}

* If-else statement – the if/else statement can be used to allow an alternate path for your logic. I want to execute certain code of this condition is true. Otherwise, I want to execute this other code.

var x = 5;

if (x ==5)

{

Console.WriteLine("x is 5!");

}

else

{

Console.WriteLine("x is not 5!");

}

* If-else if-else statement – allows for multiple outcome logic scenarios.

var x = 5;

if (x ==5)

{

Console.WriteLine("x is 5!");

}

else if (x <= 10)

{

Console.WriteLine("x is in the range of 0 - 10!”);

}

else

{

Console.WriteLine("x is larger than 10");

}

* Switch statement – use the switch statement when you have the need to provide multiple code paths for the potential values of a single variable. Consider a RoomType of “Queen”. The code below will follow each case down the line until it finds one that matches the current value of RoomType. Notice the lack of meaningful code for the case "Single". This allows you to create the ability to account for different values, but have the same outcode. You can code it so it “falls through” from single to double. A break keyword is required for each case that has meaningful code (not a comment) to tell the compiler that this is the end of that particular code path.

string RoomType = "Queen";

int RoomPrice;

switch (RoomType)

{

case "Single":

// Falls through to double

case "Double":

RoomPrice = 99;

break;

case "Queen":

RoomPrice = 109;

break;

case "Double Queen":

RoomPrice = 119;

break;

case "King":

RoomPrice = 139;

break;

case "Double King":

RoomPrice = 199;

break;

case "Suite":

RoomPrice = 250;

break;

default: // can also have a fallback value

RoomPrice = 99;

break;

}

# Looping constructs

* For loop – the for loop evaluates a set of initialization conditions, and while the conditions are true, each statement in the block below executes repeatedly.

A for loop consists of 3 code sections, all separated by a semicolon:

*(for-initializer; for-condition ; for-iterator)*

In the code below, an ***int*** called ***current*** is created and initialized to zero. This is the first section. The next section is the for-condition. This is a any expression that evaluates to true or false. In this case, ***current < numbers.Length***. The last section is called the iterator. This section merely changes the value of the initializer. You can have loops that:

Count by 1 from zero to some max value.

Count by 2 from zero to some max value.

Count in reverse from some max value down to zero.

The dynamics of the for loop:

First, the initializer is created.

Next, the for-condition is checked. Is 0 less than the total number of items in the referenced array?

If so, the statements are executed.

Then, the following steps are then repeated until the condition fails.

* The iterator is executed which raises the value of ***current*** by 1.
* The condition is then checked again. If the condition is still true, the statements are executed again

int[] numbers = { 1, 2, 3, 4, 5 };

*(for-initializer ; for-condition ; for-iterator)*

for (int currrent = 0; current < numbers.Length; current++)

{

Console.WriteLine(numbers[current]);

}

* Foreach loop – the foreach loop evaluates a block of statements once for each item in a collection. This collection must implement the IEnumerable interface for the foreach statement to work. Each item in the collection is passed into the block of statements as the variable declared in the foreach statement.

In the code below, each number in the array is passed into the statement block as the variable called ***number***

int[] numbers = { 1, 2, 3, 4, 5};

foreach (var number in numbers)

{

Console.WriteLine(number);

}

* While loop – while loop is like the for-loop but without the automatic initializer and iterator functionality.

As you enter the while, the condition is checked. If it is true, the statements are executed. The condition is then checked again to see if it should execute the statements again. It is completely possible that the condition is never true. That is why a while look is referred to as a zero-to-many loop.

Because there is no automatic iterator like the for loop, it is important to place code within the loop that will affect the condition.

int x = 0; // initializer

while (x < 10)

{

Console.WriteLine(x);

x++; // code that affects the condition

}

* Do loop – do loops are like the while loop with one small difference. A do loop executes one to many times whereas the while loop executes zero to many times. A while loop checks the condition before entering the loop statements even once. The do just DOES…then checks to see if it should do it again.

int x = 0; // initializer

do

{

Console.WriteLine(x);

x++; // code that affects

} while (x<10);

# Methods

* Method Signature – A method consists of a signature and a body. The signature consists of an access modifier, return type, method name and parameters

public void SomeMethod(int param1, int param2)

{

// body

}

* Method Overloading – Method overloading is where two or more methods have the same name, but different parameters, either different in number or different in datatype. Overloading methods is a common way to add or remove a parameter from a method without breaking existing code or for making a parameter optional.

public void SomeMethod(int param1)

{

}

public void SomeMethod(int param1, int param2)

{

}

* Optional Parameters – Parameters can also be made optional in C# by simply assigning them a value in the parameter declaration.

public void Count(int startValue =1,int endValue = 10)

{

}

This following method can be called either passing the parameters, thereby causing the default values to be ignored, or without passing the parameters causing the default values to be used in the method.

Count();

Count(1, 5);

Note: Optional parameters must appear at the end of the signature and must not be followed by any required parameters.

* Named Arguments – In cases where you have more than one optional parameter and the developer wishes to provide a value for one but not the other, C# gives us the ability to pass arguments by name. Simply prefix your value with the name of the parameter followed by the colon (“:”) to provide a value for a specific parameter.

Count(endValue: 5);

* Param Array – you may encounter situations where you have an unknown number of parameters which you might need to be able to accept. Forcing developers that call your method to first create an array of strings does work, but it can make things a little complicated. You might wish to just allow them to pass a comma-separated list of items at the end. You can do this by simply adding the keyword ***params*** before the datatype of the parameter

public void ProcessFiles(int processType, params string[] files)

{

}

You can pass an array of strings, or simply just have a comma-separated list of filenames at the end of the parameter list

ProcessFiles(0, new string[] { "file1", "file2", "file3" }); // pass an array

ProcessFiles(0, "file1", "file2", "file3"); // pass a comma-separated list

Note: Param Arrays must appear at the end of the signature and must not be followed by any other parameters

# Value Types vs Reference Types

All variables in C# fall into either the category of value types, or reference types.

**Value type list:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sbyte | short | int | long | byte |
| Ushort | uint | ulong | float | double |
| Decimal | char | datetime |

**And any struct (discussed later)**

**Reference type list:**

|  |  |  |  |
| --- | --- | --- | --- |
| Object | string | interfaces | arrays |

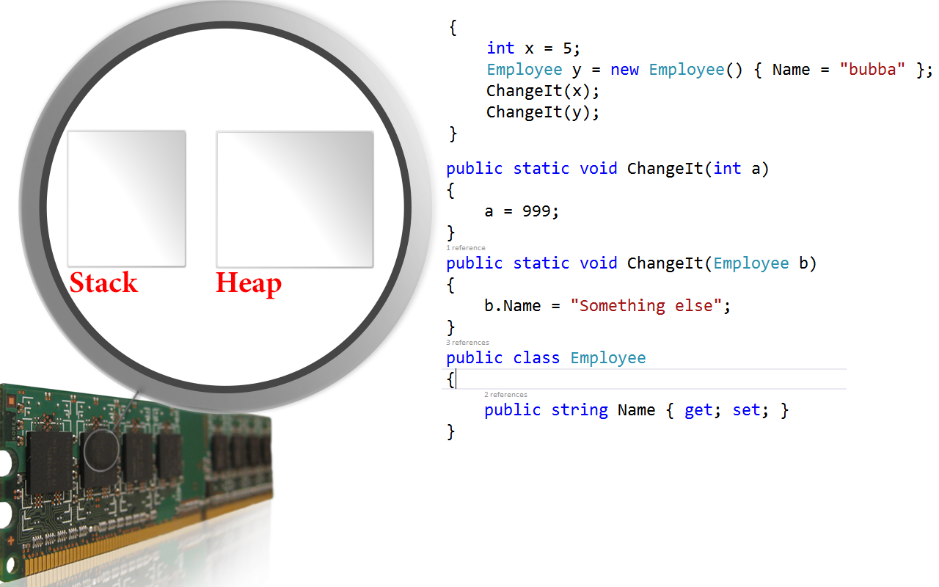
**And any class (discussed later)**

It is important to understand the difference between value types and reference types in terms of how they are dealt with in memory.

.Net uses a data structure called a **stack** to keep track of, among other things, the memory being used in your code. The size of this data structure needs to be large enough to handle the potential amount of memory variables we are tossing around in each application, but small enough to maintain a small footprint and achieve maximum efficiency. That size in a .NET they arrived at is about 1M. This size allows many datatypes to be allocated with all their data on the stack in their entirety, thus achieving the best performance. However, realistically, many of our objects are somewhat large. In fact, a single object could consume memory greater than the entire stack! So for some types, we allocated the data elsewhere is a massive pile of memory called the heap, and we store on the stack merely a reference, or pointer to where this memory lives. We get maximum efficiency for smaller types, and also the freedom to have larger types as well.

The datatypes that are completely stored on the stack are called value types, and the datatypes where just a reference is stored on the stack are called reference types. Their name is referring to “what is stored on the stack”…the entire value? (value types) or just a reference? (reference types).

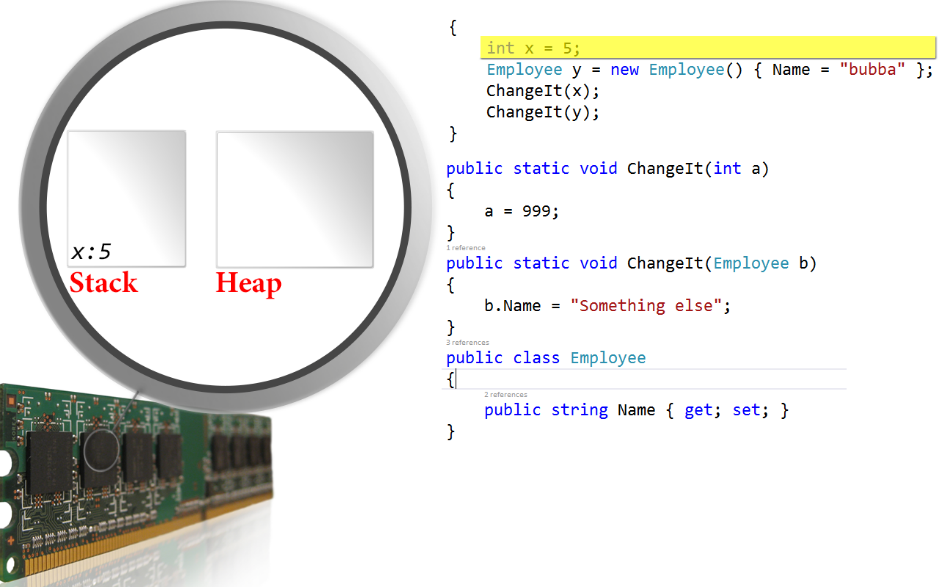
Which one you use makes a difference in how that memory is treated within your application.



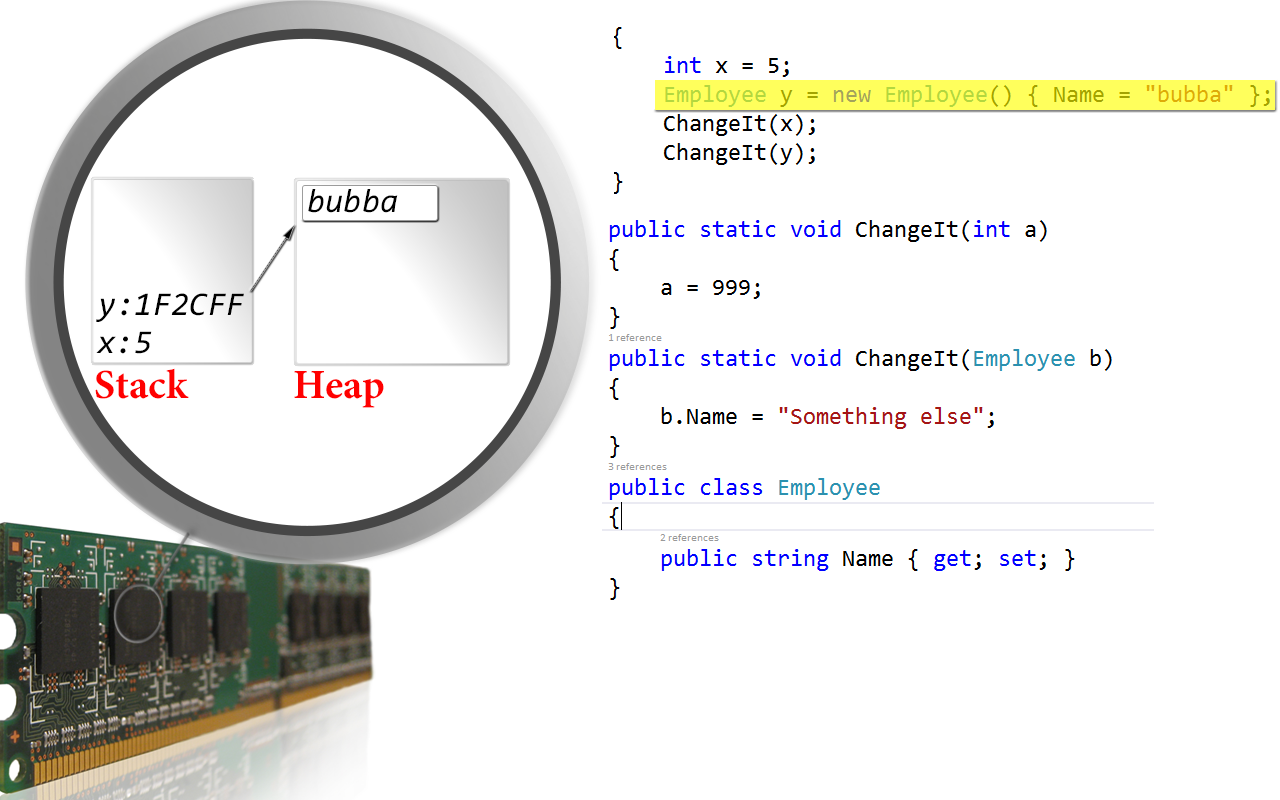
Consider the code above…int is a value type, and employee is a reference type. Both of these are passed into a method and within that method the value of them is being changed.

At the end of this code, will the value of x equal 999? Or will it stay 5? Likewise, will the NAME of our employee stay “bubba”? Or will it change to “Something else”?

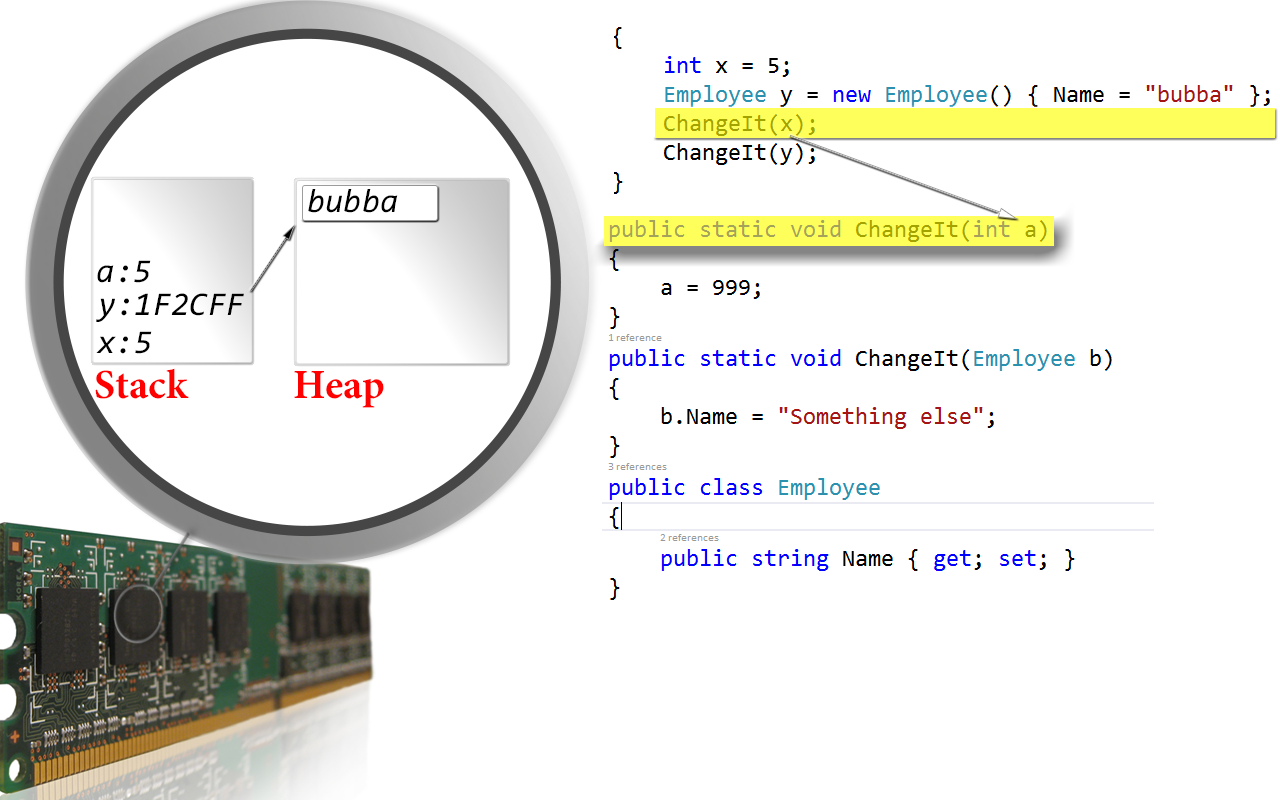
It helps to see what is happening in the memory as this code progresses.



As we execute our first line of code (above), a new memory variable is allocated on the stack, and since this is a value type, the entire value of this object is stored on the stack.

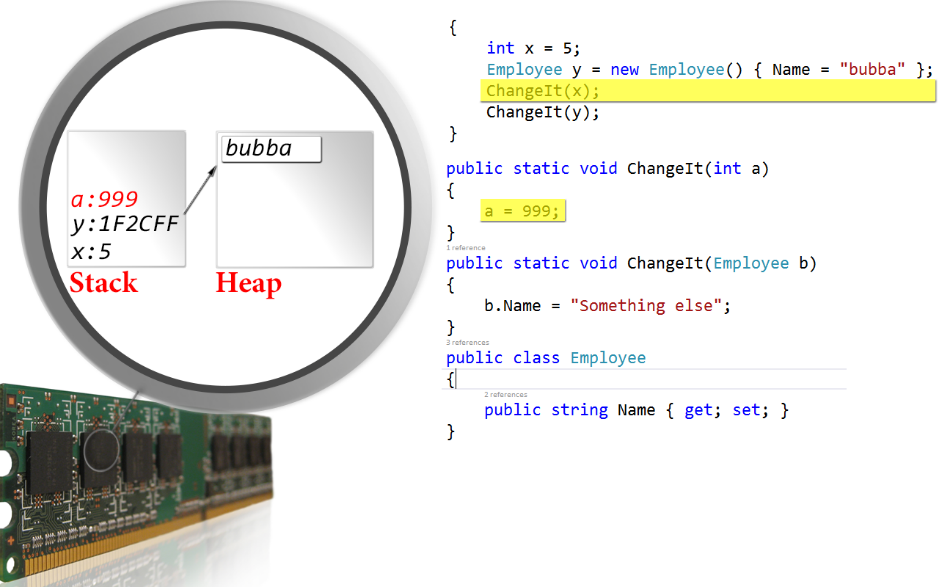


As we execute the second line of code (above), we see a new variable y is placed on the stack, but the actual contents of this object is being placed on the heap. The variable y on the stack merely holds the memory address (called reference) of where this data on the heap exists. It is “pointing” to that memory.

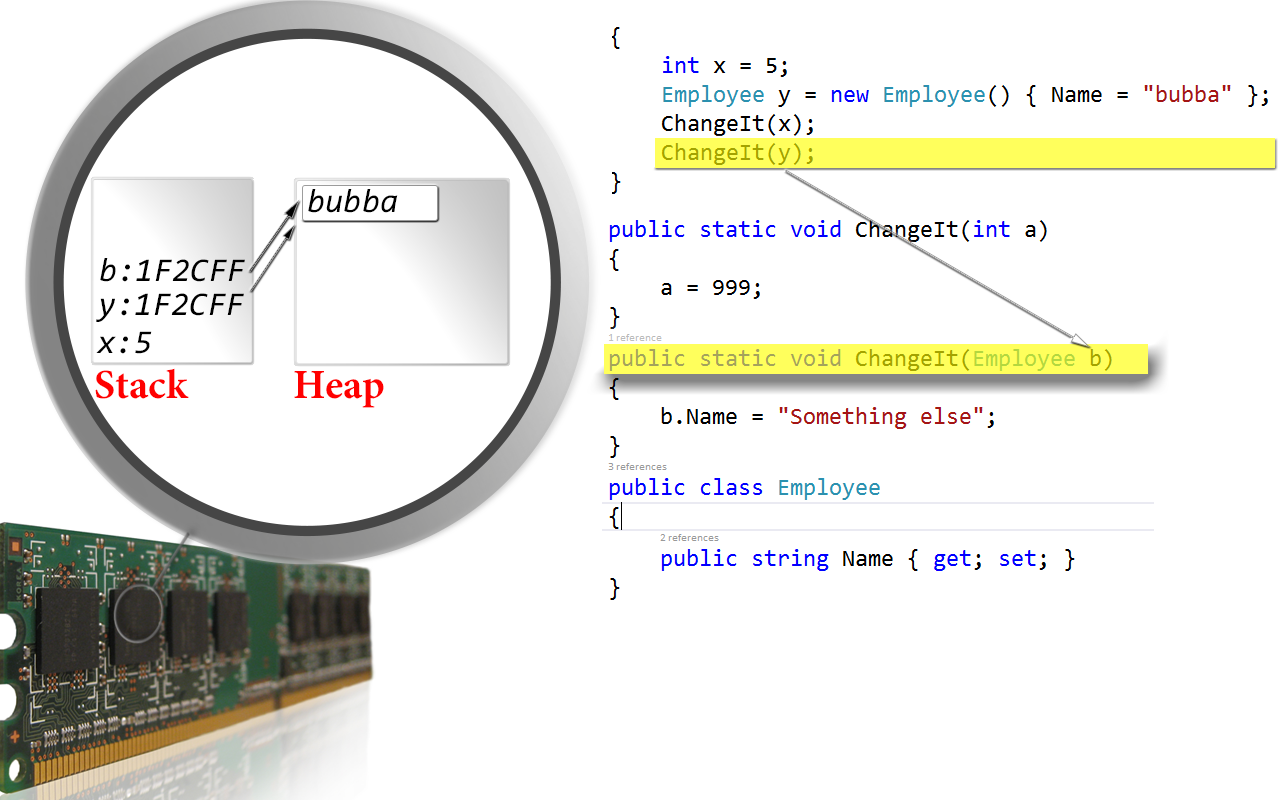


Now as we make the call to the method ChangeIt(), we can clearly see that there needs to be a variable called ***a*** placed on the stack. But where should the value come from? The value of ***x***! So we go to that spot on the stack and copy the value from that spot on the stack into our new variable ***a***.

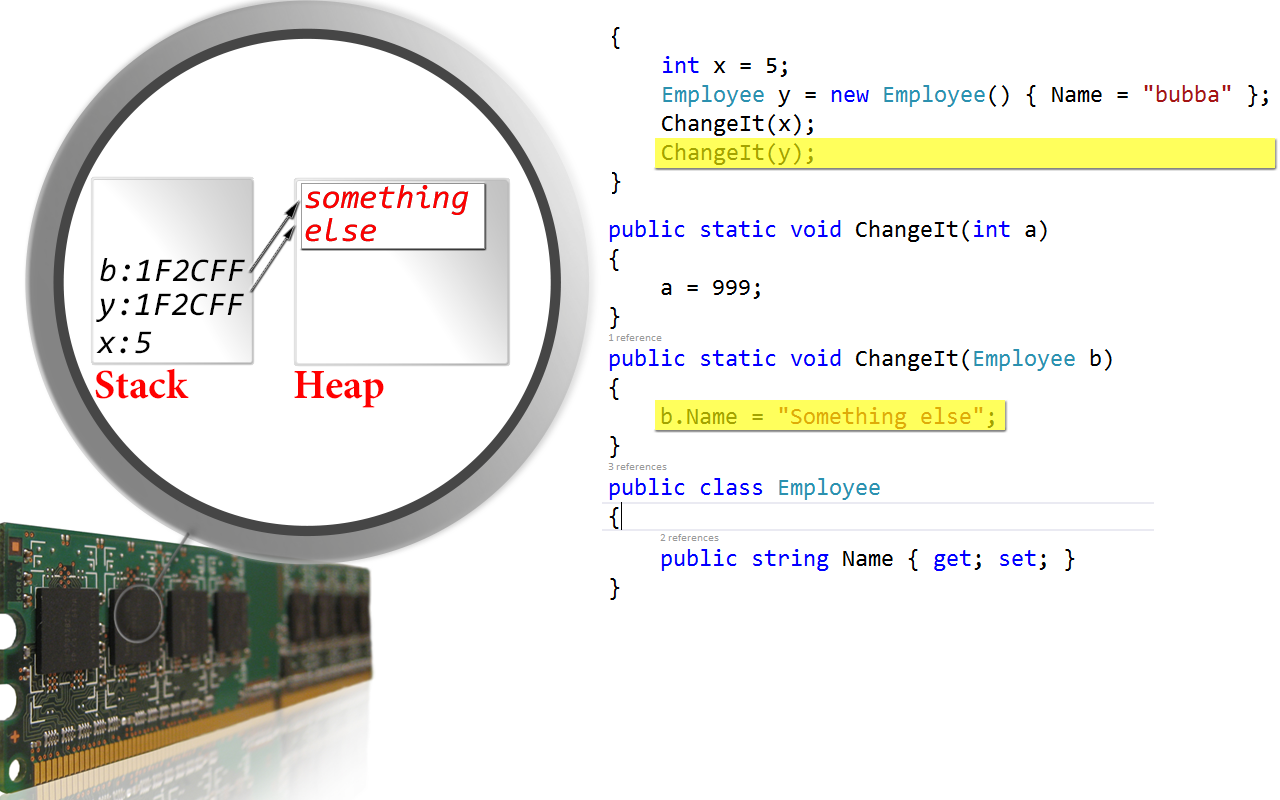
You can clearly see…***x*** and ***a*** are two completely different variables, so you might be able to imagine what is going to happen next.



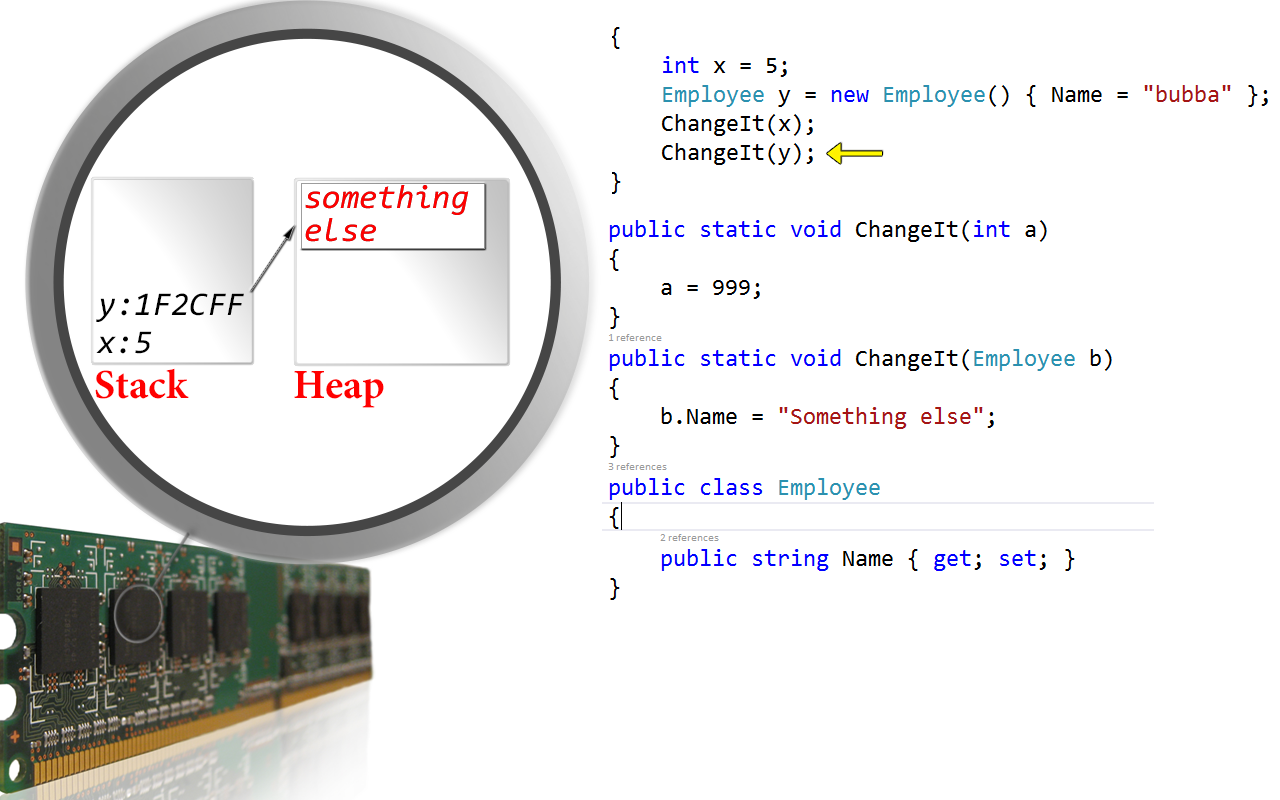
The variable a gets changed to 999, leaving x sitting at 5, just like it was before we started all this nonsense.



Now looking at the last call (above). A variable ***b*** is required. We add ***b*** to the stack, and since ***y*** is being passed into the method, we go to the spot where ***y*** exists on the stack and copy that value from there. But notice now that both ***y*** **and** ***b*** are both pointing to the same piece of memory.



As we execute the code change b.Name to “Something else”, you can plainly see the memory change. And since both ***b*** and ***y*** are pointing to the same piece of memory, they both see the changes.



At the end of the code, we see x has remained the same value it was initially set to. Only the reference type was changed.

Value types are passed by Value by default.

Reference types are always passed by value.

If you want the same functionality from your value type, you must add the ***ref*** keyword:

In the method declaration prior to the datatype

public static void ChangeIt(ref int a)

{

a = 999;

}

Since this is not the normal function of value types, C# requires you to explicitly tell the compiler that Yes, you really did mean to do that. Therefore, you also need the ***ref*** keyword before the ***x*** in the method call

ChangeIt(ref x);

# Exception Handling

* Try/Catch block – gracefully handling errors is as easy as putting the code that could potentially fail inside of a try/catch block. When the error is encountered, control immediately leaves the try block and flows into the catch block.

try

{

// code that could fail

// code that will not run if the above fails

}

catch (Exception ex)

{

// gracefully handle the error

// log it, calm user, offer suggestions and then exit method

}

* Multiple Catch blocks – C# allows you to have more than one catch block to enable handling of different types of exceptions in different ways. You can have several blocks to handle specific errors, and then have an **Exception** block which will catch all other errors.

try

{

// code that could fail

// code that will never run

}

catch (DivideByZeroException ex)

{

}

catch (Exception ex)

{

}

* Working with the Exception class – automatically passed into a catch block is an exception class that contains several properties that give detailed information about the error that just occurred.

try

{

}

catch (Exception ex)

{

var ActualErrorMessage = ex.Message;

var ApplicationRunning = ex.Source;

var MethodRunning = ex.TargetSite;

var EntireStackTrace = ex.StackTrace;

var InnerException = ex.InnerException;

}

* Finally – when using files and database connections, exceptions can cause these to be left open and cause a failure later on in the code. A finally block will always run and is designed to be used to close those files that may have been left open due to an exception occurring before the close code.

try

{

// open file

// code that could fail

// code that will never run

}

catch (Exception ex)

{

}

finally

{

// close the file here

}

* Throwing an exception – there are many cases where communication between layers is done by using exceptions. You can throw an exception in your code when you recognize an invalid scenario by using the throw keyword

throw new ArgumentException("Age cannot be zero");

* Rethrowing an exception – there may be some cases where you are in a catch handler and realize that this is an exception that you should not handle gracefully. This should be sent up the chain to a higher level and you want to rethrow the exception as if you never caught it. Like above, you use the throw keyword, but without any instance

try

{

}

catch (Exception)

{

// Oops, didn't mean to catch this one

throw;

}

# Magic Numbers

“Magic numbers” are hard-coded values that make sense to the developer when they are being used, but can cause confusion months later and to other people trying to read the code. They are to be avoided at all costs. There are three constructs in C# to deal with magic numbers:

* Constants – constants are special variables whose value can only be set at design time and cannot be changed. Constants can make the code give the code more meaning and make it more readable.

public const decimal PI = 3.151592M;

public decimal CalculateCircumference(decimal radius)

{

return PI \* (radius \* radius);

}

Another example:

public const int ID\_COLUMN = 3;

public void Grid\_DblClick(object sender, EventArgs e)

{

var SelectedID = Grid.SelectedItem.Column[ID\_COLUMN];

}

* Readonly variables – there are times when a constant is not a sufficient option because the variable cannot be set until runtime. In this, we can use the readonly keyword to define a variable that is assignable once, either in a constructor or in a variable initializer but after that it behaves just like a constant

public class Employee

{

public readonly DateTime CreatedDate = DateTime.Now;

public int ID { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

}

* Enums – when things are a bit more complex, we have enums. Enums help a lot when we want to make sure we follow the Principle of Least Knowledge.

Principle of Least Knowledge – In order for me to use a thing, I should have to know as little about it as possible.

void SendEmail(int etype, string subj, string recip, string sender, string body)

{

switch (etype)

{

case 0: // html email

case 1: // rtf email

case 2: // plain\_text email

break;

}

}

In the above code, the principle of least knowledge is violated because there is no indication of what constitutes a legal value for *etype*. The datatype of int can be any value between -2.147 billion all the way up to +2.147 billion. But in the case of this method, only 0, 1 or 2 are legal values. The only way to know what constitutes a legal value is to look in the method, and that violates the principle of least knowledge.

This problem can be fixed easily by creating an etype enum and changing the data type to etype instead of int.

public enum EType

{

html,

rtf,

plain\_text

}

And now the method code can look like this, which makes thing much more clear even in the method declaration.

void SendEmail(EType etype, string subj, string recip, string sender, string body)

{

switch (emailType)

{

case EType.html:

break;

case EType.rtf:

break;

case EType.plain\_text:

break;

}

}

At this point, when you tried to call this method, intellisense would automatically limit your choices for the etype parameter to those listed in the enum. It isn’t foolproof, because you still ***can*** pass a value of 9, but at the very least you get help with supplying a legal value.

* Resources – in some cases, you may have a great many constant values which would normally be spread out all over your application, but you would like them all in one place since they are related. As an example, think of validation error messages for every single one of your forms. There may be hundreds, and you might like them all in one place. The easiest way to accomplish this is by using a resource file.

To create a resource file, simple *right-click* on your project and go into *properties*. In the properties for your project there will be a tab along the left side that says **Resources**. You might notice that the dialog says “This project does not contain a default resources file. Click here to create one.” Click and you are ready to go!

At that point, the resources file will show up in the project and will open up so you can add all your strings, text files, audio files, image files, etc. Just keep in mind that these resources are compiled into the EXE so if you put a major amount of data in there, your EXE file will grow as well.

To use the resource file in code, simply use:

Properties.Resources.{resourcekey};

* Project settings – in those cases where you have constant values which you may wish to change without having to recompile the application or maybe you want to keep track of a user’s last five records they opened, you can use the project settings.

To create project properties, simple *right-click* on your project and go into *properties*. In the properties for your project there will be a tab along the left side that says **Settings**. You might notice that the dialog says “This project does not contain a default settings file. Click here to create one.” Click and you are ready to go!

The project properties open up, looking much like the interface given for resources above. Add a setting, giving it a name, datatype, scope and value. Application-scoped settings are read-only within the project while User-scoped settings are saved into a separate settings file per user in the AppData directory for your application. These settings can be changed from within the application.

When you build your application, these settings are stored in the app.config (or web.config for website projects)

Use the settings much like you do the resources:

Properties.Settings.Default.{SettingName};

# Struct and Class

– A struct or class is a special datatype that can be used to model complex data objects, or more clearly, to encapsulate small groups of related variables into one data structure. Structs and classes can both have the same components, all listed below. However, there are a few distinct differences between the two, and these differences will be noted below:

* A struct is **always** a value type.
* All the data in a struct is allocated on the stack which is generally limited to 1 megabyte; therefore, structs should be limited in size perhaps to less than 10 properties/fields or so.
* Since the data in a struct is allocated on the stack, it tends to be much more efficient in terms of allocation and deallocation. The garbage collector does not handle structs. When they go out of scope, they are immediately released back to the operating system.
* When a struct is passed into a method, by default, it is always passed by value. This means that **all** the data within the struct is copied over to the new variable location. In large structs being passed into a method thousands of times, this can be very inefficient. So inefficient in fact, that it more than makes up for the difference in the slow allocation speed of the class. Therefore, again, structs should be limited in size perhaps to less than 10 properties or so.
* There is no user-defined inheritance allowed when using structs, and all structs implicitly inherit from ***System.Valuetype***.
* A struct is created by using the struct keyword as seen below.

Public struct MyStruct

{

}

* A class is always a reference type.
* Classes support inheritance, but only from a single type.
* Classes support polymorphism.
* The data for a class is allocated on the heap and merely a reference (or pointer) to where the data exists in memory is placed on the stack. This two-step process for allocation can make classes much slower to allocate than structs. Classes are ***always*** slower to deallocate since the garbage collector does that work only when it needs to, and then only when that memory is no longer reachable by an in-scope variable.
* Because the data is allocated on the heap, classes can be quite large without causing problems.
* Classes are always passed by reference. Since the only thing you are passing into a method is the address where the data lives, class efficiency in passing variable into methods is much greater than structs in this regard.
* A class is created by using the class keyword as seen below.

public class MyClass

{

}

## Struct or class components

* Fields – fields are variables of any type that are defined within a struct or class. They are sometimes called ***member variables*** or ***backing variables***. Fields are members of their containing type. Fields should be prefixed with an access modifier of public or private in a struct, and public, private or protected in a class. Fields without an access modifier are automatically set to private by the compiler, which is not visually obvious and leads to confusion, so it is recommended to always use an access modifier. You will usually see fields prefixed with an underscore to make them easily identifiable. With the exception of the available access modifiers as mentioned, there is no discernable difference between declaring a field in a class vs. a struct, so only the struct is shown below.

public struct Mystruct

{

// fields

public int \_InternalTemp;

public int \_ExternalTemp;

public DateTime \_CurrentTimestamp;

}

* Constructors – constructors are special methods that are automatically called when you create an instance of either a struct or a class. They cannot have access modifiers or a return type, and they always have the same name as the struct or class they are contained in. Constructors can be overloaded like any other method to provide many ways to initialize the object. Constructors declared in a struct have special rules that do not apply to *constructors declared in classes*:
  1. Default constructors (those with no parameters) are not allowed in a struct.
  2. All constructors must assign a value to all fields in a struct. Always.
  3. Constructors can make assignments to properties; however, in a struct they cannot do so until an assignment has been made to all fields.
  4. The ***this*** keyword which commonly means “the object you are currently in” means something different in the constructor of a struct. Avoid the usage in a struct. (See C# language specification 6.7.6).

public struct Mystruct

{

// fields

public int \_InternalTemp;

public int \_ExternalTemp;

public readonly DateTime \_CurrentTimestamp;

// constructor

public Mystruct(int internalTemp, int externalTemp)

{

\_InternalTemp = internalTemp;

\_ExternalTemp = externalTemp;

\_CurrentTimestamp = DateTime.Now;

}

// overloaded constructor

public Mystruct(int internalTemp)

{

\_InternalTemp = internalTemp;

\_ExternalTemp = 0;

\_CurrentTimestamp = DateTime.Now;

}

}

There is no discernable difference between declaring a constructor in a class vs. a struct. However there are some differences in the way the compiler treats a class that has no constructor declared vs. one where one or more are declared:

1. If you don’t declare a constructor in a class, the compiler will implicitly generate a default construct for you. Remember, a default constructor is a constructor with no parameters.
2. If you declare even one constructor, the compiler will ***not*** generate anything for you, so if you need a default constructor make sure you add one when you add the other constructor(s).

* Methods – all methods are declared within a struct or class and they can be declared as public or private for structs.

* Properties – providing direct access to the fields defined within a struct or class is rarely a desirable thing; however, you often do need to enable a class or struct to expose a public way of getting and setting those values while still maintaining encapsulation so the details behind the scenes remain hidden. This is achieved by using properties. Properties basically break your public access into two components: a get accessor and a set accessor. Whenever the user of your class or struct tries to assign a value to you property, it triggers set accessor, and likewise, whenever the user of your class or struct tries to get the value of a property, it triggers the get accessor.
* Indexers - Indexers are special class member that provides array-like indexing capabilities for easy object property access. The uses for indexers is rather narrow, with the most common one being the functionality of the *IDataErrorInfo* interface used in validation.
* Events/Delegates

Class