

As for what to use when:

* For values which are "naturally exact decimals" it's good to use decimal. This is usually suitable for any concepts invented by humans: financial values are the most obvious example, but there are others too. Consider the score given to divers or ice skaters, for example.
* For values which are more artefacts of nature which can't really be measured exactly anyway, float/double are more appropriate. For example, scientific data would usually be represented in this form. Here, the original values won't be "decimally accurate" to start with, so it's not important for the expected results to maintain the "decimal accuracy". Floating binary point types are much faster to work with than decimals.

**Local functions**

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions

**Static class**

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static>

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members>

**Local Functions**

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions>

* Querying data from a database, IEnumerable execute a select query on the server side, load data in-memory on a client-side and then filter data.
* Querying data from a database, IQueryable execute the select query on the server side with all filters.
* IEnumerable exists in System.Collections Namespace.
* IQueryable exists in System. Linq Namespace.
* Both IEnumerable and IQueryable are forward collection.

|  |  |
| --- | --- |
| **Static Class** | **Non-Static Class** |
| Static class always contains static members. | Non-static class may contain both static and non-static methods. |
| Static class does not contain an instance constructor. | Non-static class contains an instance constructor. |

ECMA – <https://www.ecma-international.org/publications-and-standards/standards/ecma-334/>

-------------------------------------------------------------------------------------------------------------------------------

**Strongly type**

22

For e.g you cannot Multiply or Divide two different types i.e String vs Integer

var answer = 1 \* "1"; // you cannot do this

You have to explicity cast it, this is known as strongly typed

Where as if you see in php

$x = "3" \* 1; // is correct in php

So here you dont need to explicitly cast it.

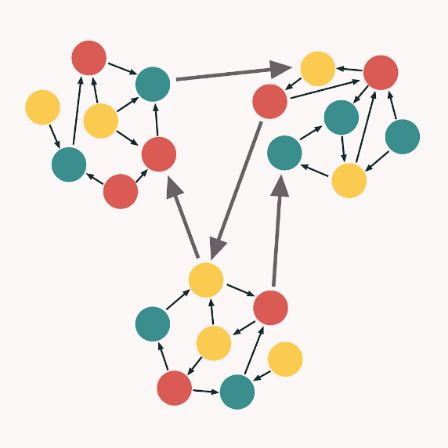
-----------------------------------------------------------------------------------------------------------

.net Framework

<https://en.wikipedia.org/wiki/.NET_Framework>

Binary files - A binary file is a computer file that is not a text file. The term "binary file" is often used as a term meaning "non-text file"

**high cohesion and low coupling**



Array

The array is the single most important data structure in computer programming. The computer itself is nothing but a collection of arrays of switches. A byte is an array of 8 binary bits. An integer (in many languages) is an array of 4 bytes and the order of the 4 bytes that make a 32 bit integer can vary even on a single machine (GPU’s can use different ordering than the CPU’s they talk to). A string is an array of characters (can be one or two bytes per character depending on encoding). All of a computer’s memory is just a big array of bytes (or words that are multiples of a byte).

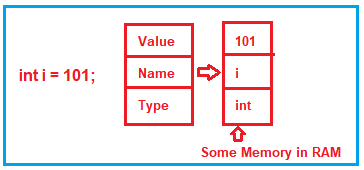
When I first learned about all the advanced data structures like linked lists and binary trees, I assumed that they were what we were supposed to use and they would be fast and efficient. Unfortunately, they are not fast and efficient, and using them often complicates your code when you could have used a simple array. While it may seem harder to use ‘primitive’ structures like arrays, you will be rewarded by much faster performance and in many cases, simpler code. Arrays are also pretty consistent in coding syntax and behavior across all computer languages.

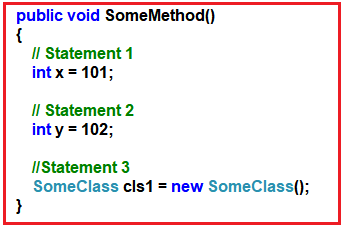
Arrays are fast because they let you directly access a single element’s data in one computer cycle regardless of the size of the array. Because of the operating system and virtual memory and all that, there may be a few more than one actual computer cycle involved to retrieve an element but the time to retrieve the first element is the same as the time to retrieve 100th element and the 500,000th element. No other data structure you build can be faster than an array. All the ‘advanced’ structures are implemented using arrays (and generally badly implemented because they need to be safe across all the different ways they can be used by programmers). If you need a custom data structure, build it on simple arrays and it will be fast.

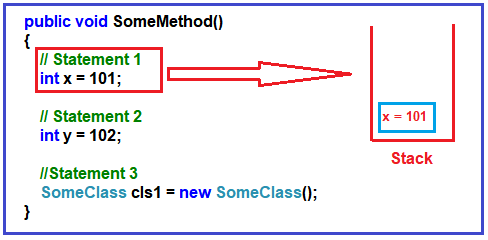
**If you have set of values and the number of values will not change during the current process and you have an integer you can use or calculate easily to reference those values, use an array. You will be rewarded with simpler code that runs faster.**

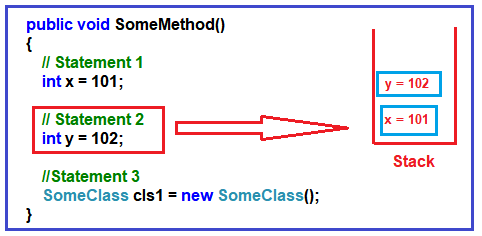
**Value types vs reference types**

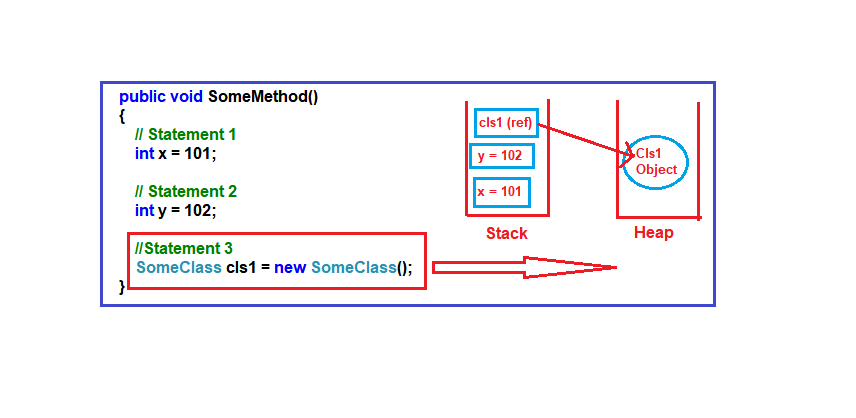
**Stack – LIFO / Heap - FIFO**



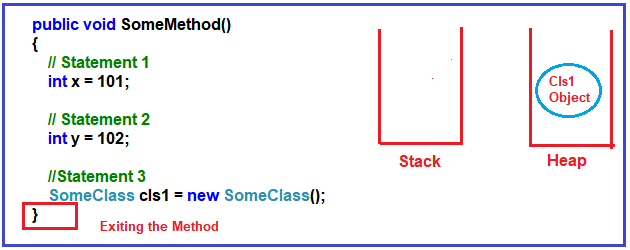


****





**When the three statements are executed, then the control will exist from the method. When it passes the end control i.e. the end curly brace “}”, it will clear all the memory variables which are created on the stack. It will de-allocate the memory in ‘LIFO’ fashion from the stack. For better understanding please have a look at the below image.**

****

**Excute new Keyword – de-allocate**

**It will not de-allocate the heap memory. Later, the heap memory will be de-allocated by the garbage collector. Now you may have one question in your mind is why two types of memory, can’t we just allocate everything on just one memory type?**

**Primitive Data Types**

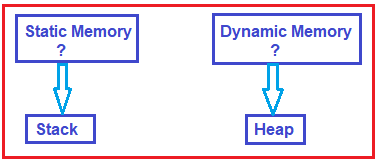
The most famous [primitive data types](http://msdn.microsoft.com/en-us/library/ms228360(v=vs.90).aspx) are: int, [*object*](http://msdn.microsoft.com/en-us/library/system.object(v=vs.90).aspx), *short*, *char*, *float*, *double*, *char*, *bool*. They are called primitive because they are the main built-in types, and could be used to build other data types.

In C#, primitive data types are actually objects, It means when you write the following code, a variable *foo* is actually an Object.

int foo = 10;

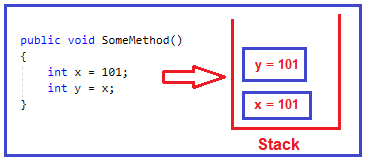
As we know, in C#, the primitive data types such as int, double, bool, etc. they just hold a single value. On the other hand, the reference data types or object data types are complex i.e. an object data type or reference data type can have reference to other objects as well as other primitive data types.

So, the reference data type holds references to other multiple values, and each one of them must be stored in memory. Object types need dynamic memory while primitive data types need static memory. Please have a look at the following image for a better understanding.

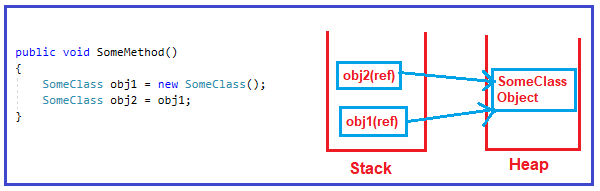


**As we understood the concept of Stack and Heap, Now, let us move forward and understand the concept value types and reference types in detail. The Value types are the types that hold both data and memory in the same location. On the other hand, a reference type is a type that has a pointer that points to the actual memory location.**

**Value types**



**Reference type**



**The memory allocation which is done on the stack is gone when the control moves out from the method i.e once the method completes its execution. On the other hand, the memory allocation which is done on the heap needs to be de-allocated by the garbage collector.**

**When an object stored on the heap is no longer use, that means the object does not have any reference pointing, then the object is eligible for garbage collection. At some point in time, the garbage collector will de-allocate this object from the heap.**

**-----------------------------------------------------------------------**

The last place I had to unbox something was when writing some code that retrieved some data from a database (I wasn't using LINQ to SQL, just plain old ADO.NET):

int myIntValue = (int)reader["MyIntValue"];

Basically, if you're working with older APIs before **generics**, you'll encounter boxing. Other than that, it isn't that common.

**Value type – Heap**

Arrow

Description automatically generated with medium confidence

**Reference ype**

Diagram

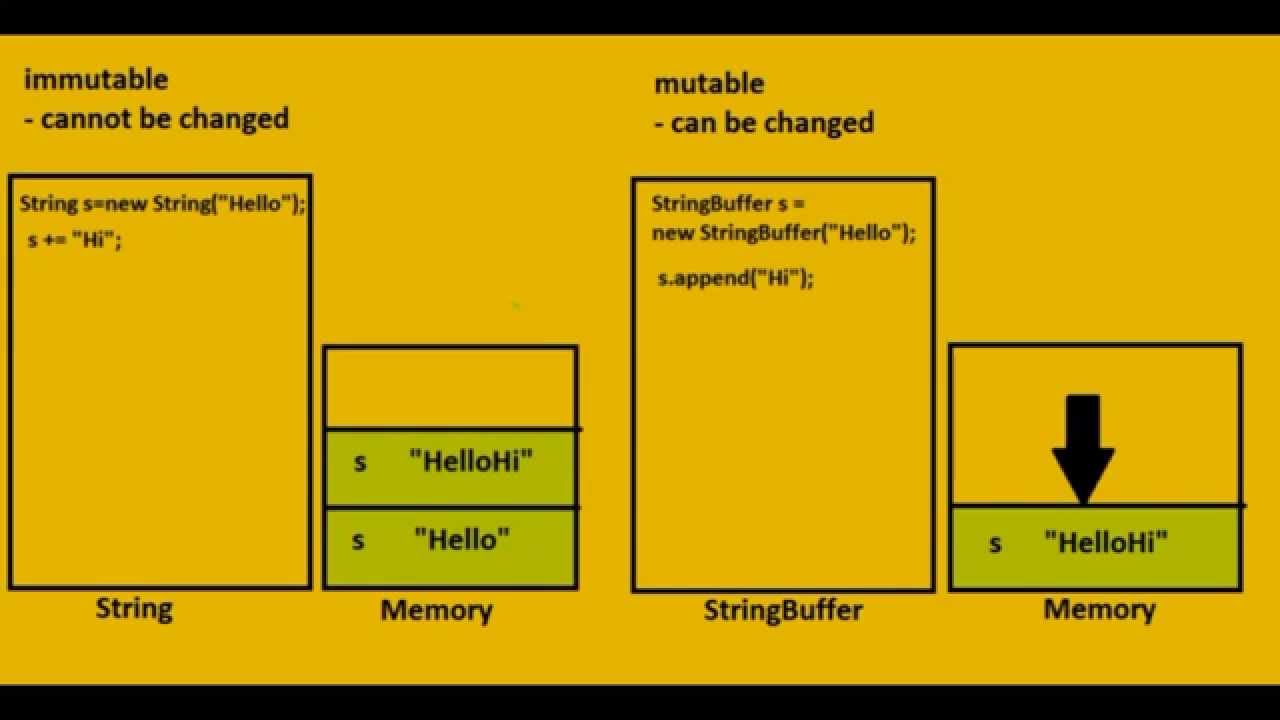
Description automatically generated

**Default value of Reference type = Null**

Diagram

Description automatically generated with medium confidence

**String vs String Builder**



StringBuilder is used to represent a *mutable string* of characters**. Mutable means the string which can be changed**. So String objects are immutable but StringBuilder is the mutable string type. It will not create a new modified instance of the current string object but do the modifications in the existing string object. The complete functionality of StringBuilder is provided by StringBuilder class which is present in System.Text namespace.

**Need of the StringBuilder:** As stated above that the String class objects are immutable which means that if the user modifies any string object it will result into the creation of a new string object. It makes the use of string costly. So when the user needs the repetitive operations on the string then the need of StringBuilder come into existence. It provides the optimized way to deal with the repetitive and multiple string manipulation operations.

|  |  |
| --- | --- |
| Method | Description |
| StringBuilder.Append | Appends information to the end of the current StringBuilder. |
| StringBuilder.AppendFormat | Replaces a format specifier passed in a string with formatted text. |
| StringBuilder.Insert | Inserts a string or object into the specified index of the current StringBuilder. |
| StringBuilder.Remove | Removes a specified number of characters from the current StringBuilder. |
| StringBuilder.Replace | Replaces a specified character at a specified index. |

**Hashtable**

Diagram

Description automatically generated

**Static Method vs Non-Static Method**

**A static method belongs to the class and a non-static method belongs to an object of a class.** The static methods can by accessed directly from the class, while non-static methods (or instance methods as I like to call them) have to be accessed from an instance.

An example would be the static method "Show" from the static class MessageBox . When you need a messagebox, you just call a static method to show it.

**System.Windows.Forms.MessageBox.Show("Halo World!!");**

If it weren't static, you would first have to create an instance of the class that contains it it would be like:

**System.Windows.Forms.MessageBox msg = new System.Windows.Forms.MessageBox();**

**msg.Show("Halo World!!");**

A static method is shared by all instances of the class. Whenever a method is called in C++/Java/C#, an implicit argument "this" reference is passed along with/without the other parameters. In case of a static method call, the "this" reference is not passed as static methods belong to a class and hence do not have the "this" reference.

A non-static method can only be called on an object of a class that it belongs to. A static method can access only static members. A non-static method can access both static and non-static members because at the time when the static method is called, the class might not be instantiated (if it is called on the class itself). In the other case, a non-static method can only be called when the class has already been instantiated.

* **State of an object** - The state or attributes are the built-in characteristics or properties of an object. For example, a T.V has the size, colour, model etc.
* **Behaviour of the object** - The behavior or operations of an object are its predefined functions. For example, a T.V. can show picture , change channels, tune for a channel etc. in object oriented programming terminology the behavior is implemented through methods.
* **Object identity** - Each object is uniquely identifiable. For example, the fridge can not become the T.V.

Property in C# is a member of a class that provides a flexible mechanism for classes to expose private fields. Internally, C# properties are special methods called accessors. A C# property have two accessors, get property accessor and set property accessor. A get accessor returns a property value, and a set accessor assigns a new value. The value keyword represents the value of a property

**Delegates** have the following properties:

1. Delegates are similar to C++ function pointers, but are type safe.
2. Delegates allow methods to be passed as parameters.
3. Delegates can be used to define callback methods.
4. Delegates can be chained together; for example, multiple methods can be called on a single event.
5. Methods don't need to match the delegate signature exactly.
6. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code that can call the referenced method, without having to know at compile time which method will be invoked.
7. An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

**Asynchronuous**

The await keyword provides a non-blocking way to start a task, then continue execution when that task completes

The methods are renamed from their original version to include the "**Async**" suffix.

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

Egg eggs = await FryEggsAsync(2);

Console.WriteLine("eggs are ready");

Bacon bacon = await FryBaconAsync(3);

Console.WriteLine("bacon is ready");

Toast toast = await ToastBreadAsync(2);

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("toast is ready");

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

}

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/>

**Boxing** is the process of converting a value type to the type object or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap.

**Unboxing** extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system, in which a value of any type can be treated as an object.

The advantage of boxing is that you can pass, say, an integer around as an object. The advantage of unboxing is you get you native integer performance back.