

Thomas Barker  
Jonathan Phillips

# **C# vs Java**

## **Abstract**

Both Java and C# are object oriented languages that have evolved from C++. Some even claim that C# is an evolution of Java. From a quick glance, the languages are very similar and essentially accomplish the same things. The point of our research is to focus on the nuances of the languages from an architectural perspective. Despite the marketing war between Java and Microsoft, or more generally, open source vs. big brother, we will take an objective view of the two languages to identify circumstances where one language might be ideal to the other.

## Table of Contents

<b><i>Introduction</i></b> _____	<b>4</b>
<b><i>Code Structure</i></b> _____	<b>5</b>
Pass by reference _____	5
<b><i>How Does it Work – Compilation and Execution</i></b> _____	<b>6</b>
<b>Overview – The Benefits of an Intermediate language</b> _____	<b>6</b>
<b>JVM</b> _____	<b>7</b>
Overview _____	7
Loading Classes in the JVM _____	8
The Loading Phase _____	8
The Linking Phase _____	9
Verification _____	9
<b>.NET</b> _____	<b>10</b>
The Common Language Runtime _____	10
Attributes _____	11
<b>Just In Time Compiler (JIT) – Compiling to Native and Running Code</b> _____	<b>11</b>
<b><i>Mobility</i></b> _____	<b>12</b>
Is C# and the .NET framework only available for PC architecture? _____	12
<b><i>Throwing out the Garbage</i></b> _____	<b>12</b>
<b>Throwing out the garbage the .NET way</b> _____	<b>13</b>
Reserving address space _____	13
Place an object on the heap _____	13
Garbage Collection – How it Works _____	14
Garbage Collection – Optimization _____	15
<b>Throwing out the garbage the JVM way</b> _____	<b>16</b>
The Old and the New _____	17
<b><i>IDEs – What is the Difference</i></b> _____	<b>17</b>
<b><i>Conclusion</i></b> _____	<b>18</b>
<b><i>Who did What</i></b> _____	<b>21</b>

## ***Introduction***

---

To accomplish our goals, we will look at 4 aspects of the languages, primarily focusing on their relationship to a computer's architecture. Our research will focus on code structure, code compilation and execution, mobility, and memory management.

### **Code Structure:**

We will identify key differences in code structure between the two languages. We will avoid basic syntax and semantic differences and focus on structural differences that have a meaningful impact in the programming environment and performance.

### **Code Compilation and Execution:**

We will study the Java Virtual Machine, how it works, and how it allows for cross platform compatibility. While studying how the Java Virtual Machine works, we will see how java programs are compiled from a high-level language into java bytecode, and how that in turn is interpreted and executed by the JVM.

For C#, we will look at how code is managed in the .NET framework. This will allow us to explore some powerful features specific to .NET such as the CLR (Common Language Runtime).

### **Mobility:**

Java runs on the Java Virtual Machine which allows cross platform compatibility, a feature that C# currently does not have without some difficulty. Sun officially supports Linux, Windows, and Solaris but there are third party vendors that have made java available on other systems such as MacOs. This means that a programmer can write a program once in any platform (Linux, Windows, MacOs, etc) and be able to compile and run the program on any other platform without making any changes. Although there are solutions to make C# platform independent, C# primarily runs natively on PCs. Unlike Java, it is a multi-language platform; a feature that attempts a similar idea to Java's "Write once run anywhere..." philosophy, but concentrates on multiple languages not multiple platforms.

### **Memory Management**

How do JVM and the .NET framework manage memory resources and handle garbage collection? Though the overall processes and purpose is the same, key differences exist that can meaningfully impact performance.

## *Code Structure*<sup>1</sup>

---

In terms of structure, C# and java are very similar. You could easily transfer a program from Java to C#. Of course there might be different names for the compliment classes, but they almost always perform the same task. The main differences that impact performance involve extra functionality in C# relate to memory management and metadata.

Java almost isolates the programmer from memory entirely, while C# gives you so much flexibility that you could almost morph any C# program into something that closely resembles memory management in a C++ program. Examples of some of these deviations from Java are listed below.

### **Pass by reference**

In C#, you can pass any type by reference by using the ref keyword. For instance, the method call

```
foo(ref int number1, ref int number2)
```

would pass both number1 and number 2 as references as opposed to value.

### **Deterministic Object Cleanup**

You can essentially override the garbage collector and manage your own garbage clean up when you want. You accomplish this by inheriting the IDisposable interface and overriding the Dispose method.

### **Unsafe code**

Unlike Java, C# allows unsafe code and pointer manipulation

### **Attributes**

This is related to the metadata of an executable and will be covered later in the paper

Additionally, there are a few differences between Java and C# that are not related to memory management that are worth discussing. These include:

---

<sup>1</sup> For more information about syntax and code structure of C#, please refer to the O'Reilly book [Programming C#](#) by Jesse Liberty.

## Structs

Structs are a simpler version of a class. They can do just about anything that a class can do except inherit another class or interface. Additionally, they are a value type, not a reference type.

## Preprocessor directives

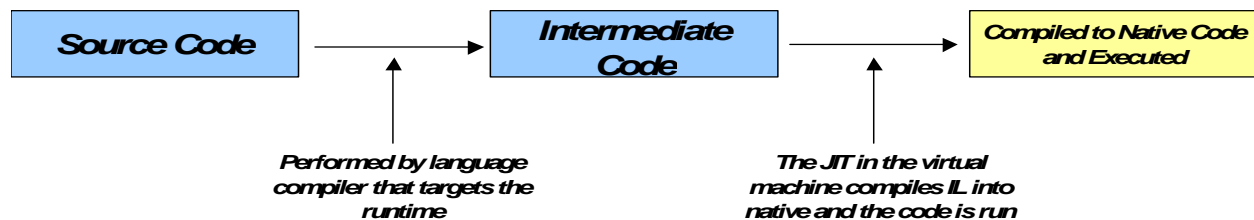
C# has a limited collection of preprocessor directives like C and C++. Some of these include #if, #elseif and #define. C# does not use #include.

## *How Does it Work – Compilation and Execution*

---

### **Overview – The Benefits of an Intermediate language**

Overall, the general process of compilation and execution is the same for Java and C#. The figure below shows the general process that both of these languages go through for compilation and execution.



Both languages compile their source code into an intermediate language. This immediate language, Microsoft's intermediate language (MSIL) for Microsoft and bytecode for Java, are an abstraction of the CPU. Both are low level languages that are similar to LC3. By compiling to an intermediate language, both languages succeed in creating a CPU independent platform. The JVM exploits this framework by creating an architecture independent platform. So any language that targets the JVM bytecode can run on virtually any commercially available platform. Currently there are a few languages that target the bytecode, though the primary one, which the Java Virtual Machine was originally created for, is Java.

Unlike JVM's focus on platform independence and despite having created a CPU independent paradigm, .NET focuses on making their platform available to a myriad of languages from C# and Python to Cobol and Fortran. Additionally, extra information is stored into the assemblies, or executable code, to facilitate language interoperability. This means a

language written in Fortran could access information in an executable written in C#. Also, any code that targets MSIL will benefit from memory management and verification.

In the following sections, we will go under the hood of the java virtual machine to get a comprehensive view of how a virtual machine manages code. The second section will discuss .NET's common language runtime (CLR).

## **JVM**

### ***Overview***

All CPUs perform the same basic operations as one another. However, programs designed for a specific CPU will not perform on another CPU. Since there are so many different types of processors and computers, it is a major problem for developers to write general-purpose programs for everybody. The developers of Java decided to try to solve this problem and created the Java Virtual Machine. The JVM is an abstraction of a CPU which creates a virtual computer that is implemented on top of the physical machine. This implementation allows the JVM to be installed on any machine and a program that is written in Java<sup>2</sup> can then be run anywhere that has the JVM installed. Depending on the architecture of the computer, a different implementation of the virtual machine is required. An example of one such implementation of the JVM is Sun Microsystems Java Development Kit (JDK), which runs on Solaris and Windows operating systems.

In order to run a Java program a compiler translates the Java code into class files, which can then be used by the JVM. The data in a class file is not necessarily stored in a file; it can be stored in a database, over a network, as part of a Java archive file (JAR), or a number of other ways. Java uses a class called `ClassLoader` which contains subclasses that can load the data from a class file that is stored in any of the aforementioned ways. The instructions created during the compilation process are expressed in bytecodes which resembles the ISA of most processors. A major difference between the JVM and the ISA of most CPUs is how each interacts with memory. The ISA of most CPUs treat memory as if it were a large array of bytes while the JVM treats memory as a collection of objects, which are used in Java's object oriented paradigm. Treating memory in such a way allows the JVM to use a stack-based architecture where objects

---

<sup>2</sup> Actually it supports any language that targets the JVM bytecode

are pushed onto the stack as they are created and popped off as they are used and no longer needed.

The JVM is conceptually divided into four separate spaces; the class area, the Java stack, the heap, and the native method area. The class area, where the code and constants are kept, is divided into a method area where method implementations are kept and a constant pool where constants reside. In order to make the JVM more stable, all of the properties of a class, once created, are immutable. This encourages consistency amongst the methods and objects of a particular class. The methods of a class will have the same code each time they are invoked and all of the objects of the class will have the same fields.

The Java stack holds methods that have been called along with the data associated with each method. When a method is called a data space known as a stack frame is created. The collection of stack frames for a program is known as the Java stack. Like stacks we have studied, the latest method invocation is placed on the top of the Java stack until it is done executing, then it is popped off of the stack and the stack frame below picks up where it left off execution. Stack frames contain an operand stack, an array of local variables, and a program counter which points to the currently executing instruction in the stack frame.

The heap area stores objects which are associated with a class in the class area of the JVM. There is also space for storing the nonstatic fields of a class, as well as all of the nonstatic fields in a classes superclass and so on. The final area of the JVM is the native method stack. This is where methods that are not written in Java are created and used. Native methods usually use a stack to keep track of their own state when they are compiled into machine code.

### ***Loading Classes in the JVM***

Classes in Java are loaded in two phases: the loading phase and the linking (also known as resolution) phase. Java has the power of reflection, which allows a program to examine itself. Through reflection, Java can find out the fields and methods of a class and it can even invoke methods and read or assign fields. The ClassLoader uses reflection in the loading phase to translate the class file into bytecode, a format the JVM can understand.

### ***The Loading Phase***

During the loading phase, the bytes from a class file are loaded into the JVM, where the ClassLoader loads the superclass of the superclass and so on. Subclasses of ClassLoader

implement different ways of loading class files from different locations; on the disk, in a database, over the network. The loadClass method starts the ClassLoader.

```
Class loadClass(String name, boolean resolve);
```

The name is the name of the class to be loaded, and the resolve tells whether or not to proceed to the linking phase. Then the defineClass method checks to make sure the class file is properly formatted and it calls loadClass to load the superclass. The loadClass function loads the superclass which uses the defineClass to define it until the ClassLoader finds java.lang.Object or it detects it has gone in a loop. If ClassLoader finds itself in a loop a ClassCircularityError is thrown. After this process is complete defineClass returns a Class object. The JVM now knows the name of the class being loaded as well as all of its superclasses, and the names and types of its fields and methods.

### ***The Linking Phase***

In the linking phase, a class is verified to make sure it meets certain criteria and is initialized. If the parameter resolve in the loadClass method is true, the resolveClass method is called, which verifies that classes obey certain rules. Some of the rules that classes must follow are all methods required by interfaces are implemented, instructions use constants that exist and that are of the correct type, methods don't overflow the stack, and methods don't try to use an int as a reference. As long as a class follows all of the rules it can be initialized. The JVM allocates space for all static fields and assigns default values to the fields (0 for numbers, null for objects, or some other value if there is a constant value associated with the field). The final step of this process is calling the <clinit> method of the class, where the Java compiler places all the code that appears outside of the methods. Once this step has completed successfully, the class is ready to be executed. If some exception is thrown by the <clinit> method, the class is not ready and an ExceptionInInitializerException is thrown by the resolveClass method from the ClassLoader.

### ***Verification***

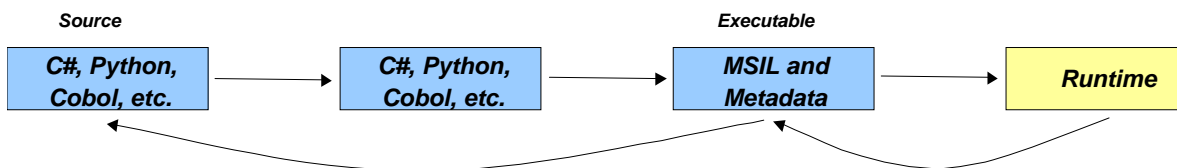
Verification is a very important process of loading classes into the JVM. It guarantees that certain parts of a computer, ie locations in memory, are not corrupted by a program. Certain locations in memory are reserved for the operating system and if they are altered in any way by something other than the operating system, the machine will most likely crash. There are different ways programs can attempt to do this. One such way is to overflow the stack which can corrupt parts of reserved memory. Another way would be to cast an object in order to obtain a

pointer to a location in memory the program is not allowed to access. Verification makes sure this does not happen by checking that objects are always used in their proper manner. This is not the only role of verification, it also makes sure that classes are structurally valid, constant references are correct, all instructions are valid, the stack and local variables contain values according to their types, as well as making sure the class does in fact exist and that the class has the necessary methods and fields. The JVM performs all of these checks by looking at the bytecodes generated before the program is executed. By doing this, the JVM ensures that if a program is either written poorly, or more importantly, written with malicious intent, it will not run on the machine. It also allows programs to be executed more quickly since it does not have to check for certain kinds of errors.

## .NET

### *The Common Language Runtime*

.NET uses the Common Language Runtime (CLR) to manage and execute code. It is the core of the .NET framework. Like the JVM, the CLR loads classes, links them and performs verification. While Java tries to manage code in the JVM with the philosophy “write once run anywhere”, the CLR allows the programmer to create a working system comprised of one and more programming languages that can work with each other. To facilitate this multi-language environment, code abides by the following paradigm:



So assume we have a source file in C# that we want to run. First we have to choose a compiler appropriate for C#. This compiler of course should target the CLR used in .NET by compiling the code into the intermediate language MSIL which will be contained in an executable file. In addition to the MSIL, the executable file will also have metadata. Metadata fully describes all “types” in the MSIL and is independent of the source language. Types includes built in “value types” (integers, characters, etc.) and “reference types” related to Objects, Interfaces and Pointers. The reason for the arrow pointing to the source code from the executable relates to source code referencing language independent metadata from another executable file (Microsoft calls them “assemblies”). Since the metadata is language independent, this facilitates code

interoperability. Finally, there is yet another feedback loop at runtime. If a program requires another MSIL, the MSIL must be run to instantiate its types and provide them to the program that is running (Watkins 2000).

To facilitate language interoperability, any language that intends to reference another must have their metadata conform to the Common Language Specifications (CLS). The CLS creates a common ground for all languages supported in the .NET framework. It is not intended to support all features for all languages, but is a subset of features that are common across most languages (Watkins 2000).

### ***Attributes and Reflection***

An attribute is a powerful feature in C# that allows the developer to create metadata for methods and classes of a program. Reflection is used to access metadata in an executable, and if a method is found, it can even invoke it. In result, a program can access an executable's metadata and act appropriately. So what does this mean? A piece of source code can actually read information in an executable in addition to invoking behaviors of the executable. So you can dynamically create an instance of a type that is imbedded in the compiled MSIL thus allowing language interoperability (Using Attributes 2007).

### **Just In Time Compiler (JIT) – Compiling to Native and Running Code**

Both the JVM and C# use a just in time compiler to compile their respective intermediate languages (IL) and execute them. After compiling code to the IL, the JIT compiles the IL to native code, which is CPU specific code that runs on the same architecture as the JIT compiler. The JIT takes into account that some of the code in the IL may never be reached. So, the first time a piece of code is reached by the JIT, it is compiled into native code and cached into memory. Additionally, a stub is attached to the reached code in order to instruct the compiler to directly go to the native code in order to avoid multiple compilations. Avoiding multiple compilations then optimizes the code significantly (Compiling MSIL 2007).

During compilation, another method may be used to convert code to native. This is called install-time code generation. This method works exactly like the JIT, but retrieves blocks of related code. This is analogous to the example of retrieving tools in your workshop. You don't go back and forth to get the tools you need, instead you grab your toolbox to save time (Compiling MSIL 2007).

## *Mobility*

---

### *Is C# and the .NET framework only available for PC architecture?*

Actually, no. Microsoft does provide their common language infrastructure (nicknamed “Roto”) to run all supported .NET languages on a collection of operating systems, including OSX and FreeBSD, for non-commercial purposes. Though the source code differs to the commercial CLR to make the language more accessible, it allows developers and academics alike to get under the hood of the .NET framework, helping a seasoned programmer better understand certain aspects of the framework such as the garbage collector or the JIT (Shared Source 2007).

There are also open source movements to bring the .NET framework to all platforms. The most popular one right now is Mono, which is provided by Novell. Mono provides a collection of comparable .NET tools including a C# compiler and a CLR. Currently Mono has a large number of JITs to support an impressive collection of processors. During Mono’s creation, Microsoft began patenting pieces of C#. This then led to patent violations in Mono, which of course led to discussions as to whether or not Microsoft would destroy the Mono project and sue its users. In the end, Microsoft decided not to sue Novell or Novell’s customers for use of Mono. Nonetheless, it seems the pardon applies to Novell customers and developers only. Despite this unfortunate circumstance, Microsoft and Novell are working together to encourage more integrated products (Mono 2007).

So to answer the BIG question: Is .NET platform independent?

If you are writing code for non-commercial use, you can use Roto for platform independence. If you are writing non-PC code for commercial purposes using Mono (or anyone else for that matter), and you are not a Novell customer or developer, then you are breaking the law.

## *Throwing out the Garbage*

---

Both Java and C# have made the life of the developer substantially easier. No more dangling pointers, or the horrendous task of managing a heap. These evolutions of C++ and C are a similar improvement to what the digital camera did to a photographer’s workflow. No longer

does a photographer have to have their creative process hindered with time spent elbow deep managing photo fluids. Likewise, a software engineer can focus on programming as opposed to being hindered by memory management.

So how do they do this?

### **Throwing out the garbage the .NET way**

Like its predecessors, C# requires that all resources be allocated from the managed heap. The main difference though, is that memory is freed automatically when developers would have to free memory manually in the C and C++ environment.

Garbage collection, which is managed by the CLR, goes through the following steps:

1. runtime reserves a contiguous region of address space
2. place an object on the heap
3. garbage collection when the heap is full

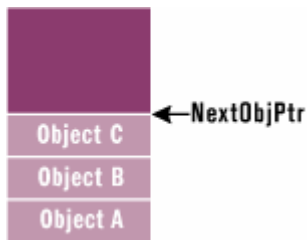
#### ***Reserving address space***

When code is run, a contiguous region of space is reserved. This is the managed heap where objects are stored and maintained. Additionally, a pointer is stored that references the next location in memory where an object can be stored. When the contiguous space is initialized, this pointer points to the base or starting point where an object can be stored in the managed heap (MS Garbage Collection, Part 1 2000).

#### ***Place an object on the heap***

During runtime, when the “new” operator is encountered, runtime determines whether or not there is available space for the new object. If space is available, the object is placed on the stack. Additionally, the pointer that is maintained in the managed heap is incremented such that the next reference is to the next block of contiguous memory. The example below depicts a visual representation of a stack after four objects (objects A, B, C) have been placed on the managed heap. So far, this method of memory management is easier and should be faster than C or C++. In C or C++, when adding an object, runtime must traverse through a linked list to find a sufficiently large block of memory to store an object or data structure. Once a block is found,

the pointers in the managed heap have to be appropriately adjusted so that the next addition will properly traverse the managed heap. For .NET, you just add an object to a location according to the heap pointer. From a high level perspective, adding an object to the heap in C# is a constant time algorithm as opposed to the linear time in C or C++. Although adding an object is efficient, things start to get hairy and a performance hit occurs when garbage collection occurs (MS Garbage Collection, Part 1 2000).



(figure from MS Garbage Collection, Part 1 2000)

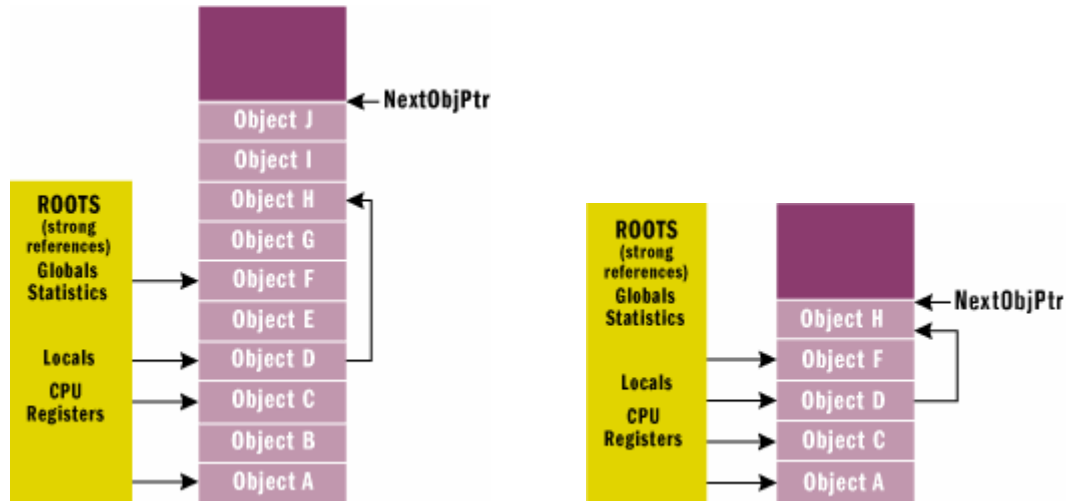
### ***Garbage Collection – How it Works***

When the heap is full, the garbage collector (GC) must be triggered to collect unused objects. If an object is no longer used, the garbage collector will remove it from the heap, which will make more memory available. As easy as this is to say, implementing this in the background is quite amazing and complicated. I will go through a high level, but thorough example of how garbage collection is performed in C# .NET and Java (MS Garbage Collection, Part 1 2000).

The .NET garbage collector first assumes that all objects on the managed heap are garbage. Then, the GC analyzes the applications “roots” to determine which objects are referenced. Application “roots” are any used resources by the application that might access an object such as a CPU register or thread. The just-in-time compiler maintains information on the roots and makes them available to the garbage collector so it can appropriately free memory (MS Garbage Collection, Part 1 2000).

Once the GC has “root” information, it creates a tree of used objects for each root. The GC traverses this tree to essentially mark all referenced objects in the heap. Additionally, an object can reference another object, possibly indicating an object that is not referenced by the system, but clearly can’t be collected. If a root references an object, this is considered a “strong reference”. If an object references another object, this is considered a “weak” reference. Any objects that don’t eventually tie into a root are considered garbage and are removed. Once objects are removed, the remaining objects are shifted to create a contiguous block of objects and

on top of that, a contiguous block of available memory. In addition, all pointers to the blocks are no longer valid and need to be updated by the GC. This collection algorithm is called the “Mark/Compact” algorithm (MS Garbage Collection, Part 1 2000).

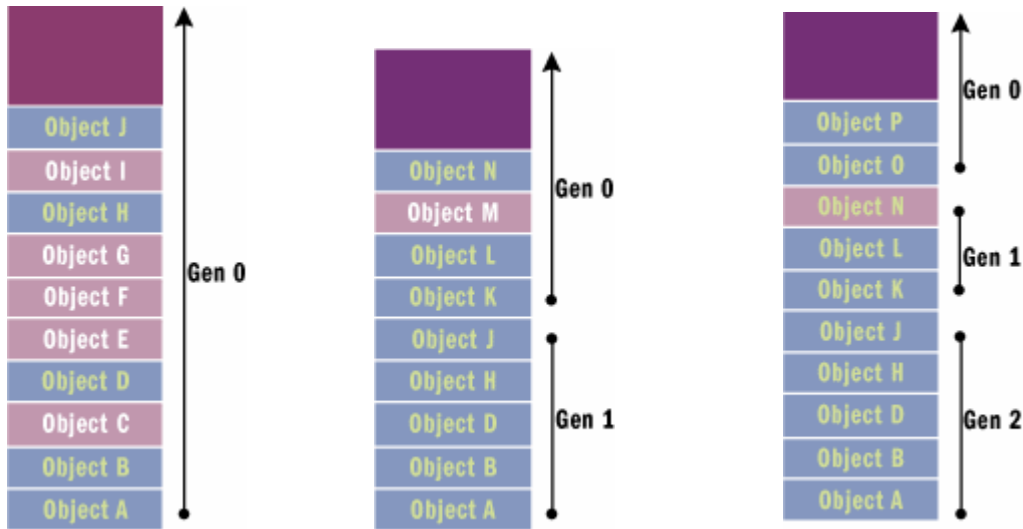


(figures from MS Garbage Collection, Part 1 2000)

### Garbage Collection – Optimization

As one can imagine, the process above could be a serious hit to performance. Every time the memory fills up, the program has to come to a dead stop to clean up the managed heap. This process includes checking roots, mapping roots, removing objects, moving objects, and cleaning up pointers. My suspicion is the time complexity of some these operations are linear or worse. This could be a serious inconvenience for a user. Can we do better?

The .NET platform tries to optimize garbage collection in a couple of ways. First, there are some common sense optimizations used to avoid doing the same work twice. For instance, when creating the tree to keep track of referenced objects, the GC makes sure it doesn't put references in more than once. If an object is referenced, then it is referenced and should not be collected. No need to keep track of it twice. Additionally, The .NET platform distributes blocks of the managed memory into “generations”. Generations are marked according to how long they have survived collection. .NET has 3 levels of generations. Generation 0 are newly created objects, generation 1 objects have survived 1 round of collection and generation 2 has survived at least 2 rounds of collection. This allows the garbage collector to optimize performance by only targeting generation 0 for collection. If collection in generation 0 is not sufficient, then it would go through generation 1 and then generation 2. The figure below illustrated these ideas:



(figures from MS Garbage Collection, Part 2 2000)

In the first figure, nothing has been garbage collected yet. These are all new objects. Eventually the heap fills up and garbage collection occurs. All highlighted objects are garbage collected and the remaining objects are put in generation 1. New objects are then stored in generation 0 in the second pass, the heap fills up, and garbage collection occurs once again. This time though, the GC only looks at generation 0 and finds one object that needs collection. Then The remaining objects in generation 0 are put into generation 1 while the objects in generation 1 are put into generation 2 (MS Garbage Collection, Part 2 2000).

### **Throwing out the garbage the JVM way**

To notice the differences between these two platforms, you really need to dive into the details. From a high level, the process is the same. A managed heap fills up, the collector is turned on, and unused objects are cleaned up to provide extra heap space. Now since anyone can create their own implementation of the JVM, they could use any garbage collection algorithm they wanted. This makes the JVM a powerful tool that enables the programmer to tailor garbage collection to their needs while the .NET framework is inflexible in this regard. The problem is the user would have to be familiar enough with whichever implementation of the JVM they are using and their code to make an educated guess on what algorithm to use. For the purpose of this paper, we will look at garbage collection used in Sun's standard implementation of the JVM.

To perform garbage collection with the JVM, the steps are essentially the same: instantiate memory, place objects onto the heap, once the heap is large enough to trigger

collection, unused objects are removed from the heap. Like .NET, sun's JVM uses generational garbage collection for optimization, but it is implemented in a very different way.

### ***The Old and the New***

Generally, the JVM creates two different heaps, one is reserved for recently instantiated objects while the second is reserved for older objects. When an object has survived collection in the newer generation a sufficient amount of times, it is transferred to the older generation heap. For garbage collection, the young heap uses a Copying Collector algorithm. The Copy Collector algorithm uses two heaps, a "from" heap and a "to" heap. The "from" heap stores new objects, once it is full, it transfers the live objects to the other heap while removing any memory space in-between the objects. Now the objects in the recently filled heap aren't deleted per say, they are just overwritten. The roles of the heaps now change. New objects are stored on the heap with the live objects and once it is filled, it will transfer its live objects to the other heap (Nagarajayya and Mayer 2002).

By default, the JVM implements the same mark and collect algorithm used by .NET on the older generation heap. However, Sun's JVM is flexible enough to allow the user to select amongst a collection of garbage collection algorithms to suit the applications needs (Nagarajayya and Mayer 2002).

Like .NET, once garbage collection occurs, the application halts until collection is complete. Generally collection occurs frequently with the younger generation. However, due to the heaps size, collection is almost instantaneous. Performance problems generally occur when the older heap triggers collection. Since the older heap is substantially larger than the younger heap, and Garbage collection time is directly related to the size of a heap, when the older heap is collected, performance slows down significantly (Nagarajayya and Mayer 2002).

### ***IDEs – What is the Difference***

---

While not a major focus for the topic, it is important to get an understanding of the difference between the development tools available for both Java and C#. For C#, the only IDE available is Visual Studio which is of course developed by Microsoft can be purchased at a hefty price tag (free express versions do exist). A very powerful feature provided by Visual Studio is its GUI builder. Any buttons, text areas, drop down menus, file menus, etc are added at the click of a button with all of the code generated in the background. Arranging the GUI is just as easy since things can be dragged with the mouse and placed in any location on the GUI window.

There should not be any compatibility issues since Visual Studio and the programs are being developed and run on top of Microsoft platforms.

For Java there are two major IDEs available, Eclipse and NetBeans. Both IDEs are available at no cost, a major plus over Visual Studio, and both are powerful in their own respects. In general, the coding environment in both of these IDEs is virtually identical to each other and Visual Studio. For a long time, developers preferred Visual Studio because of its superior GUI development environment described above. However, Eclipse and NetBeans have essentially caught up, making it more and more difficult for Microsoft to charge money for their development tools.

## *Conclusion*

---

Java and C# are structured differently, despite the fact that they are object-oriented languages and they look similar syntactically. So to choose an appropriate language depends on what you want to do. Both languages are capable of building complicated systems to do anything a computer can do, though they have advantages to each other in certain circumstances. If you need platform independence, clearly Java is the best choice. If you need to operate in a Multilanguage environment, C# would be the clear choice. Additionally, if you are building a high performance application, C# might be the best route with its memory management capabilities. However, if you are intimately familiar with the JVM, you could optimize its settings to encourage a performance boost during runtime. Or, if you are a masochist, you could write your own JVM to optimize your code. Finally, if you are writing programs extensively for the PC platform, you may want to use C#. Microsoft made the OS along with many of the standard drivers used to communicate with hardware (such as direct X). So if they built everything, you would imagine everything would be tightly integrated and easy to use.

This project is obviously not an exhaustive study of both platforms. Further research could include writing the same programs for both languages and measuring the time it takes to run the programs. For Java, the programs could be executed on different platforms and we could see how well they perform based on the machine. In .NET the program could be written in different languages supported by the platform and we could determine how well .NET handles the different languages. Then we could precisely determine which language is better suited for solving a particular problem. Other areas of interest would include web development and the

ease of creating applications for the web in either Java or C#.

## **Bibliography**

*Compiling MSIL to Native Code*. Microsoft MSDN. 1 April 2007.

<<http://msdn2.microsoft.com/en-us/library/ht8ecch6.aspx>>

Engel, Joshua. *Programming for the Java Virtual Machine*. Massachusetts: Addison Wesley, 1999.

*Extending Metadata Using Attributes*. Microsoft MSDN. 1 April 2007.

<<http://msdn2.microsoft.com/en-us/library/5x6cd29c.aspx?>>

Liberty, Jesse. *Programming C#*. Sebastopol, CA: O'Reilly Media, Inc., 2005

*Mono (software)*. 5 April 2007. Wikipedia. 1 April 2007.

<[http://en.wikipedia.org/wiki/Mono\\_%28software%29](http://en.wikipedia.org/wiki/Mono_%28software%29)>

Nagarajayya, Nagendra and J. Steven Mayer. *Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory*. July 2002. Sun Microsystems. 1 April 2007.

<<http://developers.sun.com/techtopics/mobility/midp/articles/garbage/index.html>>

Richter, Jeffrey. *Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework*. November 2000. Microsoft MSDN. 1 April 2007.

<<http://msdn.microsoft.com/msdnmag/issues/1100/GCI/>>

Richter, Jeffrey. *Garbage Collection – Part2: Automatic Memory Management in the Microsoft .NET Framework*. December 2000. Microsoft MSDN. 1 April 2007.

<<http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/default.aspx>>

*Shared Source Common Language Infrastructure*. 27 January 2007. Wikipedia. 1 April 2007.

<[http://en.wikipedia.org/wiki/Shared\\_Source\\_Common\\_Language\\_Infrastructure](http://en.wikipedia.org/wiki/Shared_Source_Common_Language_Infrastructure)>

Watkin, Damien. *Handling Language Interoperability with the Microsoft .NET Framework*.

October 2000. Microsoft MSDN. 1 April 2007. <<http://msdn2.microsoft.com/en-us/library/ms973862.aspx>>

## *Who did What*

---

### Tommy Barker

- Abstract
- Introduction
- .Net – Compilation and Execution
- .NET portion of the proposal
- Garbage Collection .NET
- Garbage Collection JVM
- General formatting and editing

### Jonathan Phillips

- Parts of Introduction
- JVM Overview
- Loading Classes in the JVM
- IDEs – What is the Difference
- Conclusion
- Java portion of the proposal
- Editing and some formatting