

A COMPARISON OF MICROSOFT'S C# PROGRAMMING LANGUAGE TO SUN MICROSYSTEMS' JAVA PROGRAMMING LANGUAGE

By

[Dare Obasanjo](#)

Introduction

The C# language is an object-oriented language that is aimed at enabling programmers to quickly build a wide range of applications for the Microsoft .NET platform. The goal of C# and the .NET platform is to shorten development time by freeing the developer from worrying about several low level plumbing issues such as memory management, type safety issues, building low level libraries, array boundschecking , etc. thus allowing developers to actually spend their time and energy working on their application and business logic instead. As a Java developer the previous sentence could be described as "a short description of the Java language and platform" if the words C# and the .NET platform were replaced with words Java and the Java platform.

What follows is an overview of similarities and differences between the language features and libraries of the C# and Java programming languages based on my experience using both languages. All code snippets below were tested on [Microsoft's .NET Framework version 2.0](#) for C# snippets and [Java™ Platform, Standard Edition version 6](#) for the Java snippets.

Quick Index

A. [The More Things Change The More They Stay The Same](#)

This section describes concepts and language features that are almost exactly the same in C# and Java.

1. [We Are All Objects](#)
2. [Keyword Jumble](#) **UPDATED**
3. [Of Virtual Machines and Language Runtimes](#)
4. [Heap Based Classes and Garbage Collection](#)
5. [Arrays Can Be Jagged](#)
6. [No Global Methods](#)
7. [Interfaces, Yes. Multiple Inheritance, No.](#)
8. [Strings Are Immutable](#)
9. [Unextendable Classes](#)
10. [Throwing and Catching Exceptions](#)
11. [Member Initialization at Definition and Static Constructors](#)
12. [Boxing](#) **NEW!**

B. [The Same But Different](#)

This section describes concepts and language features that differ either only in syntax or in some similarly minor manner between C# and Java.

1. [Main Method](#)
2. [Inheritance Syntax](#)
3. [Run Time Type Identification \(is operator\)](#)
4. [Namespaces](#)
5. [Constructors, Destructors and Finalizers](#)
6. [Synchronizing Methods and Code Blocks](#)
7. [Access Modifiers](#)
8. [Reflection](#)
9. [Declaring Constants](#)
10. [Primitive Types](#)

11. [Array Declarations](#)
12. [Calling Base Class Constructors and Constructor Chaining](#)
13. [Variable Length Parameter Lists](#) **NEW!**
14. [Generics](#) **NEW!**
15. [for-each Loop](#) **NEW!**
16. [Metadata Annotations](#) **NEW!**
17. [Enumerations](#) **NEW!**

C. [An Ever So Slight Feeling of Dèjà Vu](#)

This section describes concepts and language features that exist in C# that are similar to those that exist in Java but with a significant difference.

1. [Nested classes](#)
2. [Threads and Volatile Members](#)
3. [Operator Overloading](#)
4. [switch Statement](#)
5. [Assemblies](#)
6. [Collections](#) **UPDATED.**
7. [goto \(no longer considered harmful\)](#)
8. [Virtual Methods \(and final ones too\)](#)
9. [File I/O](#)
10. [Object Serialization](#)
11. [Documentation Generation from Source Code Comments](#) **UPDATED.**
12. [Multiple Classes in a Single File](#)
13. [Importing Libraries](#)
14. [Events](#)
15. [Cross Language Interoperability](#)

D. [Now For Something Completely Different](#)

This section describes language features and concepts that exist in C# and have no Java counterpart.

1. [Deterministic Object Cleanup](#)
2. [Delegates](#) **UPDATED.**
3. [Value Types \(Structs\)](#)
4. [Run Time Type Identification \(as operator\)](#)
5. [Properties](#) **UPDATED.**
6. [Multidimensional Arrays](#)
7. [Indexers](#)
8. [Preprocessor Directives](#) **UPDATED.**
9. [Aliases](#)
10. [Runtime Code Generation](#)
11. [Pointers and Unsafe Code](#)
12. [Pass by Reference](#)
13. [Verbatim Strings](#)
14. [Overflow Detection](#)
15. [Explicit Interface Implementation](#)
16. [Friend Assemblies](#) **NEW!**
17. [The Namespace Qualifier](#) **NEW!**
18. [Iterators \(Continuations\)](#) **NEW!**
19. [Partial Types](#) **NEW!**
20. [Static Classes](#) **NEW!**
21. [Nullable Types](#) **NEW!**
22. [Anonymous Methods](#) **NEW!**

E. [Wish You Were Here](#)

This section describes language features and concepts that exist in Java and have no C# counterpart.

1. [Checked Exceptions](#)
2. [Cross Platform Portability \(Write Once, Run Anywhere\)](#)
3. [Extensions](#)
4. [strictfp](#)
5. [Dynamic Class Loading](#)
6. [Interfaces That Contain Fields](#)
7. [Anonymous Inner Classes](#)
8. [Static Imports](#) **NEW!**

F. [Conclusion\(2001\)](#)

G. [Conclusion \(2007\)](#) **NEW!**

H. [Resources](#)

I. [Acknowledgements](#)

The More Things Change The More They Stay The Same

1. We Are All Objects

Just like Java, C# has a single rooted class hierarchy where all classes in C# are subclasses of `System.Object` the same way all Java classes are subclasses of `java.lang.Object`. The methods of the two languages' Object classes share some similarities (e.g. `System.Object`'s `ToString()` to `java.lang.Object`'s `toString()`) and differences (`System.Object` does not have analogs to `wait()`, `notify()` or `notifyAll()` in `java.lang.Object`).

NOTE: In C#, the object class can either be written as `object` or `Object`. The lower case "object" is a C# keyword which is replaced with the class name "System.Object" during compilation.

2. Keyword Jumble

There are a large number of syntactic similarities between Java and C#, similarly almost every Java keyword has a C# equivalent except for a few like `transient`, `throws` and `strictfp`. Below is a table of Java and C# keywords with the Java keywords in red while the equivalent C# keywords are in blue.

C# keyword	Java keyword	C# keyword	Java keyword	C# keyword	Java keyword	C# keyword	Java keyword
abstract	abstract	extern	native	operator	N/A	throw	throw
as	N/A	false	false	out	N/A	true	true
base	super	finally	finally	override	N/A	try	try
bool	boolean	fixed	N/A	params	...	typeof	N/A
break	break	float	float	partial	N/A	uint	N/A
byte	N/A	for	for	private	private	ulong	N/A
case	case	foreach	for	protected	N/A	unchecked	N/A
catch	catch	get	N/A	public	public	unsafe	N/A
char	char	goto	goto ¹	readonly	N/A	ushort	N/A
checked	N/A	if	if	ref	N/A	using	import
class	class	implicit	N/A	return	return	value	N/A
const	const ¹	in	N/A	sbyte	byte	virtual	N/A
continue	continue	int	int	sealed	final	void	void
decimal	N/A	interface	interface	set	N/A	volatile	volatile
default	default	internal	protected	short	short	where	extends
delegate	N/A	is	instanceof	sizeof	N/A	while	while
do	do	lock	synchronized	stackalloc	N/A	yield	N/A
double	double	long	long	static	static	:	extends
else	else	namespace	package	string	N/A	:	implements
enum	N/A	new	new	struct	N/A	N/A	strictfp
event	N/A	null	null	switch	switch	N/A	throws
explicit	N/A	object	N/A	this	this	N/A	transient ²

NOTE: Although `goto` and `const` are Java language keywords they are unused in the Java language.

NOTE: The `[NonSerialized]` attribute in C# is equivalent to the `transient` keyword in Java.

3. Of Virtual Machines and Language Runtimes

Just like Java is typically compiled to Java byte code which then runs in managed execution environment (the Java Virtual Machine or JVM) so also is C# code compiled to an Intermediate Language (IL) which then runs in the Common Language Runtime (CLR). Both platforms support native compilation via [Just In Time compilers](#).

NOTE: While the Java platform supports interpretation of byte code or byte code being JITed then run natively, the .NET platform

only supports native execution of C# code because the IL code is **always** natively compiled before running.

4. Heap Based Classes and Garbage Collection

In Java objects are created on the heap using the `new` keyword. Most classes in C# are created on the heap by using the `new` keyword. Also just as the JVM manages the destruction of objects so also does the CLR via a [Mark and Compact garbage collection algorithm](#)

NOTE: C# also supports stack-based classes, called value types, which are discussed further below.

5. Arrays Can Be Jagged

In languages like C and C++, each subarray of a multidimensional array must have the same dimensions. In Java and C# arrays do not have to be uniform because jagged arrays can be created as one-dimensional arrays of arrays. In a jagged array the contents of the array are arrays which may hold instances of a type or references to other arrays. For this reason the rows and columns in a jagged array need not have uniform length as can be seen from the following code snippet:

```
int [][]myArray = new int[2][];
myArray[0] = new int[3];
myArray[1] = new int[9];
```

The above code snippet is valid for both C# and Java.

6. No Global Methods

Just like Java and unlike C++, methods in C# have to be part of a class either as member or static methods.

7. Interfaces, Yes. Multiple Inheritance, No

C#, like Java, supports the concept of an interface which is akin to a pure abstract class. Similarly C# and Java both allow only single inheritance of classes but multiple inheritance (or implementation) of interfaces.

8. Strings Are Immutable

C# has a `System.String` class which is analogous to the `java.lang.String` class. Both classes are immutable meaning that the values of the strings cannot be changed once the strings have been created. In both instances methods that appear to modify the actual content of a string actually create a new string to return, leaving the original string unchanged. Thus the following C# and Java code does not modify the string in either case

C# Code

```
String csString = "Apple Jack";
csString.ToLower(); /* Does not modify string, instead returns lower case copy of string */
```

Java Code

```
String jString = "Grapes";
jString.toLowerCase(); /* Does not modify string, instead returns lower case copy of string */
```

To create a string-like object that allows modification in C# it is advisable to use the `System.Text.StringBuilder` class whereas in Java one would use the `java.lang.StringBuffer` class.

NOTE: In C#, the string class can either be written as `string` or `String`.

9. Unextendable Classes

Both Java and C# provide mechanisms to specify that a class should be the last one in an inheritance hierarchy and cannot be used as a base class. In Java this is done by preceding the class declaration with the `final` keyword while in C# this is done by preceding the class declaration with the `sealed` keyword. Below are examples of classes that cannot be extended in either language

C# Code

```
sealed class Student {
    string fname;
    string lname;
    int uid;
    void attendClass() {}
}
```

Java Code

```
final class Student {
    String fname;
    String lname;
```

```

    int uid;
    void attendClass() {}
}

```

10. Throwing and Catching Exceptions

Exceptions in C# and Java share a lot of similarities. Both languages support the use of the try block for indicating guarded regions, the catch block for handling thrown exceptions and the finally block for releasing resources before leaving the method. Both languages have an inheritance hierarchy where all exceptions are derived from a single Exception class. Exceptions can be caught and rethrown after some error handling occurs in both languages. Finally, both languages provide a mechanism for wrapping exceptions in one another for cases where a different exception is rethrown from the one that was caught. An example of using the exception wrapping capability is a three tier application where a `SQLException` is thrown during database access but is caught, examined, then an application specific exception is thrown. In this scenario the application specific exception can be initialized with the original `SQLException` so handlers of the application specific exception can access the original exception thrown if needed. Below are two equivalent code samples that show the similarities between exceptions in both languages.

NOTE: Although exceptions in both languages support methods for getting a stack trace, only Java exceptions have methods that allow one to alter the stack trace.

C# Code

```

using System;
using System.IO;

class MyException: Exception{

    public MyException(string message): base(message){ }

    public MyException(string message, Exception innerException):
        base(message, innerException){ }

}

public class ExceptionTest {

    static void DoStuff(){
        throw new FileNotFoundException();
    }

    public static void Main(string[] args){

        try{

            try{

                DoStuff();
                return; //won't get to execute

            }catch(IOException ioe){ /* parent of FileNotFoundException */

                throw new MyException("MyException occurred", ioe); /* rethrow new exception with inner exception s
                }

            }finally{

                Console.WriteLine("***Finally block executes even though MyException not caught***");

            }

        } //Main(string[])

    } // ExceptionTest
}

```

Java Code

```

class MyException extends Exception{

    public MyException(String message){ super(message); }

    public MyException(String message, Exception innerException){ super(message, innerException); }

}

public class ExceptionTest {

    static void doStuff(){
        throw new ArithmeticException();
    }

}

```

```

public static void main(String[] args) throws Exception{
    try{
        try{
            doStuff();
            return; //won't get to execute

        }catch(RuntimeException re){ /* parent of ArithmeticException */

            throw new MyException("MyException occurred", re); /* rethrow new exception with cause specified */
        }

    }finally{
        System.out.println("***Finally block executes even though MyException not caught***");
    }
}

} //main(string[])
} // ExceptionTest

```

11. Member Initialization at Definition and Static Constructors

Instance and static variables can be initialized at their point of definition in both C# and Java. If the member variable is an instance variable, then initialization occurs just before the constructor is called. Static members are initialized sometime before the first usage of the member and before the first creation of an instance of the class. It is also possible to specify a block of code that should run before the class is used either via creation of an instance variable or invocation of a static method. These code blocks are called static constructors in C# and static initialization blocks in Java. Static constructors are invoked before the first invocation of a static method in the class and before the first time an instance of the class is created.

C# Code

```

using System;

class StaticInitTest{

    string instMember = InitInstance();

    string staMember = InitStatic();

    StaticInitTest(){
        Console.WriteLine("In instance constructor");
    }

    static StaticInitTest(){
        Console.WriteLine("In static constructor");
    }

    static String InitInstance(){
        Console.WriteLine("Initializing instance variable");
        return "instance";
    }

    static String InitStatic(){
        Console.WriteLine("Initializing static variable");
        return "static";
    }

    static void DoStuff(){
        Console.WriteLine("Invoking static DoStuff() method");
    }

    public static void Main(string[] args){
        Console.WriteLine("Beginning main()");

        StaticInitTest.DoStuff();

        StaticInitTest sti = new StaticInitTest();

        Console.WriteLine("Completed main()");
    }
}

```

Java Code

```

class StaticInitTest{

    String instMember = initInstance();

    String staMember = initStatic();

    StaticInitTest(){
        System.out.println("In instance constructor");
    }

    static{
        System.out.println("In static constructor");
    }

    static String initInstance(){
        System.out.println("Initializing instance variable");
        return "instance";
    }

    static String initStatic(){
        System.out.println("Initializing static variable");
        return "static";
    }

    static void doStuff(){
        System.out.println("Invoking static DoStuff() method");
    }

    public static void main(String[] args){

        System.out.println("Beginning main()");

        StaticInitTest.doStuff();

        StaticInitTest sti = new StaticInitTest();

        System.out.println("Completed main()");

    }
}

```

OUTPUT FROM BOTH EXAMPLES:

```

In static constructor
Beginning main()
Invoking static DoStuff() method
Initializing instance variable
Initializing static variable
In instance constructor
Completed main()

```

12. Boxing

In situations where value types need to be treated as objects, the .NET and Java runtimes automatically converts value types to objects by wrapping them within a heap-allocated reference type in a process called *boxing*. The process of automatically convert an object to its corresponding value type such as converting an instance of `java.lang.Integer` to an `int` is known as *unboxing*. Below are examples of various situations where boxing occurs in both runtimes.

C# Code

```

using System;
using System.Collections;

//stack allocated structs also need to be boxed to be treated as objects
struct Point{

    //member fields
    private int x;
    private int y;

    public Point (int x, int y){

        this.x = x;
        this.y = y;

    }

    public override string ToString(){

```

```

        return String.Format("{0}, {1})", x, y);
    }

} //Point

class Test{

    public static void PrintString(object o){
        Console.WriteLine(o);
    }

    public static void Main(string[] args){

        Point p = new Point(10, 15);
        ArrayList list = new ArrayList();
        int z = 100;

        PrintString(p); //p boxed to object when passed to PrintString
        PrintString(z); //z boxed to object when passed to PrintString

        // integers and float boxed when stored in collection
        // therefore no need for Java-like wrapper classes
        list.Add(1);
        list.Add(13.12);
        list.Add(z);

        for(int i =0; i < list.Count; i++)
            PrintString(list[i]);

    }

}

```

Java Code

```

import java.util.*;

class Test{

    public static void PrintString(Object o){

        System.out.println(o);
    }

    public static void PrintInt(int i){

        System.out.println(i);
    }

    public static void main(String[] args){

        Vector list = new Vector();
        int z = 100;
        Integer x = new Integer(300);
        PrintString(z); //z boxed to object when passed to PrintString
        PrintInt(x); //x unboxed to int when passed to PrintInt

        // integers and float boxed when stored in collection
        // therefore no need for Java wrapper classes
        list.add(1);
        list.add(13.12);
        list.add(z);

        for(int i =0; i < list.size(); i++)
            PrintString(list.elementAt(i));

    }

}

```

The Same But Different

1. Main Method

The entry point of both C# and Java programs is a main method. There is a superficial difference in that the `Main` method in C# begins with an uppercase "M" (as do all .NET Framework method names, by convention) while the `main` method in Java begins with a lowercase "m" (as do all Java method names, by convention). The declaration for the main method is otherwise the same in both cases except for the fact that parameter to the `Main()` method in C# can have a `void` parameter.

C# Code

```
using System;

class A{

    public static void Main(String[] args){

        Console.WriteLine("Hello World");

    }

}
```

Java Code

```
class B{

    public static void main(String[] args){

        System.out.println("Hello World");

    }

}
```

It is typically recommended that one creates a main method for each class in an application to test the functionality of that class besides whatever main method actually drives the application. For instance it is possible to have two classes, A and B, which both contain main methods. In Java, since a class is the unit of compilation then all one has to do is invoke the specific class one wants run via the command line to run its main method. In C# one can get the same effect by compiling the application with the /main switch to specify which main should be used as the starting point of the application when the executable is created. Using test mains in combination with conditional compilation via [preprocessor directives](#) is a powerful testing technique.

Java Example

```
C:\CodeSample> javac A.java B.java
```

```
C:\CodeSample> java A
Hello World from class A
```

```
C:\CodeSample> java B
Hello World from class B
```

C# Example

```
C:\CodeSample> csc /main:A /out:example.exe A.cs B.cs
```

```
C:\CodeSample> example.exe
Hello World from class A
```

```
C:\CodeSample> csc /main:B /out:example.exe A.cs B.cs
```

```
C:\CodeSample> example.exe
Hello World from class B
```

So in Java's favor, one doesn't have to recompile to change which main is used by the application while a recompile is needed in a C# application. However, On the other hand, Java doesn't support conditional compilation, so the main method will be part of even your released classes.

2. Inheritance Syntax

C# uses C++ syntax for inheritance, both for class inheritance and interface implementation as opposed to the `extends` and `implements` keywords.

C# Code

```
using System;

class B:A, IComparable{

    int CompareTo(){}

    public static void Main(String[] args){

        Console.WriteLine("Hello World");

    }

}
```

Java Code

```
class B extends A implements Comparable{

    int compareTo(){}

    public static void main(String[] args){

        System.out.println("Hello World");

    }

}
```

```
}  
}
```

Since C# is aimed at transitioning C++ developers the above syntax is understandable although Java developers may pine for the Java syntax especially since it is clear from looking at the class declaration in the Java version whether the class is subclassing a class or simply implementing an interface while it isn't in the C# version without intimate knowledge of all the classes involved. Although it should be noted that in .NET naming conventions, interface names have an upper-case "I" prepended to their names (as in `IClonable`), so this isn't an issue for programs that conform to standard naming conventions.

3. Run Time Type Identification (is operator)

The C# `is` operator is completely analogous to Java's `instanceof` operator. The two code snippets below are equivalent.

C# Code

```
if(x is MyClass)  
    MyClass mc = (MyClass) x;
```

Java Code

```
if(x instanceof MyClass)  
    MyClass mc = (MyClass) x;
```

4. Namespaces

A C# namespace is a way to group a collection of classes and is used in a manner similar to Java's `package` construct. Users of C++ will notice the similarities between the C# namespace syntax and that in C++. In Java, the package names dictate the directory structure of source files in an application whereas in C# namespaces do **not** dictate the physical layout of source files in directories only their logical structure. Examples below:

C# Code

```
namespace com.carnage4life{  
  
    public class MyClass {  
  
        int x;  
  
        void doStuff(){}  
  
    }  
  
}
```

Java Code

```
package com.carnage4life;  
  
public class MyClass {  
  
    int x;  
  
    void doStuff(){}  
  
}
```

C# namespace syntax also allows one to nest namespaces in the following way

C# Code

```
using System;  
  
namespace Company{  
  
    public class MyClass { /* Company.MyClass */  
  
        int x;  
  
        void doStuff(){}  
  
    }  
  
    namespace Carnage4life{  
  
        public class MyOtherClass { /* Company.Carnage4life.MyOtherClass */  
  
            int y;  
  
            void doOtherStuff(){}  
  
        }  
  
    }  
  
}
```

```

        public static void Main(string[] args){

            Console.WriteLine("Hey, I can nest namespaces");
        }

        }// class MyOtherClass
        }// namespace Carnage4life
    }// namespace Company

```

5. Constructors, Destructors and Finalizers

The syntax and semantics for constructors in C# is identical to that in Java. C# also has the concept of destructors which use syntax similar to C++ destructor syntax but have the mostly the same semantics as Java finalizers. Although finalizers exist doing work within them is not encouraged for a number of reasons including the fact that there is no way to control the order of finalization which can lead to interesting problems if objects that hold references to each other are finalized out of order. Finalization also causes more overhead because objects with finalizers aren't removed after the garbage collection thread runs but instead are eliminated after the finalization thread runs which means they have to be maintained in the system longer than objects without finalizers. Below are equivalent examples in C# and Java.

NOTE: In C#, destructors(finalizers) automatically call the base class finalizer after executing which is **not** the case in Java.

C# Code

```

using System;

public class MyClass {

    static int num_created = 0;
    int i = 0;

    MyClass(){
        i = ++num_created;
        Console.WriteLine("Created object #" + i);
    }

    ~MyClass(){
        Console.WriteLine("Object #" + i + " is being finalized");
    }

    public static void Main(string[] args){

        for(int i=0; i < 10000; i++)
            new MyClass();

    }

}

```

Java Code

```

public class MyClass {

    static int num_created = 0;
    int i = 0;

    MyClass(){
        i = ++num_created;
        System.out.println("Created object #" + i);
    }

    public void finalize(){
        System.out.println("Object #" + i + " is being finalized");
    }

    public static void main(String[] args){

        for(int i=0; i < 10000; i++)
            new MyClass();

    }

}

```

6. Synchronizing Methods and Code Blocks

In Java it is possible to specify synchronized blocks of code that ensure that only one thread can access a particular object at a time and then create a critical section of code. C# provides the `lock` statement which is semantically identical to the synchronized statement in Java.

C# Code

```

public void WithdrawAmount(int num){

```

```

    lock(this){

    if(num < this.amount)
        this.amount -= num;

    }

}

```

Java Code

```

public void withdrawAmount(int num){

    synchronized(this){

    if(num < this.amount)
        this.amount -= num;

    }

}

```

Both C# and Java support the concept of synchronized methods. Whenever a synchronized method is called, the thread that called the method locks the object that contains the method. Thus other threads cannot call a synchronized method on the same object until the object is unlocked by the first thread when it finishes executing the synchronized method. Synchronized methods are marked in Java by using the `synchronized` keyword while in C# it is done by annotating the method with the `[MethodImpl(MethodImplOptions.Synchronized)]` attribute. Examples of synchronized methods are shown below

C# Code

```

using System;
using System.Runtime.CompilerServices;

public class BankAccount{

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void WithdrawAmount(int num){

        if(num < this.amount)
            this.amount - num;

    }

} //BankAccount

```

Java Code

```

public class BankAccount{

    public synchronized void withdrawAmount(int num){

        if(num < this.amount)
            this.amount - num;

    }

} //BankAccount

```

7. Access Modifiers

Below is a table mapping C# access modifiers to Java's. C++ fans who were disappointed when Sun changed the semantics of the `protected` keyword in Java 2 will be happy to note that the C# `protected` keyword has the same semantics as the C++ version. This means that a `protected` member can only be accessed by member methods in that class or member methods in derived classes but is inaccessible to any other classes. The `internal` modifier means that the member can be accessed from other classes in the same [assembly](#) as the class. The `internal protected` modifier means that a member can be accessed from classes that are in the same assembly or from derived classes.

C# access modifier	Java access modifier
private	private
public	public
internal	protected
protected	N/A
internal protected	N/A

NOTE: The default accessibility of a C# field or method when no access modifier is specified is `private` while in Java it is

protected (except that derived classes from outside the package cannot inherit the field).

8. Reflection

The ability to discover the methods and fields in a class as well as invoke methods in a class at runtime, typically called reflection, is a feature of both Java and C#. The primary difference between reflection in Java versus reflection in C# is that reflection in C# is done at the [assembly](#) level while reflection in Java is done at the class level. Since assemblies are typically stored in DLLs, one needs the DLL containing the targeted class to be available in C# while in Java one needs to be able to load the class file for the targeted class. The examples below which enumerate the methods in a specified class should show the difference between reflection in C# and Java.

C# Code

```
using System;
using System.Xml;
using System.Reflection;
using System.IO;

class ReflectionSample {

    public static void Main( string[] args){

        Assembly assembly=null;
        Type type=null;
        XmlDocument doc=null;

        try{
            // Load the requested assembly and get the requested type
            assembly = Assembly.LoadFrom("C:\\WINNT\\Microsoft.NET\\Framework\\v1.0.2914\\System.XML.dll");
            type = assembly.GetType("System.Xml.XmlDocument", true);

            //Unfortunately one cannot dynamically instantiate types via the Type object in C#.
            doc = Activator.CreateInstance("System.Xml", "System.Xml.XmlDocument").Unwrap() as XmlDocument;

            if(doc != null)
                Console.WriteLine(doc.GetType() + " was created at runtime");
            else
                Console.WriteLine("Could not dynamically create object at runtime");

        }catch(FileNotFoundException){
            Console.WriteLine("Could not load Assembly: system.xml.dll");
            return;
        }catch(TypeLoadException){
            Console.WriteLine("Could not load Type: System.Xml.XmlDocument from assembly: system.xml.dll");
            return;
        }catch(MissingMethodException){
            Console.WriteLine("Cannot find default constructor of " + type);
        }catch(MemberAccessException){
            Console.WriteLine("Could not create new XmlDocument instance");
        }

        // Get the methods from the type
        MethodInfo[] methods = type.GetMethods();

        //print the method signatures and parameters
        for(int i=0; i < methods.Length; i++){

            Console.WriteLine ("{0}", methods[i]);

            ParameterInfo[] parameters = methods[i].GetParameters();

            for(int j=0; j < parameters.Length; j++){
                Console.WriteLine (" Parameter: {0} {1}", parameters[j].ParameterType, parameters[j].Name);
            }

        }//for (int i...)

    }
}
```

Java Code

```
import java.lang.reflect.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;

class ReflectionTest {

    public static void main(String[] args) {

        Class c=null;
        Document d;
```

```

try{

    c = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument().getClass();
    d = (Document) c.newInstance();

    System.out.println(d + " was created at runtime from its Class object");

} catch (ParserConfigurationException pce){
    System.out.println("No document builder exists that can satisfy the requested configuration");
} catch (InstantiationException ie){
    System.out.println("Could not create new Document instance");
} catch (IllegalAccessException iae){
    System.out.println("Cannot access default constructor of " + c);
}

// Get the methods from the class
Method[] methods = c.getMethods();

//print the method signatures and parameters
for (int i = 0; i < methods.length; i++) {

    System.out.println( methods[i]);

    Class[] parameters = methods[i].getParameterTypes();

    for (int j = 0; j < parameters.length; j++) {
        System.out.println("Parameters: " + parameters[j].getName());
    }
}
}
}

```

One might notice from the above code samples that there is slightly more granularity in the C# Reflection API than the Java Reflection API as can be seen by the fact that C# has a `ParameterInfo` class which contains metadata about the parameters of a Method while Java uses `Class` objects for that which lose some information such as the name of the parameter.

Sometimes there is a need to obtain the metadata of a specific class encapsulated as an object. This object is the `java.lang.Class` object in Java and the `System.Type` object in C#. To retrieve this metadata class from an instance of the target class, the `getClass()` method is used in Java while the `GetType()` method is used in C#. If the name of the class is known at compile time then one can avoid creating an instance of the class just to obtain the metadata class by doing the following

C# Code

```
Type t = typeof(ArrayList);
```

Java Code

```
Class c = java.util.ArrayList.class; /* Must append ".class" to fullname of class */
```

9. Declaring Constants

To declare constants in Java the `final` keyword is used. Final variables can be set either at compile time or run time. In Java, when the `final` is used on a primitive it makes the value of the primitive immutable while when used on object references it makes the reference constant meaning that the reference can only point to only one object during its lifetime. Final members can be left uninitialized when declared but then must be defined in the constructor.

To declare constants in C# the `const` keyword is used for compile time constants while the `readonly` keyword is used for runtime constants. The semantics of constant primitives and object references in C# is the same as in Java.

Unlike C++, it is not possible to specify an immutable class via language constructs in either C# or Java. Neither is it possible to create a reference through which it's impossible to modify a mutable object.

C# Code

```
using System;

public class ConstantTest{

    /* Compile time constants */
    const int i1 = 10;    //implicitly a static variable

    // code below won't compile because of 'static' keyword
    // public static const int i2 = 20;

    /* run time constants */
    public static readonly uint l1 = (uint) DateTime.Now.Ticks;
}
```

```

    /* object reference as constant */
    readonly Object o = new Object();

    /* uninitialized readonly variable */
    readonly float f;

    ConstantTest() {
        // uninitialized readonly variable must be initialized in constructor
        f = 17.21f;
    }
}

```

Java Code

```

import java.util.*;

public class ConstantTest{

    /* Compile time constants */
    final int i1 = 10;    //instance variable
    static final int i2 = 20; //class variable

    /* run time constants */
    public static final long l1 = new Date().getTime();

    /* object reference as constant */
    final Vector v = new Vector();

    /* uninitialized final */
    final float f;

    ConstantTest() {
        // uninitialized final variable must be initialized in constructor
        f = 17.21f;
    }
}

```

NOTE: The Java language also supports having `final` parameters to a method. This functionality is non-existent in C#.

The primary use of final parameters is to allow arguments to a method to be accessible from within [inner classes](#) declared in the method body.

10. Primitive Types

For every Java primitive type there is a corresponding C# type which has the same name (except for `byte`). The `byte` type in Java is signed and is thus analogous to the `sbyte` type in C# and **not** the `byte` type. C# also has unsigned versions of some primitives such as `ulong`, `uint`, `ushort` and `byte`. The only significantly different primitive in C# is the `decimal` type, a type which stores decimal numbers without rounding errors (at the cost of more space and less speed).

Below are different ways to declare real valued numbers in C#.

C# Code

```

decimal dec = 100.44m; //m is the suffix used to specify decimal numbers
double dbl = 1.44e2d; //e is used to specify exponential notation while d is the suffix used for double

```

11. Array Declarations

Java has two ways in which one can declare an array, one which is backwards compatible with the notation used in C & C++ and another which is generally accepted as being clearer to read, C# uses only the latter array declaration syntax.

C# Code

```

int[] iArray = new int[100]; //valid, iArray is an object of type int[]
float fArray[] = new float[100]; //ERROR: Won't compile

```

Java Code

```

int[] iArray = new int[100]; //valid, iArray is an object of type int[]
float fArray[] = new float[100]; //valid, but isn't clear that fArray is an object of type float[]

```

12. Calling Base Class Constructors and Constructor Chaining

C# and Java automatically call base class constructors, and both provide a way to call the constructor of the base class with specific parameters. Similarly both languages enforce that the call to the base class constructor occurs before any initializations in the derived constructor which prevents the derived constructor from using members that are yet to be initialized. The C# syntax for calling the base class constructor is reminiscent of the C++ initializer list syntax.

Both languages also provide a way to call a constructor from another which allows one to reduce the amount of code duplication that can occur in constructors. This practice is typically called constructor chaining.

C# Code

```
using System;

class MyException: Exception
{
    private int Id;

    public MyException(string message): this(message, null, 100){ }

    public MyException(string message, Exception innerException):
        this(message, innerException, 100){ }

    public MyException(string message, Exception innerException, int id):
        base(message, innerException){
            this.Id = id;
        }
}
```

Java Code

```
class MyException extends Exception{
    private int Id;

    public MyException(String message){
        this(message, null, 100);
    }

    public MyException(String message, Exception innerException){
        this(message, innerException, 100);
    }

    public MyException( String message,Exception innerException, int id){
        super(message, innerException);
        this.Id = id;
    }
}
```

13. Variable Length Parameter Lists

In C and C++ it is possible to specify that a function takes a variable number of arguments. This functionality is used extensively in the printf and scanf family of functions. Both C# and Java allow one to define a parameter that indicates that a variable number of arguments are accepted by a method. In C#, the mechanism for specifying that a method accepts a variable number of arguments is by using the `params` keyword as a qualifier to the last argument to the method which should be an array. In Java, the same effect is achieved by appending the string `"..."` to the typename of the last argument to the method.

C# Code

```
using System;

class ParamsTest{

    public static void PrintInts(string title, params int[] args){
        Console.WriteLine(title + ":" );
        foreach(int num in args)
            Console.WriteLine(num);
    }

    public static void Main(string[] args){
        PrintInts("First Ten Numbers in Fibonacci Sequence", 0, 1, 1, 2, 3, 5, 8, 13, 21, 34);
    }
}
```



```

}
Java Code
class Test{

    public static void PrintInts(String title, Integer... args){

        System.out.println(title + ":");

        for(int num : args)
            System.out.println(num);

    }

    public static void main(String[] args){

        PrintInts("First Ten Numbers in Fibonacci Sequence", 0, 1, 1, 2, 3, 5, 8, 13, 21, 34);

    }

}

```

14. Generics

Both C# and Java provide a mechanism for creating strongly typed data structures without knowing the specific types at compile time. Prior to the existence of the Generics feature set, this capability was achieved by specifying the type of the objects within the data structure as `Object` then casting to specific types at runtime. This technique had several drawbacks including lack of type safety, poor performance and code bloat.

The following code sample shows how one would calculate the sum of all the integers in a collection using generics and using a collection of `Objects` so that both approaches can be compared.

```

C# Code
using System;
using System.Collections;
using System.Collections.Generic;

class Test{

    public static Stack GetStackB4Generics(){
        Stack s = new Stack();
        s.Push(2);
        s.Push(4);
        s.Push(5);

        return s;
    }

    public static Stack<int> GetStackAfterGenerics(){
        Stack<int> s = new Stack<int>();
        s.Push(12);
        s.Push(14);
        s.Push(50);

        return s;
    }

    public static void Main(String[] args){

        Stack s1 = GetStackB4Generics();
        int sum1 = 0;

        while(s1.Count != 0){
            sum1 += (int) s1.Pop(); //cast
        }

        Console.WriteLine("Sum of stack 1 is " + sum1);

        Stack<int> s2 = GetStackAfterGenerics();
        int sum2 = 0;

        while(s2.Count != 0){
            sum2 += s2.Pop(); //no cast
        }

        Console.WriteLine("Sum of stack 2 is " + sum2);

    }

}
Java Code
import java.util.*;

class Test{

    public static Stack GetStackB4Generics(){

```

```

        Stack s = new Stack();
        s.push(2);
        s.push(4);
        s.push(5);

        return s;
    }

    public static Stack<Integer> GetStackAfterGenerics(){
        Stack<Integer> s = new Stack<Integer>();
        s.push(12);
        s.push(14);
        s.push(50);

        return s;
    }

    public static void main(String[] args){

        Stack s1 = GetStackB4Generics();
        int sum1 = 0;

        while(!s1.empty()){
            sum1 += (Integer) s1.pop(); //cast
        }

        System.out.println("Sum of stack 1 is " + sum1);

        Stack<Integer> s2 = GetStackAfterGenerics();
        int sum2 = 0;

        while(!s2.empty()){
            sum2 += s2.pop(); //no cast
        }

        System.out.println("Sum of stack 2 is " + sum2);
    }
}

```

Although similar in concept to templates in C++, the Generics feature in C# and Java is not implemented similarly. In Java, the generic functionality is implemented using *type erasure*. Specifically the generic type information is present only at compile time, after which it is erased by the compiler and all the type declarations are replaced with `Object`. The compiler then automatically inserts casts in the right places. The reason for this approach is that it provides total interoperability between generic code and legacy code that doesn't support generics. The main problem with type erasure is that the generic type information is not available at run time via reflection or run time type identification. Another consequence of this approach is that generic data structures types must always be declared using objects and not primitive types. Thus one must create `Stack<Integer>` instead of `Stack<int>` when working integers.

In C#, there is explicit support for generics in the .NET runtime's instruction language (IL). When the generic type is compiled, the generated IL contains place holders for specific types. At runtime, when an initial reference is made to a generic type (e.g. `List<int>`) the system looks to see if anyone already asked for the type or not. If the type has been previously requested, then the previously generated specific type is returned. If not, the JIT compiler instantiates a new type by replacing the generic type parameters in the IL with the specific type (e.g. replacing `List<T>` with `List<int>`). It should be noted that if the requested type is a reference type as opposed to a value type then the generic type parameter is replaced with `Object`. However there is no casting done internally by the .NET runtime when accessing the type.

In certain cases, one may need create a method that can operate on data structures containing any type as opposed to those that contain a specific type (e.g. a method to print all the objects in a data structure) while still taking advantage of the benefits of strong typing in generics. The mechanism for specifying this in C# is via a feature called *generic type inferencing* while in Java this is done using *wildcard types*. The following code samples show how both approaches lead to the same result.

C# Code

```

using System;
using System.Collections;
using System.Collections.Generic;

class Test{

    //Prints the contents of any generic Stack by
    //using generic type inference
    public static void PrintStackContents<T>(Stack<T> s){
        while(s.Count != 0){
            Console.WriteLine(s.Pop());
        }
    }

    public static void Main(String[] args){

        Stack<int> s2 = new Stack<int>();
        s2.Push(4);
        s2.Push(5);
        s2.Push(6);
    }
}

```

```

        PrintStackContents(s2);

        Stack<string> s1 = new Stack<string>();
        s1.Push("One");
        s1.Push("Two");
        s1.Push("Three");

        PrintStackContents(s1);
    }
}
Java Code
import java.util.*;

class Test{

    //Prints the contents of any generic Stack by
    //specifying wildcard type
    public static void PrintStackContents(Stack<?> s){
        while(!s.empty()){
            System.out.println(s.pop());
        }
    }

    public static void main(String[] args){

        Stack <Integer> s2 = new Stack <Integer>();
        s2.push(4);
        s2.push(5);
        s2.push(6);

        PrintStackContents(s2);

        Stack<String> s1 = new Stack<String>();
        s1.push("One");
        s1.push("Two");
        s1.push("Three");

        PrintStackContents(s1);
    }
}

```

Both C# and Java provide mechanisms for specifying constraints on generic types. In C# there are three types of constraints that can be applied to generic types

1. A derivation constraint indicates to the compiler that the generic type parameter derives from a base type such an interface or a particular base class
2. A default constructor constraint indicates to the compiler that the generic type parameter exposes a public default constructor
3. A reference/value type constraint constrains the generic type parameter to be a reference or a value type.

In Java, only the derivation constraint is supported. The following code sample shows how constraints are used in practice.

C# Code

```

using System;
using System.Collections;
using System.Collections.Generic;

public class Mammal {
    public Mammal(){};

    public virtual void Speak(){};
}

public class Cat : Mammal{
    public Cat(){};

    public override void Speak(){
        Console.WriteLine("Meow");
    }
}

public class Dog : Mammal{
    public Dog(){};

    public override void Speak(){
        Console.WriteLine("Woof");
    }
}

public class MammalHelper<T> where T: Mammal /* derivation constraint */,
                                   new() /* default constructor constraint */{

    public static T CreatePet(){
        return new T();
    }
}

```

```

        public static void AnnoyNeighbors(Stack<T> pets){
            while(pets.Count != 0){
                Mammal m = pets.Pop();
                m.Speak();
            }
        }
    }

public class Test{

    public static void Main(String[] args){

        Stack<Mammal> s2 = new Stack<Mammal>();
        s2.Push(MammalHelper<Dog>.CreatePet());
        s2.Push(MammalHelper<Cat>.CreatePet());

        MammalHelper<Mammal>.AnnoyNeighbors(s2);
    }
}

Java Code
import java.util.*;

abstract class Mammal {
    public abstract void speak();
}

class Cat extends Mammal{
    public void speak(){
        System.out.println("Meow");
    }
}

class Dog extends Mammal{
    public void speak(){
        System.out.println("Woof");
    }
}

public class Test{

    //derivation constraint applied to pets parameter
    public static void AnnoyNeighbors(Stack<? extends Mammal> pets){
        while(!pets.empty()){
            Mammal m = pets.pop();
            m.speak();
        }
    }

    public static void main(String[] args){

        Stack<Mammal> s2 = new Stack<Mammal>();
        s2.push(new Dog());
        s2.push(new Cat());

        AnnoyNeighbors(s2);
    }
}

```

C# also includes the default operator which returns the default value for a type. The default value for reference types is null, and the default value for value types (such as integers, enum, and structures) is a zero whitewash (filling the structure with zeros). This operator is very useful when combined with generics. The following code sample exercises the functionality of this operator.

```

C# Code
using System;

public class Test{

    public static T GetDefaultForType(){
        return default(T); //return default value of type T
    }

    public static void Main(String[] args){

        Console.WriteLine(GetDefaultForType<int>());
        Console.WriteLine(GetDefaultForType<string>());
        Console.WriteLine(GetDefaultForType<float>());
    }
}

```

15. for-each Loop

The for-each loop is an iteration construct that is popular in a number of scripting languages (e.g. Perl, PHP, Tcl/Tk), build tools (GNU Make) and function libraries (e.g. for_each in <algorithm> in C++). The for-each loop is a less verbose way to iterate

through arrays or classes that implement the the `System.Collections.IEnumerable` interface in C# or the `java.lang.Iterable` interface in Java.

In C#, the keywords `foreach` and `in` are used when creating the for-each loop while in Java the keyword `for` and the operator `:` are used.

C# Code

```
string[] greek_alphabet = {"alpha", "beta", "gamma", "delta", "epsilon"};

foreach(string str in greek_alphabet)
    Console.WriteLine(str + " is a letter of the greek alphabet");
```

Java Code

```
String[] greek_alphabet = {"alpha", "beta", "gamma", "delta", "epsilon"};

for(String str : greek_alphabet)
    System.out.println(str + " is a letter of the greek alphabet");
```

16. Metadata Annotations

Metadata annotations provide a powerful way to extend the capabilities of a programming language and the language runtime. These annotations can be directives that request the runtime to perform certain additional tasks, provide extra information about an item or extend the abilities of a type. Metadata annotations are common in a number of programming environments including Microsoft's COM and the Linux kernel.

C# attributes provide a way to add annotations (i.e. metadata) to a module, type, method, parameter or member variable. Below are descriptions of a few attributes that are intrinsic to .NET and how they are used to extend the capabilities of the C#.

- I. `[MethodImpl(MethodImplOptions.Synchronized)]`: is used to specify that a access to a method by multiple threads is protected by a lock to prevent concurrent access to the method and is similar to the `synchronized` in Java.
- II. `[Serializable]`: is used to mark a class as serializable and is similar to a Java class implementing the `Serializable` interface.
- III. `[FlagsAttribute]`: is used to specify that an enum should support bitwise operations. This is particularly important for enumerations where the target can have multiple values.

C# Code

```
//declaration of bit field enumeration
[Flags]
enum ProgrammingLanguages{
    C      = 1,
    Lisp   = 2,
    Basic  = 4,
    All    = C | Lisp | Basic
}

aProgrammer.KnownLanguages = ProgrammingLanguages.Lisp; //set known languages = "Lisp"
aProgrammer.KnownLanguages |= ProgrammingLanguages.C; //set known languages = "Lisp C"
aProgrammer.KnownLanguages &= ~ProgrammingLanguages.Lisp; //set known languages = "C"

if((aProgrammer.KnownLanguages & ProgrammingLanguages.C) > 0){ //if programmer knows C
    //.. do something
}
```

- IV. `[WebMethod]`: is used in combination with ASP.NET to specify that a method should be available over the web as a web service automatically. Doing the same in Java involves [configuring JAXP, UDDI, and J2EE as well as have to create an Enterprise Java Bean](#) which involves at least two interfaces and one implementation class plus setting up the deployment descriptor. For more information on webservices in C#, examine the [Your First C# Web Service page on CodeProject](#).

It is possible to access the attributes of a module, class, method or field via [reflection](#). This is particularly useful for seeing if a class supports certain behavior at runtime or for extracting metadata about a class for usage by others. Developers can create their own custom attributes by subclassing the `System.Attribute` class. What follows is an example of using an attribute to provide information about the author of a class then using reflection to access that information.

C# Code

```
using System;
using System.Reflection;
[AttributeUsage(AttributeTargets.Class)]
public class AuthorInfoAttribute: System.Attribute{

    string author;
    string email;
    string version;
```

```

public AuthorInfoAttribute(string author, string email){
    this.author = author;
    this.email = email;
}

public string Version{
    get{
        return version;
    }
    set{
        version = value;
    }
}

public string Email{
    get{
        return email;
    }
}

public string Author{
    get{
        return author;
    }
}
}

[AuthorInfo("Dare Obasanjo", "kpako@yahoo.com", Version="1.0")]
class HelloWorld{
}

class AttributeTest{
    public static void Main(string[] args){
        /* Get Type object of HelloWorld class */
        Type t = typeof(HelloWorld);

        Console.WriteLine("Author Information for " + t);
        Console.WriteLine("=====");

        foreach(AuthorInfoAttribute att in t.GetCustomAttributes(typeof(AuthorInfoAttribute), false)){
            Console.WriteLine("Author: " + att.Author);
            Console.WriteLine("Email: " + att.Email);
            Console.WriteLine("Version: " + att.Version);
        }
    }
}

```

Java annotations provide a way to add annotations (i.e. metadata) to an package, type, method, parameter, member or local variable. There are only three built-in annotations provided in the Java language which are listed below.

- I. **@Override**: is used to specify that a method is intended to override a method in a base class. If the annotated method does not override a method in the base class then an error is issued during compilation.
- II. **@Deprecated**: is used to indicate that a particular method has been deprecated. If the annotated method is used then a warning is issued during compilation.
- III. **@SuppressWarnings**: is used to prevent particular warnings from being issued by the compiler. This annotation optionally takes the name of the specific warning to suppress as an argument.

As in C# it is possible to access the annotations on a module, class, method or field via [reflection](#). However a key difference between C# attributes and Java annotations is that one can create meta-annotations (i.e. annotations on annotations) in Java but can not do the same in C#. Developers can create their own custom annotations by creating an annotation type which is similar to an interface except that the keyword `@interface` is used to define it. What follows is an example of using an attribute to provide information about the author of a class then using reflection to access that information.

Java Code

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Documented //we want the annotation to show up in the Javadocs
@Retention(RetentionPolicy.RUNTIME) //we want annotation metadata to be exposed at runtime
@interface AuthorInfo{
    String author();
    String email();
    String version() default "1.0";
}

@AuthorInfo(author="Dare Obasanjo", email="kpako@yahoo.com")
class HelloWorld{

}

public class Test{

    public static void main(String[] args) throws Exception{

        /* Get Class object of HelloWorld class */
        Class c = Class.forName("HelloWorld");
        AuthorInfo a = (AuthorInfo) c.getAnnotation(AuthorInfo.class);

        System.out.println("Author Information for " + c);
        System.out.println("=====");
        System.out.println("Author: " + a.author());
        System.out.println("Email: " + a.email());
        System.out.println("Version: " + a.version());

    }
}
```

17. Enumerations

Enums are used to create and group together a list of user defined named constants. Although on the surface the enumerated types in C# and Java seem quite similar there are some significant differences in the implementation of enumerated types in both languages. In Java, enumerated types are a full fledged class which means they are typesafe and can be extended by adding methods, fields or even implementing interfaces. Whereas in C#, an enumerated type is simply syntactic sugar around an integral type (typically an `int`) meaning they cannot be extended and are not typesafe.

The following code sample highlights the differences between enums in both languages.

C# Code

```
using System;

public enum DaysOfWeek{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}

public class Test{

    public static bool isWeekDay(DaysOfWeek day){
        return !isWeekEnd(day);
    }

    public static bool isWeekEnd(DaysOfWeek day){
        return (day == DaysOfWeek.SUNDAY || day == DaysOfWeek.SATURDAY);
    }

    public static void Main(String[] args){

        DaysOfWeek sun = DaysOfWeek.SUNDAY;
        Console.WriteLine("Is " + sun + " a weekend? " + isWeekEnd(sun));
        Console.WriteLine("Is " + sun + " a week day? " + isWeekDay(sun));

        /* Example of how C# enums are not type safe */
        sun = (DaysOfWeek) 1999;
        Console.WriteLine(sun);

    }
}
```

Java Code

```
enum DaysOfWeek{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY;

    public boolean isWeekDay(){
        return !isWeekEnd();
    }

    public boolean isWeekEnd(){
        return (this == SUNDAY || this == SATURDAY);
    }
}

public class Test{

    public static void main(String[] args) throws Exception{

        DaysOfWeek sun = DaysOfWeek.SUNDAY;
        System.out.println("Is " + sun + " a weekend? " + sun.isWeekEnd());
        System.out.println("Is " + sun + " a week day? " + sun.isWeekDay());
    }
}
```

An Ever So Slight Feeling Of Dèjà Vu

1. Nested classes

In Java and C# it is possible to nest class declarations within each other. In Java there are two kinds of nested classes; non-static nested classes also known as inner classes and static nested classes. A Java inner class can be considered as a one-to-one relationship between the inner class and its enclosing class where for each instance of the enclosing class there exists a corresponding instance of the inner class that has access to the enclosing class's instance variables and contains no static methods. On the other hand a Java static nested class is a similar to nesting a class declaration within another where the nested class has access to the static members and methods of the enclosing class.

C# has the equivalent of Java's static nested classes but has nothing analogous to Java's inner classes. The following nested class declarations below are equivalent

C# Code

```
public class Car{

    private Engine engine;

    private class Engine{

        string make;

    }

}
```

Java Code

```
public class Car{

    private Engine engine;

    private static class Engine{

        String make;

    }

}
```

NOTE: In Java a nested class can be declared in any block of code including methods, this is **not** the case in C#. The ability to create nested classes in methods in Java may seem unnecessary but combined with anonymous inner classes can provide a means of creating powerful design patterns.

2. Threads and Volatile Members

A thread is a sequential flow of control within a program. A program or process can have multiple threads running concurrently all of which may share data or run independently while performing tasks. Threads are powerful in that they allow a developer to perform multiple tasks at once in a single program or process. Advantages of threads include exploiting parallelism in

multiprocessor architectures, reducing execution time by being able to perform tasks while waiting on a blocking system calls (such as printing or other I/O), and avoiding *freezing* in GUI applications.

Java threads are created by subclassing the `java.lang.Thread` class and overriding its `run()` method or by implementing the `java.lang.Runnable` interface and implementing the `run()` method. Whereas in C#, one creates a thread by creating a new `System.Threading.Thread` object and passing it a `System.Threading.ThreadStart` [delegate](#) which is initialized with the method that is to be run as a thread. Thus, in Java a method that shall run in a multithreaded context is designed up front specifically with that in mind. On the other hand, in C# any method can be passed to a `ThreadStart` object and run in a multithreaded scenario.

In Java, every class inherits the `wait()`, `notify()` and `notifyAll()` from `java.lang.Object` which are used for thread operations. The equivalent methods in C# are the `Wait()`, `Pulse()` and `PulseAll()` methods in the `System.Threading.Monitor` class.

The example below shows a scenario where worker threads are dispatched in a specific order and must be processed in the same order upon return. Due to the non-deterministic nature of threads, on some runs the threads finish working in the order they were dispatched in and in other runs they appear out of order and thus each thread must wait until its turn comes up.

C# Code

```
using System;
using System.Threading;
using System.Collections;

public class WorkerThread{

    private int idNumber;
    private static int num_threads_made = 1;
    private ThreadSample owner;

    public WorkerThread(ThreadSample owner){

        idNumber = num_threads_made;
        num_threads_made++;
        this.owner = owner;

    }/* WorkerThread() */

    //sleeps for a random amount of time to simulate working on a task

    public void PerformTask(){

        Random r = new Random((int) DateTime.Now.Ticks);
        int timeout = (int) r.Next() % 1000;

        if(timeout < 0)
            timeout *= -1;

        //Console.WriteLine(idNumber + ":A");

        try{
            Thread.Sleep(timeout);
        } catch (ThreadInterruptedException e){
            Console.WriteLine("Thread #" + idNumber + " interrupted");
        }

        //Console.WriteLine(idNumber + ":B");

        owner.workCompleted(this);

    }/* performTask() */

    public int getIDNumber() {return idNumber;}

} // WorkerThread

public class ThreadSample {

    private static Mutex m = new Mutex();
    private ArrayList threadOrderList = new ArrayList();

    private int NextInLine(){

        return (int) threadOrderList[0];
    }

    private void RemoveNextInLine(){

        threadOrderList.RemoveAt(0);

        //all threads have shown up
        if(threadOrderList.Count == 0)
```

```

        Environment.Exit(0);
    }

    public void workCompleted(WorkerThread worker){
    try{
        lock(this){

            while(worker.getIDNumber() != NextInLine()){

                try {
                    //wait for some other thread to finish working
                    Console.WriteLine ("Thread #" + worker.getIDNumber() + " is waiting for Thread #" +
                        NextInLine() + " to show up.");

                    Monitor.Wait(this, Timeout.Infinite);

                } catch (ThreadInterruptedException e) {}

            }//while

            Console.WriteLine("Thread #" + worker.getIDNumber() + " is home free");

            //remove this ID number from the list of threads yet to be seen
            RemoveNextInLine();

            //tell the other threads to resume
            Monitor.PulseAll(this);

        }

    }catch(SynchronizationLockException){Console.WriteLine("SynchronizationLockException occurred");}

    }

    public static void Main(String[] args){

        ThreadSample ts = new ThreadSample();

        /* Launch 25 threads */
        for(int i=1; i <= 25; i++){
            WorkerThread wt = new WorkerThread(ts);
            ts.threadOrderList.Add(i);
            Thread t = new Thread(new ThreadStart(wt.PerformTask));
            t.Start();
        }

        Thread.Sleep(3600000); //wait for it all to end

    }/* main(String[]) */

} //ThreadSample

```

Java Code

```

import java.util.*;

class WorkerThread extends Thread{

    private Integer idNumber;
    private static int num_threads_made = 1;
    private ThreadSample owner;

    public WorkerThread(ThreadSample owner){

        super("Thread #" + num_threads_made);

        idNumber = new Integer(num_threads_made);
        num_threads_made++;
        this.owner = owner;

        start(); //calls run and starts the thread.
    }/* WorkerThread() */

    //sleeps for a random amount of time to simulate working on a task
    public void run(){

        Random r = new Random(System.currentTimeMillis());
        int timeout = r.nextInt() % 1000;

        if(timeout < 0)
            timeout *= -1 ;
    }
}

```

```

        try{
            Thread.sleep(timeout);
        } catch (InterruptedException e){
            System.out.println("Thread #" + idNumber + " interrupted");
        }

        owner.workCompleted(this);

    }/* run() */

    public Integer getIDNumber() {return idNumber;}

} // WorkerThread

public class ThreadSample{

    private Vector threadOrderList = new Vector();

    private Integer nextInLine(){

        return (Integer) threadOrderList.firstElement();
    }

    private void removeNextInLine(){

        threadOrderList.removeElementAt(0);

        //all threads have shown up
        if(threadOrderList.isEmpty())
            System.exit(0);
    }

    public synchronized void workCompleted(WorkerThread worker){

        while(worker.getIDNumber().equals(nextInLine())==false){

            try {
                //wait for some other thread to finish working
                System.out.println (Thread.currentThread().getName() + " is waiting for Thread #" +
                    nextInLine() + " to show up.");
                wait();
            } catch (InterruptedException e) {}

        }//while

        System.out.println("Thread #" + worker.getIDNumber() + " is home free");

        //remove this ID number from the list of threads yet to be seen
        removeNextInLine();

        //tell the other threads to resume
        notifyAll();
    }

    public static void main(String[] args) throws InterruptedException{

        ThreadSample ts = new ThreadSample();

        /* Launch 25 threads */
        for(int i=1; i <= 25; i++){
            new WorkerThread(ts);
            ts.threadOrderList.add(new Integer(i));
        }

        Thread.sleep(3600000); //wait for it all to end

    }/* main(String[]) */

} //ThreadSample

```

In many situations one cannot guarantee that the order of execution of a program will be the same as that in the source code. Reasons for the unexpected ordering of program execution include compiler optimizations that reorder statements or multiprocessor systems that fail to store variables in global memory amongst others. To work around this, both C# and Java have the concept of the `volatile` keyword which is used to tell the language runtime that reordering instructions related to accessing such fields is prohibited. There are **major differences** in the semantics of `volatile` in Java and C# which are illustrated in the example below taken from [The "Double-Checked Locking is Broken" Declaration](#)

C# Code

```

/* Used to lazily instantiate a singleton class */
/* WORKS AS EXPECTED */
class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            lock(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
}

```

Java Code

```

/* Used to lazily instantiate a singleton class */
/* BROKEN UNDER CURRENT SEMANTICS FOR VOLATILE */

class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
}

```

Although the above code snippets seem identical save for the substitution of the `synchronized` keyword with the `lock` keyword, the Java version is not guaranteed to work on all JVMs. Currently the Java Memory Model does not prevent reordering of writes to volatile variables with writes to other variables so it is possible that the new object is constructed before the helper reference is made to point at the newly created object meaning that two objects are created. Also it is possible that the helper reference is made to point at a block of memory while the object is still being created meaning that a reference to an incomplete object will be returned. In C#, the semantics of `volatile` prevent such problems from occurring because reads and writes cannot be moved backward or forward across a volatile write. Also in C#, being marked as `volatile` also prevents the Just In Time compiler from placing the variable in a register and also ensures that the variable is stored in global memory on multiprocessor systems.

For more information on the problems with the Java Memory Model and Double-Checked Locking, see the [Double-checked locking: Clever, but broken](#) article on Javaworld.

3. Operator Overloading

Operator overloading allows standard operators in a language to be given new semantics when applied in the context of a particular class or type. Operator overloading can be used to simplify the syntax of certain operations especially when they are performed very often, such as string concatenation in Java or interactions with iterators and collections in the C++ Standard Template Library.

Operator overloading is a point of contention for many developers due to the fact that it provides a lot of flexibility and power which makes it prone to abuse. There is a tendency for developers to use it poorly by doing things like overloading operators in an unintuitive manner (e.g. overloading `++` and `--` to connect and disconnect from the network), overloading operators in a manner inconsistent with their typical use (e.g. overloading `[]` to return a *copy* of an object at a particular index in a collection instead of a reference to the actual object) or overloading some operators and not others (e.g. overloading `<` but not `>`).

Overloading operators tends to be most useful when the class lends itself intuitively to using that operator. Examples of situations that intuitively suggest that operator overloading would be beneficial are overloading `[]` for use with collections, overloading `+` and `*` for use with matrices, overloading mathematical operators for use with complex numbers, and overloading the `==` and `!=` operators for classes that have some means to measure equality. Below is an example that shows how operator overloading works in C#.

NOTE: Unlike C++, C# does not allow the overloading of the following operators; `new`, `()`, `||`, `&&`, `=`, or any variations of compound assignments such as `+=`, `-=`, etc. However, compound assignment operators will call overloaded operators, for instance, `+=` would call overloaded `+`.

C# Code

```

using System;

class OverloadedNumber{

    private int value;

    public OverloadedNumber(int value){

```

```

        this.value = value;
    }

    public override string ToString(){
        return value.ToString();
    }

    public static OverloadedNumber operator -(OverloadedNumber number){
        return new OverloadedNumber(-number.value);
    }

    public static OverloadedNumber operator +(OverloadedNumber number1, OverloadedNumber number2){
        return new OverloadedNumber(number1.value + number2.value);
    }

    public static OverloadedNumber operator ++(OverloadedNumber number){
        return new OverloadedNumber(number.value + 1);
    }
}

public class OperatorOverloadingTest {

    public static void Main(string[] args){

        OverloadedNumber number1 = new OverloadedNumber(12);
        OverloadedNumber number2 = new OverloadedNumber(125);

        Console.WriteLine("Increment: {0}", ++number1);
        Console.WriteLine("Addition: {0}", number1 + number2);

    }
} // OperatorOverloadingTest

```

4. switch Statment

There are two major differences between the `switch` statement in C# versus that in Java. In C#, `switch` statements support the use of string literals and do not allow fall-through unless the label contains no statements. Fall-throughs are explicitly disallowed because they are a leading cause of hard-to-find bugs in software.

C# Code

```

switch(foo){

    case "A":
        Console.WriteLine("A seen");
        break;
    case "B":
    case "C":
        Console.WriteLine("B or C seen");
        break;

    /* ERROR: Won't compile due to fall-through at case "D" */
    case "D":
        Console.WriteLine("D seen");
    case "E":
        Console.WriteLine("E seen");
        break;
}

```

5. Assemblies

C# assemblies share a lot in common with Java JAR files. An assembly is the fundamental unit of code packaging in the .NET environment. Assemblies are self contained and typically contain the intermediate code from compiling classes, metadata about the classes, and any other files needed by the packaged code to perform its task.

Since assemblies are the fundamental unit of code packaging, several actions related to interacting with types must be done at the assembly level. For instance, granting of security permissions, code deployment, and versioning are done at the assembly level. Java JAR files perform a similar task in Java with most differences being in the implementation. Assemblies are usually stored as EXEs or DLLs while JAR files are stored in the ZIP file format.

6. Collections

A number of popular programming languages contain a collections framework which typically consists of a number of data structures for holding multiple objects as well as algorithms for manipulating the objects within the aforementioned data structures.

The primary advantage of a collections framework is that it frees developers from having to write data structures and sort algorithms every time one is needed and instead frees them up to work on the actual application logic. A secondary benefit is that collections frameworks lead to consistency across projects which means the learning curve for new developers using applications that use a collections framework is less steep when compared to a situation where one was not used.

The C# collections framework consists of the classes in the `System.Collections` and the `System.Collections.Generic` namespaces. The `System.Collections` namespace contains interfaces and abstract classes that represent abstract data types such as `ICollection`, `IEnumerator`, `IDictionary`, `ICollection`, and `CollectionBase` which enable developers to manipulate data structures independently of how they are actually implemented as long as the data structures inherit from the abstract data types. The `System.Collections` namespace also contains some concrete implementations of data structures such as `ArrayList`, `Stack`, `Queue`, `HashTable` and `SortedList`. All four of the concrete data structure implementations enable one to obtain synchronized wrappers to the collection which allows for access in a thread-safe manner. The `System.Collections.Generic` namespace has generic implementations of the key data structures in the `System.Collections` namespace including generic `List<T>`, `Stack<T>`, `Queue<T>`, `Dictionary<K,T>` and `SortedList<K,T>` classes .

The Java collections framework consists of a large number of the classes and interfaces in the `java.util` package. Instead of having a separate namespace for generic collections, the collections in the `java.util` package have been retrofitted to support generics. The Java collection framework is similar to that in C# except for the fact that it can be considered a superset of the C# collection framework since it contains a number of extra features. The Java collection framework contains data structures that are missing from those in C# such as sets and linked lists. Also the Java collections framework not only has methods that enable one to access unsafe collections in a thread safe manner but contains thread-safe versions of most of the data structures as well. Finally, the Java collections framework has a number of algorithms for manipulating the elements within the data structures including algorithms that can do the following; find the largest element based on some `Comparator`, find the smallest element, find sublists within a list, reverse the contents of a list, shuffle the contents of a list, creates immutable versions of a collection, performs sorts, and binary searches.

At the current time, the Java collections framework is more sophisticated than that available in .NET via C#.

7. goto (no longer considered harmful)

Unlike Java, C# contains the `goto` statement which can be used to jump directly from a point in the code to a label. Although much derided, `gotos` can be used in certain situations to reduce code duplication while enhancing readability. A secondary usage of the `goto` statement is the ability to mimic resumable exceptions like those in Smalltalk, as long as the exception thrown does not cross method boundaries.

NOTE: In C#, one **cannot** jump into a statement block using the `goto` statement;

C# Code

```
using System;
using System.Net.Sockets;

class GotoSample{

    public static void Main(string[] args){

        int num_tries = 0;

        retry:

        try{

            num_tries++;
            Console.WriteLine("Attempting to connect to network. Number of tries =" + num_tries);

            //Attempt to connect to a network times out
            //or some other network connection error that
            //can be recovered from

            throw new SocketException();

        }catch(SocketException){

            if(num_tries < 5)
                goto retry;

        }

    }/* Main(string[]) */
} //GotoSample
```

8. Virtual Methods (and final ones too)

One of the tenets of object oriented programming is polymorphism. Polymorphism enables one to interact with members of a type hierarchy as generic types instead of dealing with specific types. The means of implementing polymorphism typically involves having methods in a base class that may be overridden by derived classes. These methods can be invoked even though the client

has a reference to a base class type which points to an object of the derived class. Such methods are bound at runtime instead of being bound during compilation and are typically called virtual methods.

In Java all methods are virtual methods while in C#, as in C++, one must explicitly state which methods one wants to be virtual since by default they are not. To mark a method as virtual in C#, one uses the `virtual` keyword. Also, implementers of a child class can decide to either explicitly override the virtual method by using the `override` keyword or explicitly choose not to by using the `new` keyword instead. By default, in C#, the behavior of methods in a derived class that have the same signature as those in the base class is as if they were declared with the `new` keyword.

It is possible to mark methods as `final` in Java which means that the method cannot be overridden by derived classes. In C# this can be done by not marking the method as `virtual`. The major difference is that in C#, the class can still define the method but the base class version is the one that will be called if the object is used via a base class reference. Java disallows the derived class from containing a method that has the same signature as the final base class method.

Below are examples that show the differences in virtual methods in both languages.

C# Code

```
using System;

public class Parent{

    public void DoStuff(string str){
        Console.WriteLine("In Parent.DoStuff: " + str);
    }

}

public class Child: Parent{

    public void DoStuff(int n){
        Console.WriteLine("In Child.DoStuff: " + n);
    }

    public void DoStuff(string str){
        Console.WriteLine("In Child.DoStuff: " + str);
    }

}

public class VirtualTest{

    public static void Main(string[] args){

        Child ch = new Child();

        ch.DoStuff(100);
        ch.DoStuff("Test");

        ((Parent) ch).DoStuff("Second Test");
    }

} //VirtualTest
```

OUTPUT:

```
In Child.DoStuff: 100
In Child.DoStuff: Test
In Parent.DoStuff: Second Test
```

Java Code

```
class Parent{

    public void DoStuff(String str){
        System.out.println("In Parent.DoStuff: " + str);
    }

}

class Child extends Parent{

    public void DoStuff(int n){
        System.out.println("In Child.DoStuff: " + n);
    }

    public void DoStuff(String str){
        System.out.println("In Child.DoStuff: " + str);
    }

}

public class VirtualTest{

    public static void main(String[] args){

        Child ch = new Child();
```

```

        ch.DoStuff(100);
        ch.DoStuff("Test");

        ((Parent) ch).DoStuff("Second Test");
    }

} //VirtualTest

```

OUTPUT:

```

In Child.DoStuff: 100
In Child.DoStuff: Test
In Child.DoStuff: Second Test

```

The C# example can be made to produce the same output as the Java example by marking the DoStuff(string) method in the Parent class as virtual and marking the DoStuff(string) method in the Child class with the override keyword.

C# Code

```

using System;

public class Parent{

    public virtual void DoStuff(string str){
        Console.WriteLine("In Parent.DoStuff: " + str);
    }

}

public class Child: Parent{

    public void DoStuff(int n){
        Console.WriteLine("In Child.DoStuff: " + n);
    }

    public override void DoStuff(string str){
        Console.WriteLine("In Child.DoStuff: " + str);
    }

}

public class VirtualTest{

    public static void Main(string[] args){

        Child ch = new Child();

        ch.DoStuff(100);
        ch.DoStuff("Test");

        ((Parent) ch).DoStuff("Second Test");
    }

} //VirtualTest

```

OUTPUT:

```

In Child.DoStuff: 100
In Child.DoStuff: Test
In Child.DoStuff: Second Test

```

The above example can be made to produce the original results by altering the signature of the DoStuff(string) method in the Child class to

```

    public new void DoStuff(string str)

```

which states that although the DoStuff method is virtual in the base class, the child class would like to treat it as a non-virtual method.

9. File I/O

Both languages support performing I/O via Stream classes. The examples below copy the contents of a file named "input.txt" to another called "output.txt".

C# Code

```

using System;
using System.IO;

public class FileIOTest {

    public static void Main(string[] args){

        FileStream inputFile = new FileStream("input.txt", FileMode.Open);

```



```

        FileStream outputFile = new FileStream("output.txt", FileMode.Open);

        StreamReader sr      = new StreamReader(inputFile);
        StreamWriter sw      = new StreamWriter(outputFile);

        String str;

        while((str = sr.ReadLine()) != null)
            sw.Write(str);

        sr.Close();
        sw.Close();
    }
} //FileIOTest

```

Java Code

```

import java.io.*;

public class FileIO{

    public static void main(String[] args) throws IOException {

        File inputFile  = new File("input.txt");
        File outputFile = new File("output.txt");

        FileReader in    = new FileReader(inputFile);
        BufferedReader br = new BufferedReader(in);

        FileWriter out    = new FileWriter(outputFile);
        BufferedWriter bw = new BufferedWriter(out);

        String str;

        while((str = br.readLine()) != null)
            bw.write(str);

        br.close();
        bw.close();
    }
} //FileIOTest

```

10. Object Serialization

Object Persistence also known as Serialization is the ability to read and write objects via a stream such as a file or network socket. Object Persistence is useful in situations where the state of an object must be retained across invocations of a program. Usually in such cases simply storing data in a flat file is insufficient yet using a Database Management System (DBMS) is overkill. Serialization is also useful as a means of transferring the representation of a class in an automatic and fairly seamless manner.

Serializable objects in C# are annotated with the `[Serializable]` attribute. The `[NonSerialized]` attribute is used to annotate members of a C# class that should not be serialized by the runtime. Such fields are usually calculated or temporary values that have no meaning when saved. C# provides two formats for serializing classes; either as XML or in a binary format, the former is more readable by humans and applications while the latter is more efficient. One can also define custom ways an object is serialized if the standard ways are insufficient by implementing the `ISerializable` interface.

In Java, serializable objects are those that implement the `Serializable` interface while the `transient` keyword is used to mark members of a Java class as ones not to be serialized. By default Java supports serializing objects to a binary format but does provide a way of overriding the standard serialization process. Objects that plan to override default serializations can implement methods with the following signatures

```

private void readObject(java.io.ObjectInputStream stream) throws IOException, ClassNotFoundException;

private void writeObject(java.io.ObjectOutputStream stream) throws IOException

```

Since the above methods are `private` there is no interface that can be implemented to indicate that a Java class supports custom serialization using `readObject` and `writeObject`. For classes that need publicly accessible methods for custom serialization there exists the `java.io.Externalizable` interface which specifies the `readExternal()` and `writeExternal()` for use in customizing how an object is read and written to a stream.

C# Code

```

using System;
using System.IO;
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;

```

```

[Serializable]
class SerializeTest{

    [NonSerialized]
    private int x;

    private int y;

    public SerializeTest(int a, int b){

        x = a;
        y = b;

    }

    public override String ToString(){

        return "{x=" + x + ", y=" + y + "}";

    }

    public static void Main(String[] args){

        SerializeTest st = new SerializeTest(66, 61);
        Console.WriteLine("Before Binary Write := " + st);

        Console.WriteLine("\n Writing SerializeTest object to disk");
        Stream output = File.Create("serialized.bin");
        BinaryFormatter bwrite = new BinaryFormatter();
        bwrite.Serialize(output, st);
        output.Close();

        Console.WriteLine("\n Reading SerializeTest object from disk\n");
        Stream input = File.OpenRead("serialized.bin");
        BinaryFormatter bread = new BinaryFormatter();
        SerializeTest fromdisk = (SerializeTest)bread.Deserialize(input);
        input.Close();

        /* x will be 0 because it won't be read from disk since non-serialized */
        Console.WriteLine("After Binary Read := " + fromdisk);

        st = new SerializeTest(19, 99);
        Console.WriteLine("\n\nBefore SOAP(XML) Serialization := " + st);

        Console.WriteLine("\n Writing SerializeTest object to disk");
        output = File.Create("serialized.xml");
        SoapFormatter swrite = new SoapFormatter();
        swrite.Serialize(output, st);
        output.Close();

        Console.WriteLine("\n Reading SerializeTest object from disk\n");
        input = File.OpenRead("serialized.xml");
        SoapFormatter sread = new SoapFormatter();
        fromdisk = (SerializeTest)sread.Deserialize(input);
        input.Close();

        /* x will be 0 because it won't be read from disk since non-serialized */
        Console.WriteLine("After SOAP(XML) Serialization := " + fromdisk);

        Console.WriteLine("\n\nPrinting XML Representation of Object");

        XmlDocument doc = new XmlDocument();
        doc.Load("serialized.xml");
        Console.WriteLine(doc.OuterXml);

    }
}

```

Java Code

```

import java.io.*;

class SerializeTest implements Serializable{

    transient int x;

    private int y;

    public SerializeTest(int a, int b){

        x = a;
        y = b;
    }
}

```

```

    }

    public String toString(){

        return "{x=" + x + ", y=" + y + "}";

    }

    public static void main(String[] args) throws Exception{

        SerializeTest st = new SerializeTest(66, 61);
        System.out.println("Before Write := " + st);

        System.out.println("\n Writing SerializeTest object to disk");
        FileOutputStream out = new FileOutputStream("serialized.txt");
        ObjectOutputStream so = new ObjectOutputStream(out);
        so.writeObject(st);
        so.flush();

        System.out.println("\n Reading SerializeTest object from disk\n");
        FileInputStream in = new FileInputStream("serialized.txt");
        ObjectInputStream si = new ObjectInputStream(in);
        SerializeTest fromdisk = (SerializeTest)si.readObject();

        /* x will be 0 because it won't be read from disk since transient */
        System.out.println("After Read := " + fromdisk);

    }

}

```

11. Documentation Generation from Source Code Comments

Both C# and Java provide a mechanism for extracting specially formatted comments from source code and placing them in an alternate document. These comments are typically API specifications and are very useful way to provide API documentation to the users of a library. The generated documentation is also useful to share the specifications for an API between designers, developers and QA.

Javadoc is the tool used to extract API documentation from source code. Javadoc generates HTML documentation from the source code comment, an example of which is the [Java™ Platform, Standard Edition API Documentation](#) which was all generated using Javadoc. Javadoc can be used to describe information at the package, class, member and method level. Descriptions of classes and member variables can be provided with the option to add references to other classes, class members and methods.

Javadoc allows one to document the following metadata about a method:

- Description of the method.
- Exceptions thrown by the method.
- Parameters the method accepts
- Return type of the method.
- Associated methods and members.
- Indication as to whether the API has been deprecated or not.
- Version of the API the method was first added.

The deprecated information is also used by the compiler which issues a warning if a call to a method marked with the deprecated tag is encountered during compilation.

Javadoc also provides the following information automatically:

- Inherited API
- List of derived classes
- List of implementing classes for interfaces
- Serialized form of the class
- Alphabetical class listing.
- Package hierarchy in a tree format.

Since Javadoc generates HTML documentation, it is valid to use HTML in Javadoc comments. There is support for linking the generated documentation with other generated documentation available over the web. Such linking is useful when one wants readers of the documentation to be able to read the API documentation from the related sources. An example of this is the [following generated documentation which contains links](#) to the Java 2 API documentation. If no such linking is specified then the [generated documentation contains no links to other API documentation](#). Below is an example of how Javadoc comments are used

Java Code

```
/**
```

```

* Calculates the square of a number.
* @param num the number to calculate.
* @return the square of the number.
* @exception NumberTooBigException this occurs if the square of the number
* is too big to be stored in an int.
*/
public static int square(int num) throws NumberTooBigException{}

```

C# uses XML as the format for the documentation. The generated documentation is an XML file that contains the metadata specified by the user with very little additional information generated automatically. All the C# XML documentation tags have an analogous Javadoc construct while the same cannot be said for the Javadoc tags having C# XML documentation analogs. For instance, the default C# XML documentation does not have analogs to Javadoc's `@author`, `@version`, or `@deprecated` tags although such metadata can be generated by reflecting on the assembly, as Microsoft's documentation build process does. One could also create custom tags that are analogous to the Javadoc tags and more but they would be ignored by standard tools used for handling C# XML documentation including Visual Studio.NET. Also of note is that C#'s XML documentation when generated does not contain metadata about the class such as listings of inherited API, derived classes or implementing interfaces. Here is an [example of an XML file generated from C# source code](#).

The primary benefit of an XML format is that the documentation specification can now be used in many different ways. XSLT stylesheets can then be used to convert the generated documentation to ASCII text, HTML, or Postscript files. Also of note is that the generated documentation can be fed to tools that use it for spec verification or other similar tasks. It should be noted that C# currently does not have a tool analogous to Javadoc for converting the XML documentation into HTML. Microsoft is in the process of developing such a tool which is currently codenamed [SandCastle](#).

Below is an example of how C# XML documentation is used.

C# Code

```

///<summary>Calculates the square of a number.</summary>
///<param name="num">The number to calculate.</param>
///<return>The square of the number. </return>
///<exception>NumberTooBigException - this occurs if the square of the number
///is too big to be stored in an int. </exception>
public static int square(int num){}

```

12. Multiple Classes in a Single File

Multiple classes can be defined in a single file in both languages with some significant differences. In Java, there can only be one class per source file that has `public` access and it must have the same name as the source file minus the file extension. C# does not have a restriction on the number of public classes that can exist in a source file and neither is there a requirement for the name of any of the classes in the file to match that of the source file.

13. Importing Libraries

Using libraries in an application is a two-step process. First the needed libraries must be referenced somewhere in the source file which is done via the `using` keyword in C# and the `import` keyword in Java. Secondly, there must be a way to tell the compiler where to find the location of the needed library. Specifying the location of libraries that will be used by a Java program is done using the `CLASSPATH` environment variable or the `-classpath` compiler option. Assembly locations are specified with the `/r` compiler switch in C#.

14. Events

Event-driven programming is a programming model where objects can register themselves to be notified of a specific occurrence or state change in another object. Event-driven programming is also referred to as the publish-subscribe model or the observer design pattern and is very popular in graphical user interface (GUI) programming. Java and C# both have mechanisms that support events but there are significant differences. The typical publish-subscribe model has a one to many relationship between an object (publisher) and its event handlers (subscribers). A subscriber is registered by invoking a method on the publisher which then adds the subscriber to an internal collection of interested objects. When the state change that a registered subscriber is interested in occurs, a method is invoked in the publisher that cycles through the collection of subscribers and invokes a callback method on each one.

There is no general mechanism for event handling in Java. Instead there are design patterns that are used by the GUI classes which developers can take their cue from. An event is typically a subclass of the `java.util.EventObject` class, which has methods that enable setting or getting of the object that was the source of the event. A subscriber in the Java model usually implements an interface that ends with the word `Listener` (e.g. `MouseListener`, `ActionListener`, `KeyListener`, etc) which should contain a callback method that would be called by the publisher on the occurrence of the event. The publisher typically has a method that begins with `add` and ends with `Listener` (e.g. `addMouseListener`, `addActionListener`, `addKeyListener`, etc) which is used to register subscribers. The publisher also has `remove` methods for unregistering the subscribers. The aforementioned components are the primary entities in an event-driven Java program.

C# uses [delegates](#) to provide an explicit mechanism for creating a publish-subscribe model. An event is typically a subclass of the `System.EventArgs` class. Like all data classes, the event class should have a constructor that allows complete initialization without calling any other methods so that you can pass `new YourEventArgs(inits)` to the subscriber delegate. The publisher has a

protected method preceded with the word "On" (e.g. OnClick, OnClose, OnInit, etc) which is invoked when a specified event occurs, this method would then invoke the delegate passing it the source and an instance of the EventArgs object. Making the method protected allows derived classes to call it directly without the need to register a delegate. The subscriber is a method that accepts the same argument and returns the same type as the event delegate. The event delegate usually returns void and accepts two parameters; an Object which should be the source of the event and the EventArgs subclass which should represent the event that occurred.

In C#, the event is used to automatically specify that a field within a subscriber is a delegate that will be used as a callback during an event-driven situation. During compilation the compiler adds overloaded versions of the += and -= operators that are analogous to the add and remove methods that are used in Java to register and unregister a subscriber.

The example below shows a class that generates 20 random numbers and fires an event whenever one of the numbers is even.

C# Code

```
using System;

class EvenNumberEvent: EventArgs{

    /* HACK: fields are typically private, but making this internal so it
    * can be accessed from other classes. In practice should use properties.
    */
    internal int number;

    public EvenNumberEvent(int number):base(){

        this.number = number;
    }

}

class Publisher{

    public delegate void EvenNumberSeenHandler(object sender, EventArgs e);

    public event EvenNumberSeenHandler EvenNumHandler;

    protected void OnEvenNumberSeen(int num){

        if(EvenNumHandler!= null)
            EvenNumHandler(this, new EvenNumberEvent(num));
    }

    //generates 20 random numbers between 1 and 20 then causes and
    //event to occur if the current number is even.
    public void RunNumbers(){

        Random r = new Random((int) DateTime.Now.Ticks);

        for(int i=0; i < 20; i++){
            int current = (int) r.Next(20);

            Console.WriteLine("Current number is:" + current);

            //check if number is even and if so initiate callback call
            if((current % 2) == 0)
                OnEvenNumberSeen(current);
        }

    }

}

}

public class EventTest{

    //callback function that will be called when even number is seen
    public static void EventHandler(object sender, EventArgs e){

        Console.WriteLine("\t\tEven Number Seen:" + ((EvenNumberEvent)e).number);
    }

    public static void Main(string[] args){

        Publisher pub = new Publisher();

        //register the callback/subscriber
        pub.EvenNumHandler += new Publisher.EvenNumberSeenHandler(EventHandler);

        pub.RunNumbers();

        //unregister the callback/subscriber
    }

}
```

```

        pub.EventNumHandler -= new Publisher.EventNumberSeenHandler(EventHandler);
    }
}

Java Code
import java.util.*;

class EvenNumberEvent extends EventObject{

    public int number;

    public EvenNumberEvent(Object source, int number){

        super(source);
        this.number = number;
    }

}

interface EvenNumberSeenListener{

    void evenNumberSeen(EvenNumberEvent ene);

}

class Publisher{

    Vector subscribers = new Vector();

    private void OnEvenNumberSeen(int num){

        for(int i=0, size = subscribers.size(); i < size; i++){
            ((EvenNumberSeenListener)subscribers.get(i)).evenNumberSeen(new EvenNumberEvent(this, num));
        }

        public void addEvenNumberEventListener(EvenNumberSeenListener ensl){

            subscribers.add(ensl);

        }

        public void removeEvenNumberEventListener(EvenNumberSeenListener ensl){

            subscribers.remove(ensl);
        }

        //generates 20 random numbers between 1 and 20 then causes and
        //event to occur if the current number is even.
        public void RunNumbers(){

            Random r = new Random(System.currentTimeMillis());

            for(int i=0; i < 20; i++){
                int current = (int) r.nextInt() % 20;

                System.out.println("Current number is:" + current);

                //check if number is even and if so initiate callback call
                if((current % 2) == 0)
                    OnEvenNumberSeen(current);

            }//for

        }

    }//Publisher

    public class EventTest implements EvenNumberSeenListener{

        //callback function that will be called when even number is seen
        public void evenNumberSeen(EvenNumberEvent e){

            System.out.println("\t\tEven Number Seen:" + ((EvenNumberEvent)e).number);

        }

        public static void main(String[] args){

            EventTest et = new EventTest();

            Publisher pub = new Publisher();

```

```

//register the callback/subscriber
pub.addEvenNumberEventListener(et);

pub.RunNumbers();

//unregister the callback/subscriber
pub.removeEvenNumberEventListener(et);

}

}

```

15. Cross Language Interoperability

Cross language interoperability is the ability to access constructs written in one programming language from another. There are a number of ways cross language interoperability works in Java. First of all, there is the Java Native Interface (JNI) which is a mechanism that allows Java programs call native methods written in C, C++ or assembly language. The C, C++ or assembly methods must be specifically written to be called from Java. Native methods can use JNI to access Java features such as calling Java language methods, instantiating and modifying Java classes, throwing and catching exceptions, performing runtime type checking, and loading Java classes dynamically. To create a JNI program one performs the following steps:

1. Create a Java program that contains the declaration of the native method(s) marked with the `native` keyword.
2. Write a main method that loads the library created in step 6 and uses the native method(s).
3. Compile the class containing the declaration of the native method(s) and the main with the `javac` compiler.
4. Use the `javah` compiler with the `-jni` compiler option to generate a header file for the native method(s).
5. Write the native method in your language of choice (currently C, C++ or assembly).
6. Compile the header file and native source file into a shared library (i.e. a `.dll` on Windows or a `.so` file on UNIX).

Java also has the ability to interact with distributed objects that use the common object request broker architecture (CORBA) via Java IDL. CORBA is a technology that allows developers to make procedure calls on objects in a location and language agnostic manner. A CORBA application usually consists of an object request broker (ORB), a client and a server. An ORB is responsible for matching a requesting client to the server that will perform the request, using an object reference to locate the target object. When the ORB examines the object reference and checks if the target object is remote or not. If the target of the call is local then the ORB performs an inter-process communication (IPC) call. On calls to remote objects the ORB marshals the arguments and routes the invocation out over the network to the remote object's ORB. The remote ORB then invokes the method locally and sends the results back to the client via the network. CORBA has a language-agnostic interface definition language (IDL) which for which languages that support CORBA have various mappings. Java IDL supports the mappings from Java objects to CORBA IDL objects. Various ORBs support CORBA language bindings for a number of languages including C, C++, Java, Python, Lisp, Perl, and Scheme.

The most seamless way to do cross language interop in Java is when the language is compiled directly to Java byte code. This means that objects in that language are available to Java programs and Java objects are available to programs written in the target language. A good example of this is the [Jython](#) scripting language which is a version of the Python programming language that is integrated with the Java platform. Below is an example of an interactive session with Jython shows how a user could create an instance of the Java random number class (found in `java.util.Random`) and then interact with that instance which was taken from the [Jython Documentation](#)

```

C:\jython>jython
Jython 2.0 on java1.2.1
Type "copyright", "credits" or "license" for more information.
>>> from java.util import Random
>>> r = Random()
>>> r.nextInt()
-790940041
>>> for i in range(5):
...     print r.nextDouble()
...
0.23347681506123852
0.8526595592189546
0.3647833839988137
0.3384865260567278
0.5514469740469587
>>>

```

There are a number of projects in various degrees of completion that are aimed at providing a similar degree of cross language interoperability within the confines of the Java Virtual Machine. A [list of languages retargetted for the Java Virtual Machine](#) is

available on the webpage of Dr. Robert Tolksdorf. Currently, Sun Microsystems (creators of the Java language and platform) seems to be uninterested in this level of cross language interoperability and has decided to leave this to independent developers and researchers.

With seamless cross language interoperability, objects can inherit implementation from other types, instantiate and invoke methods defined on other types, and otherwise interact with objects regardless of the language the types are originally implemented in. Also tools such as class browsers, debuggers, and profilers only need to understand one format (be it Java byte codes or .NET instruction language) but can support a multitude of languages as long as they target the appropriate runtime. Also error handling across languages via exceptions is possible.

C# and the .NET runtime were created with seamless cross-language interoperability as a design goal. A language targeting the .NET common language runtime (CLR) is able to interact with other languages that conform to the [common type system](#) and when compiled include certain [metadata](#). The common type system defines how types are declared, used, and managed in the .NET runtime thus creating a framework that allows for type safety and ensures that objects written in various languages can share type information. Metadata is binary information describing the [assemblies](#), types and [attributes](#) defined in the application that are stored either in a CLR portable executable (PE) or in memory if the assembly has been loaded. Languages that are currently being developed to target the .NET runtime include APL, C#, C++, COBOL, Component Pascal, Eiffel, Haskell/Mondrian, Java, Mercury, Oberon, Perl, Python, Scheme, Smalltalk, Standard ML, and Visual Basic.

Since it is very possible that certain features in one language have no analog in another, the .NET Framework provides the [Common Language Specification](#) (CLS), which describes a fundamental set of language features and defines rules for how those features are used. The CLS rules are a subset of the common type system that is aimed at ensuring cross-language interoperability by defining a set of features that are most common in programming languages. The C# compiler is a CLS compliant compiler meaning that it can be used to generate code that complies with the CLS. The C# compiler can check for CLS compliance and issues an error when a program code uses functionality that is not supported by the CLS. To get the C# compiler to check for the CLS compliance of a piece of code, mark it with the `[CLSCompliantAttribute(true)]` attribute.

Another aspect of cross language interoperability supported by C# is interaction with COM based objects. There are mechanisms that allow developers to use COM objects from C# code and vice versa.

C# objects can utilize COM objects if a wrapper class is first created that defines the functions available in the COM object as well as some additional information. The wrapper class can then be used as if it were a regular C# object while the .NET runtime handles the complexities of marshalling arguments and the like. Creating the wrapper class can be done automatically using the `tlbimp` utility. If the utility is unable to create a type library then one must be written by hand with foreknowledge of the coclasses and interfaces being defined as well as the type library-to-assembly conversion rules.

For a COM object to utilize a C# object, a typelib must be created that describes the C# object to COM aware applications. The `tlbexp` can be used to create a typelib that describes the C# object's interface in a COM-like manner. Also the `regasm` utility can be used to register an [assembly](#) so it is available to COM. When COM objects interact with the C# object, the runtime handles whatever marshalling of data that needs to occur between COM and .NET automatically.

C# programs can also call almost any function in any DLL using a combination of the `extern` keyword and the `DllImport` attribute on the method declaration. A major advantage of this is that the method being called does not have to be specifically written to be called from C#, nor is any "wrapper" necessary-so calling existing code in DLLs is relatively simple.

Now For Something Completely Different

1. Deterministic Object Cleanup

To provide total control of releasing resources used by classes, C# provides the `System.IDisposable` interface which contains the `Dispose()` method that can be called by users of the class to release resources on completion of whatever task is at hand. Classes that manage resources such as database or file handles benefit from being disposable. Being disposable provides a deterministic way to release these resources when the class is no longer in use, which is not the case with finalizers in Java or C#. It is typical to call the `SuppressFinalize` method of the GC class in the implementation of the `Dispose` method since it is likely that finalization by the runtime won't be needed since it will be provided explicitly via the `Dispose` method. C# also has some syntactic sugar via the `using` keyword that makes releasing the resources used by classes occur in a more deterministic manner via the `Dispose` method.

If a class is disposable, it is best to make usage of the `Dispose()` method idempotent (i.e. multiple calls to `Dispose()` have no ill effects) which can be done by providing a flag that is checked within the `Dispose()` method to see if the class has already been disposed or not. The example below shows a program where a class keeps a file open up until the `Dispose()` method is called which indicates that the file no longer needs to be open.

C# Code

```
using System;
using System.IO;

public class MyClass : IDisposable {

    bool disposed = false;
    FileStream f;
    StreamWriter sw;
    private String name;
```



```

private int numShowNameCalls = 0;

MyClass(string name){

    f = new FileStream("logfile.txt", FileMode.OpenOrCreate);
    sw = new StreamWriter(f);
    this.name = name;
    Console.WriteLine("Created " + name);
}

~MyClass(){

    Dispose(false);
}

    public void Dispose(){

        if(!disposed){
            Dispose(true);
        }

    }

private void Dispose(bool disposing){

    lock(this){ /* prevents multiple threads from disposing simultaneously */

        /* disposing variable is used to indicate if this method was called from a
        * Dispose() call or during finalization. Since finalization order is not
        * deterministic, the StreamWriter may be finalized before this object in
        * which case, calling Close() on it would be inappropriate so we try to
        * avoid that.
        */
        if(disposing){
            Console.WriteLine("Finalizing " + name);
            sw.Close(); /* close file since object is done with */
            GC.SuppressFinalize(this);
            disposed = true;
        }

    }

}

public string ShowName(){

    if(disposed)
        throw new ObjectDisposedException("MyClass");

    numShowNameCalls++;
    sw.Write("ShowName() Call #" + numShowNameCalls.ToString() + "\n");

    return "I am " + name;

}

public static void Main(string[] args){

    using (MyClass mc = new MyClass("A MyClass Object")){

        for(int i = 0; i < 10; i++){

            Console.WriteLine(mc.ShowName());

        } //for

    } /* runtime calls Dispose on MyClass object once "using" code block is exited, even if exception throw

    } //Main

}

```

The above idiom is practically the same as having C++ style destructors without the worry of having to deal with memory allocation woes, making it the best of both worlds. The non-deterministic nature of finalization has long been bemoaned by Java developers, it is a welcome change to see that this will not be the case when using C#.

NOTE: Calling the Dispose() method does not request that the object is garbage collected , although it does speed up collection by eliminating the need for finalization. .

2. Delegates

Delegates are a mechanism for providing callback functions. Delegates are akin to function pointers in C or functors in C++ and are useful in the same kinds of situation. One use of delegates is passing operations to a generic algorithm based on the types being used in the algorithm. The C function `qsort()` is an example of this as are a variety of the C++ functions in `<algorithm>` like `replace_if()` and `transform()`. Another use of delegates is as a means to register handlers for a particular event (i.e. the publish-subscribe model). To get the same functionality as C# delegates in Java, one can create interfaces that specify the signature of the callback method such as is done with the `Comparable` interface although this has the drawback of forcing the method to be an instance method when it most likely should be static.

To use delegates, one first declares a delegate that has the return type and accepts the same number of parameters as the methods one will want to invoke as callback functions. Secondly one needs to define a method that accepts an instance of the delegate as a parameter. Once this is done, a method that has the same signature as the delegate (i.e. accepts same parameters and returns the same type) or has covariant return types and contravariant parameter types (i.e return type is derived from the return type of the delegate and the parameter types are ancestors of the corresponding parameters) can be created and used to initialize an instance of the delegate which can then be passed to the method that accepts that delegate as a parameter. Note that the same delegate can refer to static and instance methods, even at the same time, since delegates are multicast. The example below shows the process of creating and using instance delegates.

C# Code

```
using System;

/* Mammal class hierarchy used to show return type covariance */
public class Mammal {
    public Mammal(){};

    public virtual void Speak(){};
}

public class Cat : Mammal{
    public Cat(){};

    public override void Speak(){
        Console.WriteLine("Meow");
    }
}

public class Dog : Mammal{
    public Dog(){};

    public override void Speak(){
        Console.WriteLine("Woof");
    }
}

public class Test {

    // delegate declaration, similar to a function pointer declaration
    public delegate Mammal CallbackFunction(Dog d);

    public static Cat BarkAndScareCat(Dog d)
    {
        d.Speak();
        Cat c = new Cat();
        c.Speak();
        return c;
    }

    public static Mammal BarkAndReturnHome(Dog d)
    {
        d.Speak();
        return d;
    }

    public static void Main(string[] args){

        Dog dog          = new Dog();

        //create delegate using delegate object (old way)
        CallbackFunction myCallback = new CallbackFunction(BarkAndReturnHome);
        myCallback(dog);

        //create delegate using delegate inference (new way)
        CallbackFunction myCallback2 = BarkAndScareCat;
        myCallback2(dog);

    }
}
```

A delegate can be passed as a parameter to a method in the same way that a function pointer is passed in languages like C and C++. The following code sample shows how this is done.

C# Code

```
using System;
```

```
//delegate base
public class HasDelegates
{
    // delegate declaration, similar to a function pointer declaration
    public delegate bool CallbackFunction(string a, int b);

    //method that uses the delegate
    public bool execCallback(CallbackFunction doCallback, string x, int y)
    {
        Console.WriteLine("Executing Callback function...");
        return doCallback(x, y);
    }
}

public class FunctionDelegates
{
    public static readonly HasDelegates.CallbackFunction BarFuncCallback =
        new HasDelegates.CallbackFunction(FunctionBar);

    public static bool FunctionBar(string a, int b)
    {
        Console.WriteLine("Bar: {0} {1}", b, a);
        return true;
    }
}

public class DelegateTest {

    public static void Main(string[] args){

        HasDelegates MyDel = new HasDelegates();

        // with static delegate, no need to know how to create delegate
        MyDel.execCallback(FunctionDelegates.BarFuncCallback, "Thirty Three", 33);

    }
} // DelegateTest
```

3. Value Types (Structs)

In both Java and C#, items on the heap have to be garbage collected to reclaim the memory allocated to them when they are no longer in use while stack based objects are automatically reclaimed by the system. Memory on the stack is typically faster to allocate than heap based memory.

In Java, all classes are created on the heap while primitives are created on the stack. This can lead to situations where objects that are used similarly to primitives in a program tend to hang around and wait for garbage collection thus adding overhead to the program. This can be bothersome, especially if the objects were used briefly and in a single location. To avoid the problem of allocating heap space for such classes and then having to garbage collect them, C# has a mechanism that allows one to specify that objects of a certain class should be stack based (In fact, C#'s built-in types such as int are actually implemented as structs in the runtime library). Unlike classes, value types are always passed by value and are not garbage collected. And arrays of value types contain the actual value type objects, not references to dynamically-allocated objects-a savings of both memory and time.

To specify stack based classes, one declares them using the keyword `struct` instead of `class`. To create C# structs (also known as value types) one uses the `new` keyword to create it as is done with classes. If the struct is instantiated with the default constructor syntax then a struct with its fields zeroed out is created. However, it is not possible to define a default constructor for a struct and override this behavior.

C# Code

```
using System;

struct Point {

    public int x;
    public int y;

    public Point( int x, int y){

        this.x = x;
        this.y = y;

    }

    public override string ToString(){

        return String.Format("({0}, {1})", x, y);

    }

}
```

```

    }

    public static void Main(string[] args){

        Point start = new Point(5, 9);
        Console.WriteLine("Start: " + start);

        /* The line below wouldn't compile if Point was a class */
        Point end = new Point();
        Console.WriteLine("End: " + end);

    }

} // Point

```

4. Run Time Type Identification (as operator)

The C# `as` operator is completely analogous to C++'s `dynamic_cast` construct. The semantics of the `as` operator are that it attempts to perform an explicit cast to a type and returns `null` if the operation was unsuccessful.

C# Code

```

MyClass mc = o as MyClass;

if(mc != null)           //check if cast successful
    mc.doStuff();

```

NOTE: The `as` operator cannot be used to convert to or from value types.

5. Properties

Properties are a way to abstract away from directly accessing the members of a class, similar to how accessors (getters) and modifiers (setters) are used in the Java world. A property is accessed by users of a class as if it was a field or member variable but in actuality is a method call. Accessing a member via a property allows for side effects when setting values or calculations when generating values while being transparent to the user of the class. Properties thus provide an explicit way to decouple the implementation of the member access from how it is actually used.

It is possible to create, read-only, write-only or read-write properties depending on if the getter and setter are implemented or not. In addition, it is possible to create a property whose getter and setter have different visibility (e.g. a public getter but a private setter). The example below shows different kinds of properties.

C# Code

```

using System;

public class User {

    public User(string name){

        this.name = name;

    }

    private string name;

    //property with public getter and private setter
    public string Name{

        get{

            return name;

        }

        private set {

            name = value;

        }

    }

    private static int minimum_age = 13;

    //read-write property for class member, minimum_age
    public static int MinimumAge{

        get{

            return minimum_age;

        }

        set{

            if(value > 0 && value < 100)
                minimum_age = value;

        }

    }

}

```

```

        else
            Console.WriteLine("{0} is an invalid age, so minimum age remains at {1}", value, minimum_age);
    }
}

public static void Main(string[] args){

    User newuser = new User("Bob Hope");

    User.MinimumAge = -5; /* prints error to screen since value invalid */
    User.MinimumAge = 18;
    //newuser.Name = "Kevin Nash"; Causes compiler error since Name property is read-only

    Console.WriteLine("Minimum Age: " + User.MinimumAge);
    Console.WriteLine("Name: {0}", newuser.Name);
}
} // User

```

The use of properties as an abstraction away from whether a member access is a method call or not may fall apart if the property throws an exception. When this occurs users of the object must then treat operations that look like member accesses as if they were method calls which can lead to unintuitive looking code. The example below shows code that sets the values for the fields of a Clock class that where the setters throw an exception if the value is invalid.

C# Code

```

try{

myClock.Hours    = 28; /* setter throws exception because 28 is an invalid hour value */
myClock.Minutes  = 15;
myClock.Seconds  = 39;

}catch(InvalidTimeValueException itve){

/* figure out which field was invalid and report error */

}

```

To avoid situations like the one in the above example it is best that one avoids throwing exceptions in properties and handle the exceptions that are thrown by methods used in the property. If throwing an exception is avoidable in some situations then the documentation for the property must adequately describe the exceptions that can be thrown and the circumstances that lead to them being thrown.

6. Multidimensional Arrays

C# makes the distinction between *multidimensional* and *jagged* arrays. A multidimensional array is akin to a multidimensional array in C or C++ that is a contiguous block containing members of the same type. A jagged array is akin to an array in Java which is an array of arrays, meaning that it contains references to other arrays which may contain members of the same type or other arrays depending on how many levels the array has. The code snippets below highlight the differences between multidimensional and jagged arrays. The lack of true multidimensional arrays in Java has made it problematic to use Java in certain aspects of technical computing which has lead to various efforts to improve this position including [research efforts by IBM](#) which involved writing their own Array class to get around the shortcomings in Java arrays.

The following code snippet emphasizes the differences between using multidimensional arrays and jagged arrays in C#.

C# Code

```

using System;

public class ArrayTest {

    public static void Main(string[] args){

        int[,] multi = { {0, 1}, {2, 3}, {4, 5}, {6, 7} };

        for(int i=0, size = multi.GetLength(0); i < size; i++){

            for(int j=0, size2 = multi.GetLength(1); j < size2; j++){
                Console.WriteLine("multi[" + i + "," + j + "] = " + multi[i,j]);
            }
        }

        int[][] jagged = new int[4][];
        jagged[0] = new int[2]{0, 1};
        jagged[1] = new int[2]{2, 3};
        jagged[2] = new int[2]{4, 5};
    }
}

```

```

jagged[3] = new int[2]{6, 7};

for(int i=0, size = jagged.Length; i < size; i++){
    for(int j=0, size2 = jagged[1].Length; j < size2; j++){
        Console.WriteLine("jagged[" + i + "][" + j + "] = " + jagged[i][j]);
    }
}
}
} // ArrayTest

```

7. Indexers

An indexer is a special syntax for overloading the `[]` operator for a class. An indexer is useful when a class is a container for another kind of object. Indexers are flexible in that they support any type, such as integers or strings, as indexes. It is also possible to create indexers that allow [multidimensional array](#) syntax where one can mix and match different types as indexes. Finally, indexers can be overloaded.

C# Code

```

using System;
using System.Collections;

public class IndexerTest: IEnumerable, IEnumerator {

    private Hashtable list;

    public IndexerTest (){

        index = -1;
        list = new Hashtable();
    }

    //indexer that indexes by number
    public object this[int column]{

        get{

            return list[column];

        }

        set{

            list[column] = value;

        }

    }

    /* indexer that indexes by name */
    public object this[string name]{

        get{

            return this[ConvertToInt(name)];

        }

        set{

            this[ConvertToInt(name)] = value;

        }

    }

    /* Convert strings to integer equivalents */
    private int ConvertToInt(string value){

        string loVal = value.ToLower();

        switch(loVal){

            case "zero": return 0;
            case "one": return 1;
            case "two": return 2;
            case "three": return 3;
            case "four": return 4;
            case "five": return 5;

```

```

        default:
            return 0;
        }

        return 0;
    }

    /**
     * Needed to implement IEnumerable interface.
     */
    public IEnumerator GetEnumerator(){ return (IEnumerator) this; }

    /**
     * Needed for IEnumerator.
     */
    private int index;

    /**
     * Needed for IEnumerator.
     */
    public bool MoveNext(){
        index++;
        if(index >= list.Count)
            return false;
        else
            return true;
    }

    /**
     * Needed for IEnumerator.
     */
    public void Reset(){
        index = -1;
    }

    /**
     * Needed for IEnumerator.
     */
    public object Current{
        get{
            return list[index];
        }
    }

    public static void Main(string[] args){

        IndexerTest it = new IndexerTest();
        it[0] = "A";
        it[1] = "B";
        it[2] = "C";
        it[3] = "D";
        it[4] = "E";

        Console.WriteLine("Integer Indexing: it[0] = " + it[0]);
        Console.WriteLine("String Indexing: it[\"Three\"] = " + it[\"Three\"]);

        Console.WriteLine("Printing entire contents of object via enumerating through indexer :");

        foreach( string str in it){
            Console.WriteLine(str);
        }

    }

} // IndexerTest

```

8. Preprocessor Directives

C# includes a preprocessor that has a limited subset of the functionality of the C/C++ preprocessor. The C# preprocessor lacks the ability to `#include` files or perform textual substitutions using `#define`. The primary functionality that remains is the ability to `#define` and `#undef` identifiers and also the ability to select which sections of code to compile based on the validity of certain expressions via `#if`, `#elif`, and `#else`. The `#error` and `#warning` directives cause the errors or warnings messages following these directives to be printed on compilation. The `#pragma` directive is used to suppress compiler warning messages. Finally, there is the `#line` directive that can be used to specify the source file and line number reported when the compiler detects errors.

[C# Code](#)

```

#define DEBUG /* #define must be first token in file */
using System;

#pragma warning disable 169 /* Disable 'field never used' warning */

class PreprocessorTest{

    int unused_field;

    public static void Main(string[] args){

        #if DEBUG
            Console.WriteLine("DEBUG Mode := On");
        #else
            Console.WriteLine("DEBUG Mode := Off");
        #endif

    }

}

```

9. Aliases

The `using` keyword can be used to alias the fully qualified name for a type similar to the way `typedef` is used in C and C++. This is useful in creating readable code where the fully qualified name of a class is needed to resolve namespace conflicts.

C# Code

```

using Terminal = System.Console;

class Test{

    public static void Main(string[] args){

        Terminal.WriteLine("Terminal.WriteLine is equivalent to System.Console.WriteLine");

    }

}

```

10. Runtime Code Generation

The `Reflection.Emit` namespace contains classes that can be used to generate .NET instruction language (IL) and use it to build classes in memory at runtime or even write portable executable (PE) files to disk. This is analogous to a Java library that would allow one to create Java classes at runtime by generating Java byte codes which could then be written to disks or loaded and used within the program.

It is expected that the primary users of the `Reflection.Emit` namespace will be authors of compilers and script engines. For instance, the regular expression classes in `System.Text.RegularExpressions` use the `Reflection.Emit` library to generate a custom matching engine for each regular expression compiled.

11. Pointers and Unsafe Code

Although core C# is like Java in that there is no access to a pointer type that is analogous to pointer types in C and C++, it is possible to have pointer types if the C# code is executing in an `unsafe` context. When C# code is executing in an `unsafe` context, a lot of runtime checking is disabled which means that the program must have full trust on the machine it is running on.

Certain situations call for the use of unsafe code such as when interfacing with the underlying operating system, during interactions with COM objects that take structures that contain pointers, when accessing a memory-mapped device or in situations where performance is critical. The syntax and semantics for writing unsafe code is similar to the syntax and semantics for using pointers in C and C++. To write unsafe code, the `unsafe` keyword must be used to specify the code block as unsafe and the program must be compiled with the `/unsafe` compiler switch.

Since garbage collection may relocate managed (i.e. safe) variables during the execution of a program, the `fixed` keyword is provided so that the address of a managed variable is pinned during the execution of the parts of the program within the `fixed` block. Without the `fixed` keyword there would be little purpose in being able to assign a pointer to the address of a managed variable since the runtime may move the variable from that address as part of the mark & compact garbage collection process.

C# Code

```

using System;

class UnsafeTest{

```



```

public static unsafe void Swap(int* a, int*b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

public static unsafe void Sort(int* array, int size){
    for(int i= 0; i < size - 1; i++)
        for(int j = i + 1; j < size; j++)
            if(array[i] > array[j])
                Swap(&array[i], &array[j]);
}

public static unsafe void Main(string[] args){
    int[] array = {9, 1, 3, 6, 11, 99, 37, 17, 0, 12};

    Console.WriteLine("Unsorted Array:");

    foreach(int x in array)
        Console.Write(x + " ");

    fixed( int* iptr = array ){ // must use fixed to get address of array

        Sort(iptr, array.Length);

    }//fixed

    Console.WriteLine("\nSorted Array:");

    foreach(int x in array)
        Console.Write(x + " ");

}
}

```

12. Pass by Reference

In Java the arguments to a method are passed by value meaning that a method operates on copies of the items passed to it instead of on the actual items. In C#, as in C++ and in a sense C, it is possible to specify that the arguments to a method actually be references to the items being passed to the method instead of copies. This feature is particularly useful when one wants to create a method that returns more than one object. In Java trying to return multiple values from a method is not supported and leads to interesting anomalies like the fact that a method that swaps two numbers which has been the hallmark of freshman computer science classes for years is impossible to do in Java without resorting to coding tricks.

The C# keywords used to specify that a parameter is being passed by reference are `ref` and `out`. The difference between the keywords is that parameters passed using `ref` must be initialized to some value while those passed using `out` do not have to be.

A more detailed explanation of Java's lack of Pass By Reference semantics is available in the Javaworld article, [Does Java pass by reference or pass by value? Why can't you swap in Java?](#) by Tony Sintes

Java Code

```

class PassByRefTest{

    public static void changeMe(String s){
        s = "Changed";
    }

    public static void swap(int x, int y){
        int z = x;
        x = y;
        y = z;
    }

    public static void main(String[] args){

        int a = 5, b = 10;
        String s = "Unchanged";

        swap(a, b);
        changeMe(s);

        System.out.println("a := " + a + ", b := " + b + ", s = " + s);
    }
}

```

```
}
```

OUTPUT

a := 5, b := 10, s = Unchanged

C# Code

```
using System;

class PassByRefTest{

    public static void ChangeMe(out string s){

        s = "Changed";

    }

    public static void Swap(ref int x, ref int y){

        int z = x;
        x = y;
        y = z;

    }

    public static void Main(string[] args){

        int a = 5, b = 10;
        string s;

        Swap(ref a, ref b);
        ChangeMe(out s);

        Console.WriteLine("a := " + a + ", b := " + b + ", s = " + s);
    }
}
```

OUTPUT

a := 10, b := 5, s = Changed

13. Verbatim Strings

C# provides the ability to avoid the usage of escape sequences within string constants and instead declare strings literally. Thus backslashes, tabs, quotes and newlines can be part of a string without using escape sequences. The only caveat is that double quotes that appear within verbatim strings should be doubled. Verbatim strings are specified by prepending the @ symbol to string declarations.

C# Code

```
using System;

class VerbatimTest{

    public static void Main(){

        //verbatim string
        string filename = @"C:\My Documents\My Files\File.html";

        Console.WriteLine("Filename 1: " + filename);

        //regular string
        string filename2 = "C:\\My Documents\\My Files\\File.html";

        Console.WriteLine("Filename 2: " + filename2);

        string snl_celebrity_jeopardy_skit = @"
        Darrell Hammond (Sean Connery) : I'll take "Swords" for $400
        Will Farrell      (Alex Trebek)  : That's S-Words, Mr Connery.
        ";

        Console.WriteLine(snl_celebrity_jeopardy_skit);

    }
}
```

14. Overflow Detection

C# provides the option to explicitly detect or ignore overflow conditions in expressions and type conversions. Overflow conditions detected in code throw a `System.OverflowException`. Since overflow detection causes a performance hit it can be explicitly enabled by using the `/checked+` compiler option. One can also mandate code that must always be checked for overflow conditions by placing it in a checked block or that must always be ignored with regards to overflow detection by placing it in an unchecked block.

C# Code

```
using System;

class CheckedTest{

    public static void Main(){

        int num = 5000;

        /* OVERFLOW I */
        byte a = (byte) num; /* overflow detected only if /checked compiler option on */

        /* OVERFLOW II */
        checked{

            byte b = (byte) num; /* overflow ALWAYS detected */

        }

        /* OVERFLOW III */
        unchecked{

            byte c = (byte) num; /* overflow NEVER detected */

        }

    } //Main
}
```

15. Explicit Interface Implementation

Sometimes when a class implements an interface it is possible for there to be namespace collisions with regards to method names. For instance, a FileRepresentation class which has a graphical user interface may implement an IWindow and an IFileHandler interface. Both of these classes may have a Close method where in the case of IWindow it means to close the GUI window while in the case of IFileHandler it means to close the file. In Java, there is no solution to this problem besides writing one Close method that does the same thing when invoked via an IWindow handle or an IFileHandler handle. C# gets around this problem by allowing one to bind method implementations to specific interfaces. Thus in the aforementioned example, the FileRepresentation class would have different Close methods that would be invoked depending on whether the class was being treated as an IWindow or an IFileHandler.

NOTE: Explicit interface methods are private and can only be accessed if the object is cast to the required type before invoking the method.

C# Code

```
using System;

interface IVehicle{

    //identify vehicle by model, make, year
    void IdentifySelf();

}

interface IRobot{

    //identify robot by name
    void IdentifySelf();

}

class TransformingRobot : IRobot, IVehicle{

    string model;
    string make;
    short year;
    string name;

    TransformingRobot(String name, String model, String make, short year){

        this.name = name;
        this.model = model;
        this.make = make;
        this.year = year;

    }

    void IRobot.IdentifySelf(){

        Console.WriteLine("My name is " + this.name);

    }

    void IVehicle.IdentifySelf(){
```

```

        Console.WriteLine("Model:" + this.model + " Make:" + this.make + " Year:" + this.year);
    }

    public static void Main(){
        TransformingRobot tr = new TransformingRobot("SedanBot", "Toyota", "Corolla", 2001);

        // tr.IdentifySelf(); ERROR

        IVehicle v = (IVehicle) tr;

        IRobot r    = (IRobot) tr;

        v.IdentifySelf();

        r.IdentifySelf();

    }
}

```

OUTPUT

```

Model:Toyota Make:Corolla Year:2001
My name is SedanBot

```

16. Friend Assemblies

The friend assembly feature allows an internal type or internal member of an assembly to be accessed from another assembly. To give one assembly access to another assembly's internal types and members, the `[InternalsVisibleToAttribute]` attribute is used.

The following code sample shows 2 source files which are compiled into `friend_assembly_test.dll` and `friend_assembly_test_2.exe` which utilize the friend's assembly feature

C# Code

```

// friend_assembly_test.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
using System;

[assembly: InternalsVisibleTo("friend_assembly_test_2")]

// internal by default
class Friend
{
    public void Hello()
    {
        Console.WriteLine("Hello World!");
    }
}

// public type with internal member
public class Friend2
{
    internal string secret = "I like jelly doughnuts";
}

```

C# Code 2

```

// friend_assembly_test_2.cs
// compile with: /reference:friend_assembly_test.dll /out:friend_assembly_test_2.exe
using System;

public class FriendFinder
{
    static void Main()
    {
        // access an internal type
        Friend f = new Friend();
        f.Hello();

        Friend2 f2 = new Friend2();
        // access an internal member of a public type
        Console.WriteLine(f2.secret);
    }
}

```

17. The Namespace Qualifier

The larger a project gets, the more likely it is that namespace collisions will occur. C# has the `::` operator which is used for specifying at what scope a namespace should be resolved. The left operand indicates the scope at which to resolve the name while the right operand is the name to resolve. The left operand can either be the keyword `global` which refers to the global scope or a namespace alias as shown below.

C# Code

```
using System;
using sys = System;

namespace TestLib{

class Test{

    public class System {}

    static DateTime Console = DateTime.Now;

    public static void Main(){
        //Console.WriteLine("Hello world"); doesn't work due to static member variable named 'Console'
        //System.Console.WriteLine("Hello world"); doesn't work due to nested class named 'System'

        global::System.Console.WriteLine("The time is " + Console);

        sys::Console.WriteLine("Hello again");
    }
}
```

NOTE: The `extern alias` statement can be used to attach an alias to an assembly which can then be used as the left operand of the `::` operator. When this is done the scope used for name resolution is the top level namespace(s) within the assembly.

18. Iterators (Continuations)

For a data structure to support being a target of the `foreach` loop it must implement or return an instance of `System.Collections.IEnumerable`. However writing an `Enumerator` can be somewhat cumbersome which is where the `yield` keyword comes in. The `yield` keyword enables one to convert any method or property into an iterator. In an iterator, one simply traverses the data structure and returns its contents one by one using the `yield return` statement and indicates the end of the sequence of values with the `yield break` statement. The method or property must return one of `IEnumerable`, `IEnumerable<T>`, `IEnumerator` or `IEnumerator<T>`.

C# Code

```
using System;
using System.Collections;

class Test{

    public static string[] fruit = {"banana", "apple", "orange", "pear", "grape"};
    public static string[] vegetables = {"lettuce", "cucumber", "peas", "carrots"};

    public static IEnumerable FruitAndVeg{

        get{

            foreach(string f in fruit){
                yield return f;
            }

            foreach(string v in vegetables){
                yield return v;
            }

            yield break; //optional
        }

    }

    public static void Main(){

        foreach (string produce in Test.FruitAndVeg){
            Console.WriteLine(produce);
        }

    }
}
```

19. Partial Types

The partial types feature enables one to define a single class, struct or interface across multiple source files. This is particularly useful when dealing with automatically generated code. Prior to the existence of this feature, it was problematic to make changes to a class that contained automatically generated code because any change that required regeneration of the code could alter or remove parts of the code written by hand. With this feature the automatically generated parts of a class can live in one source file while the user generated parts of the class can live in another. This makes it less likely that changes to one will affect the other negatively and vice versa.

A partial class is specified by prefixing the keyword `partial` to the class declaration. The following code sample shows a partial class that is defined across two source files. Note how methods and properties from one source file can reference members defined in another.

C# Code

```
using System;

partial class Test{

    public static string[] fruit = {"banana", "apple", "orange", "pear", "grape"};
    public static string[] vegetables = {"lettuce", "cucumber", "peas", "carrots"};

    public static void Main(){

        foreach (string produce in Test.FruitAndVeg){
            Console.WriteLine(produce);
        }
    }
}
```

C# Code 2

```
using System;
using System.Collections;

partial class Test{

    public static IEnumerable FruitAndVeg{

        get{
            foreach(string f in fruit){
                yield return f;
            }
            foreach(string v in vegetables){
                yield return v;
            }
        }

    }

}
```

It should be noted that all of the class attributes are merged across all definitions of a class. This means that contradictory attributes (e.g. class declared as `private` in one file and `public` in another) are disallowed.

20. Static Classes

A static class is a class that has no instance members, no instance constructors and cannot be used as a base class. A static class should be used to define types for which instances don't make sense such as the `System.Environment` and `System.Math` classes.

A static class is specified by prefixing the class declaration with the keyword `static`.

C# Code

```
using System;

public static class StaticClass{

    public static void HelloWorld(){
        Console.WriteLine("Hello World");
    }

}

public class Test{

    public static void Main(string[] args){
        StaticClass.HelloWorld();
    }

}
```

21. Nullable Types

A nullable type is an instance of the `System.Nullable` type. A nullable type can represent the normal range of values for an underlying value type as well as the `null` value. For example, the type `Nullable<bool>` can represent the values `true`, `false` and `null`. A nullable type can be declared by appending the operator `?` to the name of a value type when declaring the variable. This means that `bool?` is equivalent to `Nullable<bool>`. Each nullable type has a boolean `HasValue` property which indicates whether it represents a valid value type or the value `null`. The actual value of the nullable type is stored in its `Value` property. There is also the `GetValueOrDefault()` method which either returns the value of the nullable type or the default value of the underlying value type if it is `null`.

Nullable types are particularly useful when mapping C# objects to relational database schemas since `null` is a valid value for all data types in SQL databases.

C# Code

```
using System;

public class Test{

    public static void Main(string[] args){
```

```

int? x = 5;

if(x.HasValue){
    Console.WriteLine("The value of x is " + x.Value);
}

x = null;

//prints 0
Console.WriteLine(x.GetValueOrDefault());
}
}

```

The ?? operator is called the *null coalescing operator* and is used for testing the value of a nullable type and returning its value or an alternate if its value is null. Thus `x ?? y` is equivalent to `x == (null ? y : x)`.

C# Code

```

using System;

public class Test{

    public static void Main(string[] args){

        int? x = null;
        int y = x ?? 5;

        //prints 5
        Console.WriteLine(y);
    }

}

```

22. Anonymous Methods

Anonymous methods are a companion feature to [delegates](#). An anonymous method is a code block that can be used where a delegate method is expected. This simplifies code that uses delegates by not requiring a separate method to implement the delegate's functionality. The following code sample compares the anonymous method approach to using a named delegate

C# Code

```

using System;

public class Test {

    // delegate declaration, similar to a function pointer declaration
    public delegate void CallbackFunction(string a, int b);

    public static void PrintString(string a, int b){
        for(int i = 0; i < b ; i++)
            Console.WriteLine("{0}.) {1}", i + 1, a);
    }

    public static void Main(string[] args){

        /* anonymous code block */
        CallbackFunction cf = delegate (string a, int b){
            for(int i = 0; i < b ; i++)
                Console.WriteLine("{0}.) {1}", i + 1, a);
        };

        cf("Thirty Three", 10);

        /* using a named delegate function */
        CallbackFunction cf2 = new CallbackFunction(PrintString);
        cf2("Twenty Two", 5);
    }

}

```

There are a number of restrictions to anonymous methods that must be kept in mind. For one, jump statements like `break`, `goto` and `continue` cannot be used o jump into an anonymous method from outside the code block or vice versa. Also anonymous methods cannot refer to `ref` or `out` parameters that are defined outside the scope of the method.

Wish You Were Here

1. Checked Exceptions

Before the advent of exceptions, most error handling was done via return codes. There are many advantages of exceptions over

return codes that are typically touted including the fact that exceptions

- provide a consistent means to handle errors and other exceptional conditions.
- enable an error to propagate up the call stack if not handled in the current context.
- allow developers to separate error handling code from general application logic.

Java creates an additional wrinkle by possessing both checked and unchecked exceptions. Checked exceptions are exceptions that the calling method must handle either by catching the exception or declaring that the exception should be handled by its calling method via the `throws` clause. On the other hand, unchecked exceptions (also called runtime exceptions) do not have to be caught or declared in the `throws` clause. In a sense unchecked exceptions are similar to return codes in that they can be ignored without warnings or errors being issued by the compiler. Although if an exception is ignored at runtime, your program will terminate.

Checked exceptions are typically used to indicate to a calling method that the callee failed in its task as well as pass information as to how and why it failed. One must either catch a checked exception or declare it in the `throws` clause of the method or the Java compiler will issue an error. The reasoning behind the fact that the exception must be declared in the `throws` clause is that handling whatever errors that can occur when the method is used is just as important as knowing what parameters it accepts and the kind of type it returns.

Unchecked exceptions are typically exceptions that can occur in most parts of the program and thus the overhead of explicitly checking for them outweighs their usefulness due to the massive code bloat that would ensue. Examples of situations that throw unchecked exceptions are accessing a null object reference, trying to access an out of bounds index of an array or a division by zero. In all of the aforementioned cases it would be cumbersome to put try...catch blocks around every the code (for instance a try...catch block around every object access or every array access) and they are better off as unchecked exceptions.

In C#, all exceptions are unchecked and there is no analog to the `throws` clause. One major side effect of this is that unless the creator of the API explicitly documents the exceptions thrown then it is not possible for the users of the API to know what exceptions to catch in their code leading to unrobust applications that can fail unexpectedly. Thus users of C# are reliant on the documentation skill of programmers as their primary error handling mechanism which is a less than optimal situation.

For instance in the following code snippet, the only way to know what exceptions can be thrown by the method below is to either have the source code for all the methods called within it or if the developer of the method documents all the exceptions thrown by all the methods called within it (meaning that the developers of those methods must have done the same ad infinitum).

C# Code

```
public string GetMessageFromServer(string server)
{
    //Set up variables and String to write to the server
    Encoding ASCII = Encoding.ASCII;
    string Get = "GET / HTTP/1.1\r\nHost: " + server +
        "\r\nConnection: Close\r\n\r\n";
    Byte[] ByteGet = ASCII.GetBytes(Get);
    Byte[] RecvBytes = new Byte[256];
    String strRetPage = null;

    // IPAddress and IPEndPoint represent the endpoint that will
    // receive the request
    // Get first IPAddress in list return by DNS
    IPAddress hostadd = Dns.Resolve(server).AddressList[0];
    IPEndPoint EPhost = new IPEndPoint(hostadd, 80);

    //Create the Socket for sending data over TCP
    Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp );

    // Connect to host using IPEndPoint
    s.Connect(EPhost);
    if (!s.Connected)
    {
        strRetPage = "Unable to connect to host";
        return strRetPage;
    }

    // Sent the GET text to the host
    s.Send(ByteGet, ByteGet.Length, 0);

    // Receive the page, loop until all bytes are received
    Int32 bytes = s.Receive(RecvBytes, RecvBytes.Length, 0);
    strRetPage = "Default HTML page on " + server + ":\r\n";
    strRetPage = strRetPage + ASCII.GetString(RecvBytes, 0, bytes);

    while (bytes > 0)
    {
        bytes = s.Receive(RecvBytes, RecvBytes.Length, 0);
        strRetPage = strRetPage + ASCII.GetString(RecvBytes, 0, bytes);
    }
}
```



```

    return strRetPage;
}

```

The above code snippet is from the .NET Framework Beta 2 documentation for the Socket class. Note how there no exceptions caught in the code. If this was a method in a real application as opposed to a sample, it would be **impossible** for the users of this method to know what exceptions to catch without access to the source code or without the author of the method painstakingly checking what exceptions are thrown by every single method called and then documenting them. Below is the list of exceptions that could be thrown within the method according to their entries in the Microsoft .NET framework Beta 2 documentation.

Method	Exception(s) thrown
Encoding.GetBytes	ArgumentNullException
Dns.Resolve	ArgumentNullException, SocketException
IPEndPoint constructor	ArgumentException
Socket constructor	SocketException
Socket.Connect	ArgumentNullException, SocketException
Socket.Receive	ArgumentNullException, SocketException
ASCII.GetString	[undocumented as of Beta 2]

If the author of the method does not have time to document the exceptions that may be thrown from this method or happens to leave out an important one, such as the SocketException in this case, then the users of the method could have their applications fail unexpectedly without an elegant means of recovery in place. In the above case, the main exception of interest to users would probably be the SocketException since the others are all related to the internal workings of the method and don't really have anything to do with the caller and in fact would probably be unchecked exceptions if they existed in Java. In practice the GetMessageFromServer method would check the validity of its string parameter and throw ArgumentException or one of its subclasses depending the results of the check.

The rationale for excluding checked exceptions from C# has never been fully explained by Microsoft but [this message from Eric Gunnerson of the C# team](#) sheds some light on the reasoning behind this decision. The primary reason for this choice according to Gunnerson is that examination of small programs led to the conclusion that using checked exceptions could both enhance developer productivity and enhance code quality. On the other hand experience with large software projects suggested that using checked exceptions decreased productivity with little or no increase in code quality.

[updated 12/5/2001] *It should be noted that there is agreement amongst some Java developers with Eric Gunnerson's assertion that checked exceptions have certain disadvantages. Alan Griffiths wrote an excellent article entitled [Exceptional Java](#) where he notes that checked exceptions lead to breaking encapsulation, loss of information and information overload. Bruce Eckel, author of [Thinking In Java](#) and [Thinking In C++](#), also questions the wisdom of checked exceptions in his article entitled [Does Java need Checked Exceptions?](#).*

The lack of checked exceptions in C# will be very unsettling for Java developers and may lead to program designs which are flawed. One only has to remember the anecdote about how originally a considerable amount of the exceptions in the Java API were unchecked but upon changing them to checked exceptions a number of bugs and design flaws were found in the API. Hopefully this will be remedied in later versions of C# or a third party could develop a static source code analysis tool such as [lint](#). Meanwhile C# developers must take care to document all exceptions thrown from their methods that callers should be aware of as a matter of consideration. This is not to say that documentation should not typically exist but since it is the **only** means to ensure exception safe C# code then its importance is now a much greater.

2. Cross Platform Portability (Write Once, Run Anywhere)

A major selling point of Java™ technologies is that applications written in Java are portable across a number of operating systems and platforms. Sun officially supports Linux, Windows and Solaris but other vendors have implemented Java on [a large range of platforms](#) including OS/2, AIX and MacOS. Binary compatibility across platforms using similar Java versions is the norm except for situations involving bugs in various VM implementations.

At the time of this writing C# is only available on Windows. Efforts are currently in place to port it to other platforms, including Linux and FreeBSD. Linux porting is being done as part of the [Mono project](#) developed by [Ximian](#) while the FreeBSD implementation is a Microsoft project codenamed [rotor](#).

3. Extensions

The Java extension mechanism enables developers to extend the abilities of the core Java platform. Developers can create classes and packages which are treated by the Java runtime as if they are core Java classes and packages like java.lang, java.util, java.net, etc. This means that extensions do not have to be placed on the class path since they are treated as if they are part of the core libraries such as those in the Java runtime library, rt.jar. Extensions are contained within JAR files and once installed are available to all applications running on the target platform.

A C# parallel would be the ability to create assemblies that are treated as if they are part of the System namespace contained in

the System.dll assembly.

4. **strictfp**

In Java, `strictfp` is a modifier that can be used for class, method or interface declarations to ensure strict floating point arithmetic that conforms to behavior specified in the [IEEE standard 754 for binary floating-point arithmetic](#) (IEEE 754). Within an FP-strict expression, all intermediate values must be members of the float or double value set, depending on whether the expression is evaluating floats or doubles. Within expressions that are not FP-strict, it is allowable for the JVM implementation to use an extended exponent range to represent intermediate results. The example below clarifies the difference between FP-strict and non-FP-strict expressions.

Java Code

```
public class FPTest {  
    static strictfp double halfOfSquareFP(double n){  
        return n * 4.0 * 0.5;  
    }  
  
    static double halfOfSquareNFP(double n){  
        return n * 4.0 * 0.5;  
    }  
  
    public static void main(String[] args) {  
        double d = 6.6e+307;  
  
        System.out.println(halfOfSquareFP(d));  
  
        System.out.println(halfOfSquareNFP(d));  
    }  
}  
//FPTest
```

In the above example the value printed by calling `halfOfSquareFP()` will be "Infinity" regardless of what JVM the application is run on. This is because Java enforces left-to-right evaluation of arguments and `6.6e+307` multiplied by `4.0` exceeds the maximum value for a double thus leading to all subsequent operations yielding Infinity. On the other hand the value printed on calling `halfOfSquareNFP()` may not be the same on different JVMs depending on whether the target platform and JVM implementation support storing intermediate values in an extended format that has a larger range than that of doubles. Thus on some platforms the value `1.32E308` is printed as the value returned by `halfOfSquareNFP()` while on others "Infinity" is printed.

Section 4.1.5 of the C# specification covers floating point numbers and states that due to the excessive performance costs of enforcing that certain architectures perform operations with *less* precision than is possible, there is no way to enforce FP-strictness in C#.

5. Dynamic Class Loading

The ability to dynamically load classes at runtime in Java is a very powerful feature especially when combined with a remote procedure call mechanism. Dynamic class loading enables Java applications to download the class files (i.e. byte codes) of classes that do not exist on the target machine. An object type that only exists on one machine can be transferred to other machines in a seamless and transparent manner. Thus new types can be introduced on a remote machine which allows the behavior of remote applications to be significantly extended at runtime. The following example shows an example of a remote application that accepts types that implement a certain interface, `IStockTicker`.

Java Code

```
public class MyRMIServer extends UnicastRemoteObject
    implements SomeInterface {

    public MyRMIServer() throws RemoteException{ super();}

    public String obtainName(IStockTicker ticker){

        String stock_ticker = ticker.getTicker();

        if(stock_ticker.equalsIgnoreCase("MSFT"))
            return "Microsoft Corporation";
        else if(stock_ticker.equalsIgnoreCase("SUNW"))
            return "Sun Microsystems";
        else
            return "Unknown Stock Ticker";

    }/* obtainName(IStockTicker) */
}
```

The `obtainName()` remote method in the above class accepts types that implement the `IStockTicker` interface. It is possible for this method to be invoked from a remote client which then passes a type that implements `IStockTicker`, for example `NASDAQStock`, that does not exist on the server where the `MyRMIServer` class lives. In this case the entire code needed for `NASDAQStock` class is transmitted from the client to the remote server automatically.

C# and the .NET Remoting mechanism also enable remotely downloading classes from one machine to the other but the client has to publish the assembly and the server can then load it via a URL.

For information on Java Remote Method Invokation (RMI) read the [Java tutorial on RMI](#). Information on .NET Remoting with C# is explained in this [introduction to .NET remoting](#) and this [technical overview on MSDN](#).

6. Interfaces That Contain Fields

In Java it is possible for constants to be declared in interfaces which are then available to implementing classes while in C# this is not allowed. This may not be a big issue in C# since the primary usage of constants declared in interfaces is as a poor emulation of [enumerations](#).

7. Anonymous Inner Classes

An anonymous inner class is a class declaration that occurs at the same point where an instance of that class is created. Anonymous inner classes are typically used where only one instance of a type will exist in the application. The most popular usage of anonymous inner classes is for specifying callbacks especially in the Java GUI libraries but there are other situations where anonymous inner classes are beneficial as well. Below is an example of using anonymous inner classes to implement the [State Design Pattern](#).

Java Code

```
/* An instance of this class represents the current state of a ClientView GUI. */

public abstract class ClientState{

    // This instance of the class is used to signify that the user is not logged in.
    // The only thing a user can do in this state is login and exit.

    public static ClientState NOT_LOGGED_IN = new ClientState() {

        public void setMenuState(ClientView cv) {

            cv.setMenuEnabledState(false); /* disable all menus */
            cv.menuLogin.setEnabled(true);
            cv.menuExit.setEnabled(true);

            //can't type
            cv.textArea.setEnabled(false);
        }

    }
```

```

        public String toString(){
            return "ClientState: NOT_LOGGED_IN";
        }
    };

    // This instance of the class is used to signify that the user is logged in
    // but has not yet created a document to work with. The user cannot type or save
    // anything in this mode.
    public static ClientState NO_OPEN_DOCUMENT = new ClientState() {
        public void setMenuState(ClientView cv) {

            cv.setMenuEnabledState(false); /* disable all menus */
            cv.menuLogin.setEnabled(true);
            cv.menuExit.setEnabled(true);
            cv.menuOpenFile.setEnabled(true);
            cv.menuNewFile.setEnabled(true);

            //can't type
            cv.textArea.setEnabled(false);

        }

        public String toString(){
            return "ClientState: NO_OPEN_DOCUMENT";
        }
    };

    // This instance of the class is used to signify that the user is editing a file.
    // In this mode the user can use any functionality he/she sees fit.
    public static ClientState EDITTING_DOCUMENT = new ClientState() {
        public void setMenuState(ClientView cv) {

            cv.setMenuEnabledState(true); /* enable all menus
            cv.textArea.setEnabled(true);
        }

        public String toString(){
            return "ClientState:EDITTING_DOCUMENT";
        }
    };

    // Default constructor private to stop people from directly creating instances
    // of the class.
    private ClientState() {};

    // This disables various elements of the ClientView's menu dependent on which
    // ClientState object this is.
    public abstract void setMenuState(ClientView cv);
} // ClientState

```

Below is an example of the code that would utilize the above ClientState class.

```

bool loginUser(String username, String passwd) {

    //check if already logged in
    if(myGUI.state == ClientState.NOT_LOGGED_IN)
        return true;

    //enable parts of the GUI if the user authenticates
    if(userAuthenticated(username, passwd)){

        myGUI.state = ClientState.NO_OPEN_DOCUMENT;
        myGUI.state.setMenuState(myView);
        return true;

    }

    return false;

}

/* loginUser(String, String) */

```

8. Static Imports

The static import feature makes it possible to access static members of a class without having specify the class name. This feature is intended to reduce the verbosity of code that frequently access the static members of a particular class (e.g. constants defined in

a particular helper class).

A static import similar to a regular `import` statement except that the keyword `static` is used and instead of importing a package, a specific class is imported.

Java Code

```
import static java.awt.Color.*;

public class Test{

    public static void main(String[] args) throws Exception{

        //constants not qualified thanks to static import
        System.out.println(RED + " plus " + YELLOW + " is " + ORANGE);
    }
}
```

Conclusion (2001)

Most developers, especially those with a background in C or C++, would probably agree that features like operator overloading, pointers, preprocessor directives, delegates and deterministic object cleanup make C# more expressive than Java in a number of cases. Similarly, Java developers who learn C# will be pleasantly surprised at features that are missing in Java that will seem glaring in their absence once one uses them in C#, such as boxing, enumerations and pass by reference. On the other hand the lack of checked exceptions, inner classes, cross platform portability or the fact that a class is not the smallest unit of distribution of code makes the choice of C# over Java not a clearcut case of choosing more language features without having to make any compromises.

It is my opinion that both languages are similar enough that they could be made to mirror each other without significant effort if so required by either user base. In this case, C# would have it easier than Java in that C# has less to borrow from Java than Java would have to borrow from C#. However, the true worth of a programming language that is intended for use outside of academia is how quickly the language evolves to adapt to the changing technological landscape and what kind of community surrounds the language. Some programming languages are akin to the French language under the [Les Immortels of the Académie Française](#) in France. Les immortels are charged with dictating what constitutes the official French language but they have been slow to adapt to the information age thus their edicts on what constitutes the proper French versions of new words, especially those related to technology, are usually ignored especially since they either conflict with what the general French public would rather call them and show up long after the unsanctioned words have already entered the lexicon. C++ is an example of a language that has undergone a process of balkanization closely resembling the French language under the Les Immortels of the Académie Française while Java under Sun Microsystems can be considered to be a language that has evolved with the times similar to how the English language has done.

Thus the question really becomes, which of these languages looks like it will evolve with the times and be easiest to adapt to new situations as they arise? So far, Sun has done a great job with Java although a lack of versioning support and the non-existence of a framework that enables extensibility of the language built into the platform makes drastic evolution difficult. C# with its support for versioning via the .NET framework and the existence of attributes which can be used to extend the features of the language looks like it would in the long run be the more adaptable language. Only time will tell however if this prediction is accurate.

Conclusion (2007)

As predicted in the original conclusion to this paper, a number of features have become common across both C# and Java since 2001. These features include generics, foreach loops, enumerations, boxing, variable length parameter lists and metadata annotations. However after years of convergence it seems that C# and Java are about to go in radically different directions. The current plans for C# 3.0 are highlighted in the [Language Integrated Query \(LINQ\) Project](#) which encompasses integrating a number of data oriented features including query, set operations, transformations and type inferencing directly into the C# language. In combination with some of C#'s existing features like anonymous methods and nullable types, the differences between C# and Java will become more stark over the next few years in contrast to the feature convergence that has been happening over the past few years.

Resources

1. Eckel, Bruce. [Thinking In Java](#). Prentice Hall, 2000.
2. Gunnerson, Eric. [A Programmer's Introduction To C#](#). Apress, 2001.
3. Sun Microsystems. [The Java™ Tutorial](#). <<http://java.sun.com/docs/books/tutorial/>>
4. Microsoft Corporation. [.NET Framework Programming](#). <[http://msdn2.microsoft.com/en-us/library/ms229284\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms229284(VS.80).aspx)>
5. Microsoft Corporation. [C# Language Reference](#). <[http://msdn2.microsoft.com/en-us/library/618ayhy6\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/618ayhy6(VS.80).aspx)>

Acknowledgements

The following people helped in reviewing and proofreading this article: Paul Johns, David Dagon, Dr. Yannis Smaragdakis , Dmitri Alperovitch, Dennis Lu and Sanjay Bhatia.

© 2001, 2007 [Dare Obasanjo](#)