C#

REPORT

ANAND PATIKKAL
DEEPTI SATI

Table of Contents

HISTORY OF C#

Once Java was popular, Microsoft wanted to implement their own Java Virtual Machine. But Sun sued Microsoft with the pretext that Microsoft was not implementing the JVM correctly. Then, Microsoft decided to write their own language with a virtual machine concept.

Around 1997, Microsoft started a project that was internally known as Project Lightning. There were several names being considered, one of which was the COM(Component Object Model) Object Runtime (COR). COM objects had been used for development on the Windows platform, ever since it was introduced in 1993. But it was extremely difficult to deploy and manage. 'DLL hell' became a common term for a Windows developer. 'DLL hell' is when your program does not find the 'dll' file to execute certain code. The rift between Sun and the cumbersome Component Object Model, is what drove Microsoft to come up with their own software development framework.

C# is part of a larger framework called the .net framework. This framework has several unique features that are discussed further within this document.

The codename of C# was Project Cool and was going to be a cleaner implementation of Java. It was later changed to C# based on a musical scale. Just as C++ added the "++" to "C" since it was considered to be "adding to" or "one greater than" C, the sharp (#) on a musical scale means one semi-tone above the note. So, in both cases the name implies one above or higher than the original.

Anders Hejlsberg leads development of the C# language, which has a procedural, object-oriented syntax based on C++ and includes influences from aspects of several other programming languages (most notably Java) with a particular emphasis on simplification.

DESIGN

C# is an object-oriented programming language developed by Microsoft as part of the .NET framework. It is the first component oriented language in the C/C++ family. The main goal was to produce a simple, modern and general purpose programming language.

.NET FRAMEWORK

C# is part of a larger .NET framework that provides language independency. It has a large library of pre-coded solutions to common programming problems, the Base Class Library, and manages the execution of programs written specifically for the framework. The .NET Framework is to be used for almost all current Microsoft products. Even Windows Vista was supposedly written in C#.

The pre-coded solutions that form the framework's Base Class Library cover a large range of programming needs in areas including user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications. Programs written for the .NET Framework execute in a similar manner as Java programs. The virtual machine in .NET is called the *Common Language Runtime* (CLR). It is identical to JVM in several aspects.

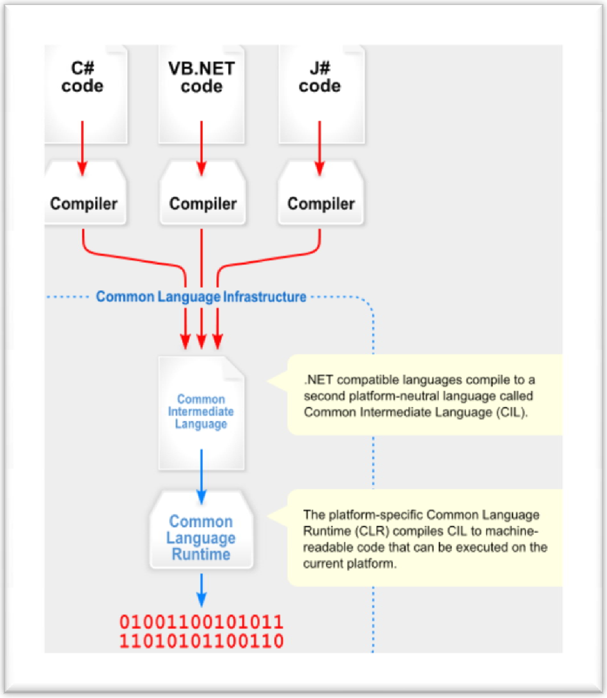


Figure: Reference from Wikipedia

.NET Framework Goals

One of the primary goals of .NET was language independence. Prior to .NET, Windows Development primarily occurred in Visual C++ and Visual Basic. As most programmers would stick to their language, VC++ or VB, support for such programmers had to be provided in the .NET framework. Hence, language independence was introduced.

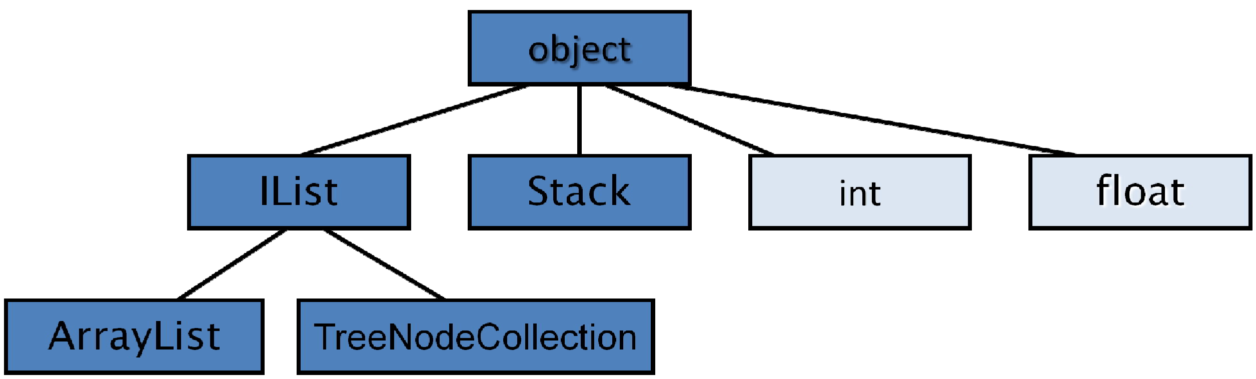
In .NET suppose you write a program in VC++ and compile it as a '.dll' (library files), then this file can be added as a reference in C#, and all the classes, functions and objects of that library can be used in C#. This is a unique and a very useful feature for current and future windows developers. For instance, VC++ can be used to write several pointer arithmetic programs, which can then be easily used in C#.

The key requirement for language independence is that all the languages in the .NET framework have a Common Type System (CTS). The CTS specification defines all possible datatypes and programming constructs supported by the CLR and how they may or may not interact with each other.

Programming languages on the .NET Framework compile into an intermediate language known as the *Common Intermediate Language* or CIL, similar to bytecode in Java. This intermediate language is not interpreted but rather compiled in a manner known as just-in-time compilation (JIT) into native code.

C# DESIGN

In C# everything is an object. Traditionally, primitive types do not interoperate with objects. This is the case in C++ and Java. For example in C++ you need to convert a char* to a "string" object so that strings can be used in the Standard Template Library (STL) classes. In Smalltalk and Lisp, everything is an object, but at a great performance cost.



As seen in the figure above, everything is inherited from the 'object' class. Most of C#'s primitive types are just an alias to their object type. For instance:

int == System.Int32

double == System.Double

The C# language specification does not state the code generation requirements of the compiler: that is, it does not state that a C# compiler must target a CLR, or generate *Common Intermediate Language* (CIL), or generate any other specific format. Theoretically, a C# compiler could generate machine code like traditional compilers of C++ or FORTRAN. But in practice, all existing C# implementations target CLI.

There are no global variables or functions. All methods and members must be declared within classes. It is possible, however, to use static methods/variables within public classes instead of global variables/functions. Local variables cannot shadow variables of the enclosing block, unlike C and C++. C# is more typesafe than C++. The only implicit conversions by default are those which are considered safe, such as widening of integers and conversion from a derived type to a base type. This is enforced at compile-time, during JIT, and, in some cases, at runtime.

PROGRAM STRUCTURE

Here is the structure of a Hello World program, in C#.

```
using System;                          // No header files
using System.Collection;               // This is part of BCL, similar to STL in C++

namespace MySpace                      // Everything is in a namespace

{
class Hello                            // Everything is within a class
   {
      static void Main() {             // Static main, similar to Java

         Console.WriteLine("Hello world");   // Same as printf() in C

      }
   }
}
```

TYPE SYSTEM

C# has a Common(or Unified) Type System (CTS). This is required to provide language independence. There are two types in C#:

1. Value Type
2. Reference Type

Value Type

Value type includes the following:

- Primitives - int, float, double, etc.
- Enums
- Structs - unlike C++, struct is a value type

Reference Type

Reference type includes the following:

- Classes
- Interfaces
- Arrays
- Delegates

Boxing and Unboxing

Boxing and unboxing are key aspects of C#, that work in conjunction with the Common Type System. Boxing is used to convert a value type to a reference type and unboxing is used to convert a reference type to a value type.

```
int i = 123;
object o = i;
int j = (int)o;
```

In the code above, the primitive int type variable is automatically boxed to an object type by the compiler at runtime. Similarly the object is unboxed to an int primitive at runtime. In Java, this has to be explicitly specified using the wrapper classes, which is something a programmer easily forgets, and which also lengthens the code unnecessarily.

Consider an Add() method to a hash table in C#, as given below:

```
Add(object key, object val);          // Signature for hash table Add method
```

In C# code, you can write:

```
Hashtable h = new Hashtable();

h.Add(5, “five”);
```

As you can see, here the compiler automatically boxes the int and the string types, to an object type. This is extremely useful, and adds a touch of generics.

C# is more strongly typed than C++, almost same as Java. Another feature of C# is that all variables are initialized. For instance, all strings are initialized to null in the following code:

```
string  a, b, c;
```

This again makes the code more elegant, and also prevents the programmer from making silly mistakes.

UNIQUE FEATURES

PROPERTIES

Properties provide a neat and easy interface to access the fields of a class. They are used to set and get member fields of a class.

```
public class MyClass
{
    private string myString;            // Class member myString
    public string MyString {            // 'MyString' property for class member myString
        get {                           // get method used to return the myString value
            return myString;
        }
        set {                           // set method used to set the value for myString
            myString = value;           // value is predefined by the compiler
        }
    }
}

MyClass object = new MyClass();
object.MyString = "PPL";                // Set myString field using the property MyString
String str = object.MyString;           // Get myString field using the property MyString
```

Properties can be very powerful, as you can use complex get and set methods. They also improve the readability of the code.

INDEXERS

Indexers allow your class object to be referenced as an array.

```
public class ListBox
{
  private string[] items;
  public string this[int index] {
    get {
      return items[index];
    }
    set {
      items[index] = value;
      Repaint();
    }
  }
}
```

Indexers are similar to properties, but the difference is that indexers are for arrays. Also, indexers can be overloaded based on the return type and the index argument. For instance the following indexer is different than the one above:

```
public int this[int index]
```

DELEGATES

Delegates are similar to function pointers in C/C++. The additional feature of delegates is that each delegate has a subscription list. When you call that delegate then all the functions within the subscription list are called. Also, functions can be added or removed from the subscription list at runtime.

```
public delegate void LogHandler(string message);     // Define the Delegate

public void ConsoleLog(string s){                    // Subscriber 1
        Console.WriteLine(s);
}

public void FileLog(string s){                       // Subscriber 2
        File.WriteLine(s);
}
```

```
LogHandler myLogger = null;                     // Declare the Delegate
myLogger += new LogHandler(ConsoleLog);         // Add to Subscription List
myLogger += new LogHandler(FileLog);            // Add to Subscription List
myLogger(message);                              // Call Subscribers
```

In the example above there are two subscribers, a console logger and a file logger. When the delegate variable is invoked, then all the subscribers are called. You can also remove functions from the subscription list as follows:

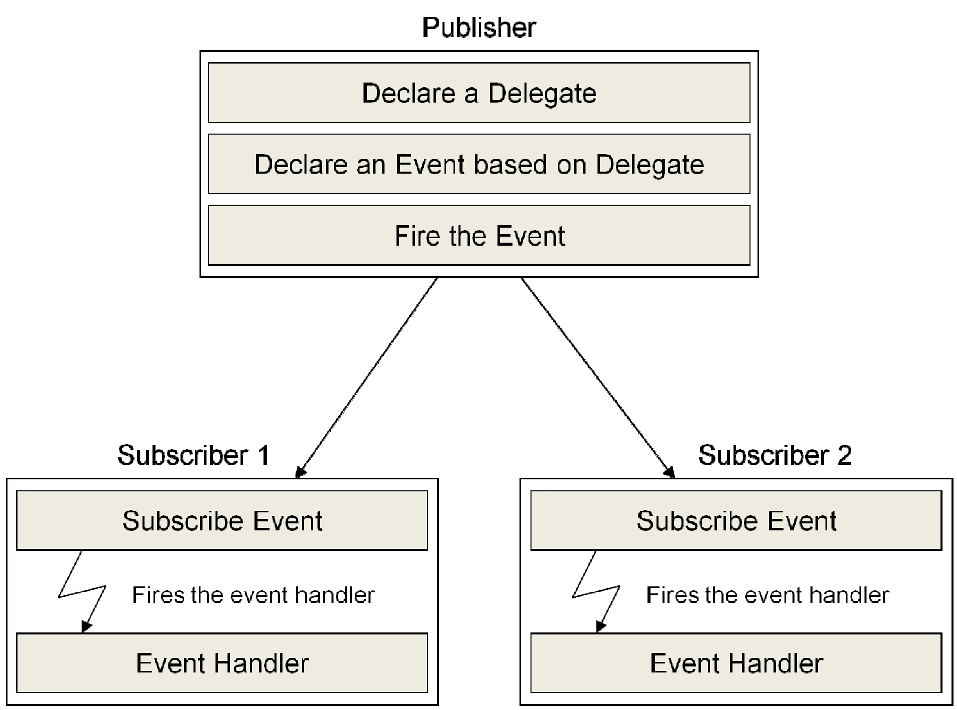
```
myLogger -= new LogHandler(ConsoleLog);         // Remove from Subscription List
```

Delegates are a strong feature of C#, especially when they are combined with events.

EVENTS

Events work in conjunction with delegates. The steps involved in how events work is as follows:

1. An event occurs
2. A firing logic is triggered
3. The firing logic in turn calls all the subscribers



An event is declared by a publisher, and several event handlers subscribe to this event. When that particular event occurs, the firing logic will call all the subscribers.

Here is a sample code for an event logger:

```
public delegate void LogHandler(string message);    // Declare a delegate
public event LogHandler Log;                         // Declare an event based on the
delegate

protected void OnLog(string message)                 // Firing logic
{
  if (Log != null)
    Log(message);
}

static void ConsoleLog(string message) {             // Event Handler
  Console.WriteLine(message);
}

Log += new LogHandler(ConsoleLog);                   // Subscribe to the Event Handler
```

ITERATORS

Iterators are similar to generators in other languages like Python and Ruby. Suppose your class behaves like a collection and you need to iterate through the members of your class object, then you use iterators.

To implement iterators in C# you need to implement the IEnumerable interface. In this interface you have to atleast implement the GetEnumerator() function. Here is an example:

```
public class MyList : IEnumerable<string>
{
    public IEnumerator<string> GetEnumerator()
    {
        foreach (string s in strings)
            yield return s;
    }
}
```

```
// Main program
foreach ( string s in mylist_object )
{
    Console.Write(s);
}
```

When you implement the IEnumerable interface you have to specify the individual item type of your collection class. In the example above, it is a List collection class, where the list item is of type string. Here the key is the “yield return” statement. When you do a foreach on your class object, then the yield will return the items one after the other.

UNSAFE CODE

C# also supports the usage of pointers. The syntax and usage are very similar to that of C/C++. In any block of code where you want to use pointers, you have to prefix the class definition, function definition or the code block with the keyword ‘unsafe’. Casting and pointer arithmetic is the same as in C/C++.

```
unsafe
{
    int * ptr, also_ptr;            // Both the variables are pointers, unlike C/C++
    ptr = &(new Int32(5));
    *ptr = 10;
    fixed (long *ptr_a = &(myClass.a))
}
```

In the example above, the code block is preceded with the keyword unsafe. When you specify ‘int *’, then the ‘*’ is attached to int and all variable names declared after that are all of ‘int*’ type. This is unlike C/C++ where you have to specify ‘*’ before each variable declaration.

One problem with pointers in C# is that the garbage collector might reallocate objects. Hence a pointer that might point to a valid object at one time, but the GC might reallocate that object, and the pointer might point to junk after that. For this, you have to use the keyword ‘fixed’ before the pointer variable declaration.

C# also allows you to allocate memory off the stack using the keyword ‘stackalloc’. This memory is not initialized, so this allocation is slightly faster.

GARBAGE COLLECTION

C# supports implicit and explicit garbage collection. The implicit garbage collector is implemented in the CLR using the Mark and Compact algorithm. In the Mark phase the live objects are found by traversing the heap from the roots. In the Compact phase the following steps are taken:

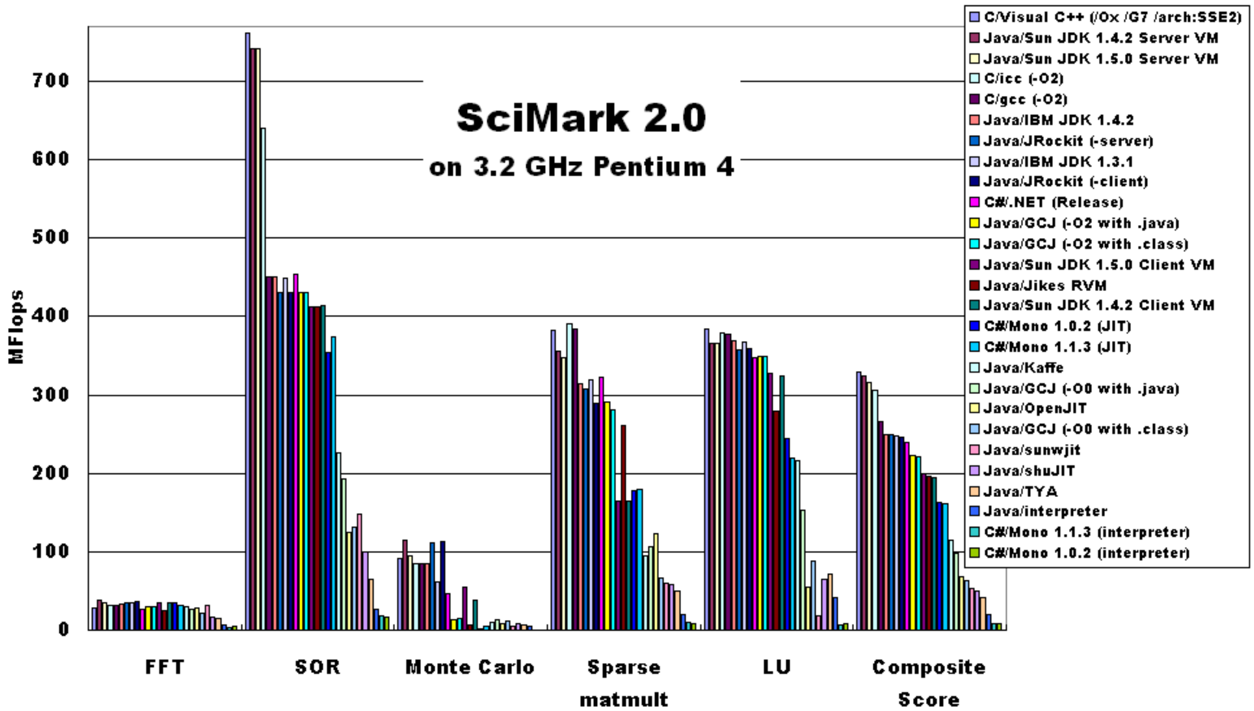
1. The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects, which is now free space.
2. The garbage collector then shifts the non-garbage objects down in memory, removing all of the gaps in the heap.
3. Moving the objects in memory invalidates all pointers to the objects. So the garbage collector modifies the application's roots so that the pointers point to the objects' new locations.
4. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well.

After all the garbage has been identified, all the non-garbage has been compacted, and all the non-garbage pointers have been fixed-up, a pointer is positioned just after the last non-garbage object to indicate the position where the next object can be added.

For explicit garbage collection you need to call System.GC.Collect(). This calls each unused object finalizer(destructor) on separate threads. Then you can optionally call System.GC.WaitForPendingFinalizers(), to wait until all the finalizers have completed.

CONCLUSION

Recent performance analysis of various JVM's(including IBM and Sun), C# .NET CLR and C/C++ compilers has shown that the Java Development Kit from IBM performs slightly better than the C# .NET CLR. But the .NET CLR runs much faster than the Sun's JVM.



Reference from http://www.shudo.net/jit/perf/#scimark2

Here is a graph that shows various highly compute intensive kernels that were measured on several compilers.

Installation of software on Windows has always been a pain, especially with COM objects and the Windows registry. Moreover, the registry is hardly maintained properly by any software, and this causes several errors as your Windows grows old. Installation of computer software must be carefully managed to ensure that it does not interfere with previously installed software, and that it conforms to increasingly stringent security requirements. The .NET framework includes design features and tools that help address these requirements. Moreover it relieves the developer from maintaining things in the Windows registry. Another problem with C# is that initializing all the variables makes the allocation slower.

C# has been quite successful, but this is mainly because of the monopoly of Windows. Currently if you need to develop programs on Windows, then the easiest way is using C#. Due to the same reason, .NET CLR has not been implemented for other operating systems. For Linux you have an open source CLR, called Mono, which is not that widely used. Another reason why C# has not been adopted on a larger scale is due to the large user base of Java. Several current applications are written in Java and to rewrite them in C# is a huge ask.

Overall, C# provides excellent features and comes with easy to use software development tools, like Visual Studio. It is definitely a language that is to look out for, especially when you can have such a huge target audience of Windows users.