A Comparison of Two Programming Languages

Java and C#

Hao Zheng

A Major Report

in The Department of Computer Science

Presented in Partial Fulfillment of the Requirements for

the Degree of Master of Computer Science

at Concordia University

Montréal, QC, Canada

Your file Votre référence
ISBN: 0-612-91160-8
Our file Notre référence
ISBN: 0-612-91160-8

Abstract

A Comparison of Two Programming Languages: Java and C#

Hao Zheng

Java programmers must be aware of the advent of C#, the .NET network environment, and a host of new supporting technologies, such as web service. Before taking the big step of migrating all development to a new environment, programmers will be eager to understand what are the advantages and disadvantages of both languages and whether it is worth while to make the big step or not. Java and C# are both good object-oriented programming languages. In general, Java and C# looks astonishingly alike: they include language features like single inheritance, interfaces, nearly identical syntax, and compilation to an intermediate format. However, C# distinguishes itself from Java with language design features borrowed from C++ and other languages, direct integration with COM (Component Object Model), and its key role in Microsoft's .NET Windows networking framework. In this report, I will compare both languages to expose the similarities and differences between them. Some new features in C# which make it interesting for java programmers are also discussed in this paper.

Contents

List of Tables

Chapter 1 Introduction

Java and C# are both strongly-typed objected oriented programming languages. Java and C# were designed with simplicity, expressiveness, and performance in mind. Both compile to machine independent code which runs in a managed execution environment. Java uses Java byte code and the Java Virtual Machine (JVM); C# used Microsoft Intermediated Language (MSIL) provided by the Common Language Runtime (CLR) of the .NET platform.

Java came on the programming language landscape around 1995. It is easily learned, and prohibits many types of programming errors. Its syntax is like that of the C language. The Java programming language evolved from a language named Oak by James Gosling in 1991. [5] Oak was first slated to appear in television set-top boxes designed to provide video-on-demand services. Just as the deals with the set-top box manufactures were falling through, the World Wide Web was coming to life. As Oak's developers began to recognize this trend, their focus shifted to the Internet and WebRunner, an Oak-enabled browser, was born. Sun formally announced Oak's name was changed to Java and WebRunner became the HotJava web browser in 1995. The excitement of the Internet attracted software vendors such that Java development tools from many vendors quickly became available. Java is particularly designed to interface with web pages and to enable distributed applications over the internet, since the web is becoming a dominant software development arena, this drives Java as a well supported and most widely used language.

Microsoft initially resisted Java but eventually it became a part of its so much famous Visual Studio and named VJ++. But the Microsoft efforts to giving Java flavor to Visual Studio did not succeed and VJ++ ultimately failed. There were of course, other reasons like poor marketing for its failure. Consequently Microsoft has not released new versions of VJ++ after 1998 and has no plans for it in near future.

Since then, Microsoft started developing a new language ---- C# (pronounced C-sharp) which conceptually is much like Java but provides additional features missing in Java. C# 1.0 language was submitted by Microsoft to the ECMA standards group in mid-2000, and released with Visual Studio .NET 2002 and 2003. C# is developed by Anders Hejlsberg for Microsoft's .NET platform, which is Microsoft's platform for XML Web service, and it is arguably the cleanest, most efficient language for .NET in popular use today. The C# language is aimed at enabling programmers to quickly build a wide range of applications. The goal of C# and the .NET platform is to shorten development time by freeing the developer from worrying about several issues such as memory management, type safety issues, building low level libraries, array bounds checking, etc. thus allowing developers to actually spend their time and energy working on their application and business logic instead. C# bears a striking resemblance to Java and improves on that language. It may well become the dominant language for building applications on Microsoft platforms.

It is important to note that this report does not cover the complete aspects of these two languages and that there are many features of these two languages which are not reported here. In addition, at the time of writing this report, in October 2003 Microsoft has released a draft of the C# 2.0 specification, which announced four new features: generics, iterator,

anonymous methods and partial types. However, since it is still a template, only a small part of iterator is discussed in this report. In chapter 2, I present the similarities of Java and C#. In chapter 3, I explain a couple of high-level, fundamental differences in scope between Java and C#. In chapter 4, I introduce some new features in C# beyond Java. In chapter 5, I illustrate the GUI part using Java Swing and C# Windows Form. In chapter 6, I close the report by evaluating the wisdom of developing application in these two languages.

Chapter 2 Similarities of C# and Java

Anders Hejlsberg, the creator of C# language from Microsoft says that the C# language definition has been primarily derived from C and C++, and many elements of the language reflect that. [11] However, C# looks astonishingly like Java; it includes language features like single inheritance, interfaces, nearly identical syntax, and compilation to an intermediate format. This section explores similar features of C# and Java. Notice that the similar features in these two languages are not necessarily identical features; therefore, there are still some differences within the similarities.

2.1 Object-oriented

The object-oriented programming paradigm first is introduced in Simula 67, a language designed for making simulations, created by Ole-Johan Dahl and Kristen Nygaard [1]. Alan Kay's group at Xerox PARC used Simula as a platform for their development of Smalltalk (first language versions in the 1970s), extending object-oriented programming importantly by the integration of graphical user interfaces and interactive program execution. Object-oriented programming became as the dominant programming methodology during the mid-1980s, largely due to the influence of C++. In the past decade Java has merged in wide use as a popular object-oriented programming language. Most recently, besides Java, C# became the most commercially important object-oriented languages.

Java and C# both are object-oriented languages that use a single rooted class hierarchy. In C#, these three most important principles of object-orientation have been preserved. To qualify as an object-oriented language, there are four main concepts that an object oriented programming language must support---Objects, Abstraction, Inheritance and Encapsulation. [1] Every class in Java is a subclass of `java.lang.Object`, and every class in C# is a subclass of `System.Object`. Both Java and C# treat data as a critical element in the program development. Every problem is decomposed into a number of entities called *objects* and then built data and functions around these entities.

One thing to note is that there are no more global function and variables, methods and fields modified by the `static` keyword are mechanisms for providing "global" functions and variables. Abstraction is the ability of a programming language to define internal types and be able to control external access to the types. C# has similar abstraction technique to Java. Encapsulation enables programmers to group actions together with their respective owners. Everything must be encapsulated inside a class. This makes a C# code more readable and also reduces naming complexity. C# class members can be declared as `private`, `protected`, `public`, `internal` or `static`, thus facilitating complete control over their encapsulation and information hiding. The last concept is inheritance. Inheritance is the foundation and fundamental support for object-orientation. It allows a class to "inherit" the characteristics and attributes of the parent class, but Java and C# do not allow multiple inheritance, hence they provide interfaces as an alternative to it.

2.2 Case Sensitivity and Naming Convention

Java and C# both are case-sensitive languages. Both languages generally follow the

naming convention that all keywords are in lower case letters and class names start with a capital letter. However, case sensitivity shouldn't be used to differentiate program elements. The multilanguage support in the .NET runtime system means that components developed in C# can be called from components written in other .NET languages such as VB.NET that might not be able to differentiate on the basis of case. This is discussed in section 3.5.

In Java naming convention, a class must be written in a file with the same name, including the capitalisation, and with the extension. For example, a class named `QuickSort` should be written in a file named `QuickSort.java`. The variable names and method names should start with lower case letter, and constants usually written in all capitals. In the naming convention used by .NET, if a single class name contains multiple words, each word's starting letter would be capital. The same applies to function names as well.

Java code:

```
// In a file named QuickSort.java, class has the same name "QuickSort"
public class QuickSort{
    // Constant in all capital letters
    final static int DATA_ARRAY_SIZE = 100;
    // Variable name in lower case letters
    String str = "QuickSort Animation Applet";
    // Method name in lower case letters
    public static void main( String args[] ){
    QuickSort appletObject = new QuickSort();
      System.out.print( str );
    }
}
```

C# code:

```
namespace QuickSortCSharp
{
    public class QuickSort
    {
        const int DATA_ARRAY_SIZE = 100;
        string str;
       // Method name in capital letters
        static void Main ( string[ ] args )
        {
          str = Console.ReadLine( ) ;
            Console.WriteLine ( str ) ;
        }
    }
}
```

2.3 Namespaces and Classes

In Java, packages are physical directories using for providing access protection and file management. Namespace in C# are analogous to packages in Java, but takes a more practical approach, because it represents a logical hierarchy rather than a physical layout of source files. Namespaces can be used to semantically group elements for organization and readability of classes and other namespaces. Names declared in one namespace will not conflict with the names declared in another namespace. It can be nested to any level.

Classes are declared very similar in both Java and C#. The Java keyword `import` is the same as C# keyword `using`, which performs the same basic function. The point at which a class begins execution is the static method `main()` in Java, and `Main()` in C#. The following code from Java and C# demonstrates the basic form:

Java code:

```
import java.lang.System;
class QuickSort {
     // entry point of a class is static main() method in Java
   public static void main(String[] args) {
      System.out.println("QuickSort Animation say hi.");
   }
}
```

C# code:

```
using System;
class QuickSort {
   // entry point of a class is static Main() method in C#
      static void Main() {
    Console.WriteLine("QuickSort Animation say hi ");
    }
}
```

Classes can be abstract: A class that is declared as abstract cannot be instantiated; it can only be used as a base class. Classes also can be unextendable: The C# keyword `sealed` is like the Java keyword `final`, which declares a class to be non-abstract, but it also cannot be used as the base of another class.

2.4 Abstract Class

An abstract class represents the common elements of a set of related classes. The concrete classes inherit the definitions in the abstract class. The abstract class forces a requirement on the concrete classes to implement the abstracted methods. The concept of abstract class is similar in Java and C#.

Both Java and C# make it easy to define an abstract class using the `abstract` keyword. The abstract class may contain abstract methods. Both in Java and C# all abstract methods should be implemented in subclass, but in C#, the implementation is provided by an overriding method, which is a member of a non-abstract class. Abstract methods also can be `virtual` method in C# but not in Java, because all methods in Java are implicitly `virtual`, so there's no need for this keyword in the language.

Java code:

```
// abstract base class declaration in Java
abstract class  SortAlgorithm {
     SortPanel parentPanel;
     //abstract class constructor
     public SortAlgorithm() {
     //call base class constructor
     super();
     }
     public void setparent(SortPanel sortPanel) {
   this.parentPanel = sortPanel;
     }
     //abstract method in base class
     abstract public void sort(int[] data);
}
//class inherited from abstract base class
class QSortAlgorithm extends SortAlgorithm
{
     //override abstract method in base class
     public void sort(int a[])
     {
          quickSort(a, 0, a.length - 1);
     }
}
```

C# code:

```
// abstract base class declaration in C#
abstract class SortAlgorithm
{
    public SortControl parentpanel;
    //abstract class constructor
    public SortAlgorithm() :base()  //explicit constructor call
    {  }
    public void setparentpanel(SortControl sortControl)
    {
        this.parentpanel = sortControl;
    }
    //abstract method declaration
    public abstract void sort(int[] data);
}
//class inherited from abstract base class
class QSortAlgorithm :SortAlgorithm
{
    //override abstract method in base class, virtual keyword can be used in which case
    //the method or the property is called a virtual member
    public override void sort(int[] a)
    {
        Console.WriteLine("override sort function here");
    }
}
```

2.5 Garbage Collection

Garbage collection is one of Java's most popular features, which is very useful for memory management and helps in reducing the complexity of application development. [6] C# has followed the Java way of automatic garbage collection, which relieves the programmer of the burden of manual memory management. With automatic garbage collection, manually

implement destructors for every object is not necessary, and C# automatically removes from memory all objects that a program no longer references. It does have a Java-like 'garbage collection' scheme in which the runtime system sporadically reclaims memory from objects automatically which occurs automatically without the knowledge of the programmer.

Java and C# provide garbage collection make the life easier for the programmer, because they don't need to do all of the disposal of memory and run-time is in charge. Both garbage collection run in the background and cleans up the objects some time after all references to them have been dropped. This is an advantage that makes no memory leaks of the system. [6]

In both Java and C#, all objects are allocated on the heap with `new` keyword which provides the objects' reference. The garbage collector then periodically frees the memory used by objects that are no longer referenced, which removes the programmer from having to deal with the memory space that is allocated to objects. Programmers don't need to worry about creating too many objects, but don't want to waste any that they've already got sitting in memory. It's a good idea not to create any more objects than really need, thus avoiding running the garbage collector too often.

2.6 Interface and Single Inheritance

Similar to Java, C# does not support multiple inheritance which is the ability of a class or interface to extend more than one class or interface; instead it provides Java's solution: interfaces. Interfaces implemented by a class specify certain functions that the class is

guaranteed to implement. Multiple inheritance causes more problems and confusion than it solves. Interfaces avoid the dangers of multiple inheritance while maintaining the ability to let several classes implement the same set of methods.

In C# just as in Java, an interface is an abstract definition of a collection of methods. When class implements an interface, it must implement all of the methods defined in the interface. A single class can implement a number of interfaces. There may be some subtle differences that surface later, such as *extends* and *implements* keywords of Java are replaced with a colon (:). In Java, the modifier *public* could be presented in a method signature; however it is illegal in C# for explicitly specifying an interface method as *public*.

Java code:

```
class Control {...}
//interface definitions
interface IControl
{
  void Paint();
}
interface IDataBound
{
   void Bind(Binder b);
}
// keyword extend and implement are used in Java to indicate inheritance
public class EditBox extends Control implements IControl, IDataBound
{
   public void Paint() {...}
   public void Bind(Binder b) {...}
}
```

C# code:

```
class Control {...}
//interface definitions
interface IControl
{
   void Paint();
}
interface IDataBound
{
   void Bind(Binder b);
}
// C# uses the colon ":" operator to indicate inheritance
public class EditBox : Control, IControl, IDataBound
{
   public void Paint() {...}
   public void Bind(Binder b) {...}
}
```

Multiple inheritance is a very powerful technique, and in fact, some problems are quite difficult to solve without it. Multiple inheritance can even solve some problems quite elegantly. However, multiple inheritance can significantly increase the complexity into the implementation. This complexity impacts casing, field access, serialization and probably lots of other places.

As with operator overloading, the designers of Java and C# decided that the increased complexity of allowing multiple inheritance far outweighed its advantages, so they eliminated it from the language. In some ways, the Java and C# language construct of interfaces compensates for this; however, the bottom line is that Java and C# do not allow conventional multiple inheritance.

Like Java, C# gives up on multiple class inheritance in favor of a single inheritance model extended by the multiple interfaces implementation. C# follows the Java path by allowing inheritance from only a single base class. An inheritance relationship signifies an IS A relationship between two classes. The lack of multiple inheritance becomes a factor when considering it is possible for a class to be classified through multiple IS A relationships. Just like Java, a class can inherit from one base class, and a class can implement multiple interfaces.

2.7 Exception Handling

Most object oriented languages provide a feature called exceptions. In the most basic sense, exceptions are unexpected events that occur within a system. Exceptions provide a way to detect problems and then handle them. Exception handling in C# is almost same as in Java except some minor differences. Both languages use of the `try` and `catch` blocks for exception handling. The `try` block indicates the region under inspection and the catch block for handles the exceptions. There can be multiple `catch` blocks to handle multiple exceptions and the `finally` block for handling any of the exceptions not handled by the previous catch blocks. This code will run whether or not an exception is generated.

In Java all exceptions are derived from `java.lang.Throwable`. In C#, they are all descended from `System.Exception`. In both Java and C# the exception is signaled by a `throw` statement that transfers control to a matching `catch` clause of a logically enclosing `try` block. The processing of a `throw` statement in the Java language specification occupies several very dense pages of technical vocabulary. The exact

language need not be reproduced here. However, there is a problem with the Java specification that needs to be confronted before anything else.

In Java, C# and other languages such as C++ and Visual Basic, `finally` statement exists to be used for processing after leaving the `try/catch` block. It executes all the statements in the finally clause, then resumes looking through the stack for a *catch* to match the pending throw. `Finally` doesn't just trap exceptions. It also traps any of the jump statements (`break, continue, return`, or `goto`) that would exit from a `try` block that contains a `finally` clause. For that matter, the `finally` clause also executes when the try block ends normally and falls through after executing the last statement.

Java has a problem related to `finally` clause that C# corrects. Java allows a `finally` block to include one of the jump statements; however, if the `final` block is entered from a `throw` statement, then execution of a `break, continue`, or `return` in the `final` block overrides the `throw` statement and aborts error handling. This is a design problem because the `finally` clause cannot actually handle the error since it doesn't have access to the exception object and cannot even determine that it has been entered as a result of a `throw` rather than from some other cause. C# solves this problem by prohibiting any jump statement in a `finally` block that would transfer control outside the block.

Java code:

```
public final void run() {
   while (Thread.currentThread() == animateTread) {
   //one try statement for the entire method makes the code is easier to read
     try {
        if (!stopRequest && sortAlgorithm != null) {
```

```
            sortAlgorithm.sort(dataArray);
            stopRequest = true;
                    System.out.print( "\n" + "result  ");
                    for(int i=0;i<dataArray.length;++i)
                        System.out.print(dataArray[i]+" ");
            System.out.print("\n");
          }
             Thread.sleep(10);
       }
          /*catch InterruptedException, which would be thrown if the thread in the
          Thread.sleep() call stops prematurely.
          */
          catch ( InterruptedException e) {
              System.out.println("Thread was interrupted in SortPanel.run()");
       }
          // finally block executes and cleans up the state of method
          finally {
              System.out.println(" Execute the finally block");
          }
    }
}
```

C# code:

```
public void PerformSort()
{
    while (Thread.CurrentThread == t)
    {
        //one try statement for the entire method makes the code is easier to read
        //concept is the same as in Java
        try
        {
            if (!stopRequest && sortAlgorithm != null)
            {
```

```
                sortAlgorithm.sort(dataArray);
                stopRequest = true;
                Console.WriteLine( "\n" + "result  ");
                for(int i=0;i<dataArray.Length;++i)
                    Console.WriteLine(dataArray[i]+" ");
                Console.WriteLine("\n");
            }
            Thread.Sleep(10);
        }
        /*catch ThreadInterruptedException, which would be thrown if the thread in the
        Thread.Sleep()  call stops prematurely.
        */
        catch ( ThreadInterruptedException e)
        {
            Console.WriteLine("Thread was interrupted in SortPanel.PerformStart()");
        }
        //finally block executes and clean up the state of the method
        finally
        {
            Console.WriteLine( "Executing finally block");
        }
    }
}
```

2.8 Arrays

Reference type is a non-primitive type, where a variable of that type evaluates to the address of the location in memory where the object referenced by the variable is stored. Arrays are reference types in both C# and Java. Because of their importance, both language implement native language syntax to declare and manipulate arrays. In Java, multi-dimensional arrays are implemented solely with single-dimensional arrays (where arrays can be members of other arrays). Java doesn't support true multidimensional arrays.

A true multidimensional array is rectangular arrays that represent an n-dimensional block. 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.

C# supports single-dimensional array, multidimensional arrays. There are two types of multidimensional array in C#, rectangular array and jagged arrays. [3] A rectangular array is a single array with more than one dimension, with the dimensions' sizes fixed in the array's declaration. 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 syntax for defining an array in looks similar in C# to that in Java, by placing empty square brackets with the type and the variable name. However, it is less flexible in C# than that in Java. For example,

```
int[] x = { 0, 1, 2, 3 };
int x[] = { 0, 1, 2, 3 };
```

These two declarations are both legal in Java. In C# only the first line is valid. The empty square brackets cannot be placed after the variable name; it must follow the type specification.

Array instances are created using the *new* keyword both in Java and C#. Both java and C# consider arrays as objects. However, the objectlike features of arrays in C# are more

extensive that those in Java. In Java, arrays are not directly exposed as objects, but the runtime system enables an instance of an array to be assigned to a variable of type `java.lang.Object` and for any of the methods of the `Object` class to be executed against it. In C#, all arrays inherit from the abstract base class `System.Array`, which derives from `System.Object`. The `System.Array` class provides functionality for working with the array, some of which is available in `java.util.Arrays`.

2.9 Declaring Constants

Java uses the `static final` keyword to declare compile time constants, and the compiler prevents any other object from changing the value of the variable. C# uses the `const` keyword to declare compile time constants while the `readonly` keyword is used for runtime constants. The semantics of constant primitives in C# and Java is the same, but there is a little difference. In Java, final members can be left uninitialized when declared but then must be defined in the constructor. In C#, constants must have a value initialized when declaring them. Failing to do so will result a C# compiler error.

Java code:

```
static final ONE =1;
// this expression is converted to 2 * 1
static final TWO = 2 * ONE;
```

C# code:

```
const int ONE = 1;
// this expression is converted to 2 * 1
int TWO = 2 * ONE;
```

Both Java and C# compiler can evaluate a constant value at compile time. However, in the previous example, marking the variable ONE *const* causes the value to be evaluated before compiling which makes the compiled program to run faster.

2.9 Constructors, Destructors and Calling Base Class Constructors

In both Java and C#, the constructor initializes an object when it is created and their syntaxes are the same. In both languages, a constructor is invoked using the *new* keyword, or use the various methods of reflection to create an instance of a class. C# supports two types of constructors: *instance constructors* and *static constructors*, while Java doesn't have the *static constructors*.

In C#, a destructor can be called explicitly to release and terminate the unmanaged resources just like in C++. The destructor in C# is similar to the *finalize* method in Java. This is because they are called by the garbage collector rather than explicitly called by the programmer. Furthermore, like Java finalizers, they cannot be guaranteed to be called in all circumstances (this always shocks everyone when they first discover this).

Java code:

```
class ConstructorExample
{
   public static void main(String[] args) {
      A.F();
      B.F();
   }
}
class A
{
```

```
    // static block in Java
    static {
        System.out.println("Init A");
    }
    public static void F() {
        System.out.println("A.F");
    }
}
class B
{
    static {
        System.out.println("Init B");
    }
    public static void F() {
        System.out.println("B.F");
    }
}
```

output:

```
Init A
A.F
Init B
B.F
```

C# code:

```
using System;
//static Constructor in C#
class StaticConstructorSample
{
    static void Main() {
        A.F();
        B.F();
    }
}
```

```
class A
{
   static A() {
      Console.WriteLine("Init A");
   }
   public static void F() {
      Console.WriteLine("A.F");
   }
}
class B
{
   static B() {
      Console.WriteLine("Init B");
   }
   public static void F() {
      Console.WriteLine("B.F");
   }
}
```

output:

```
Init A
A.F
Init B
B.F
```

In Java, parent class members can be accessed in the child class by using `super.method()` format and invoke super class' constructors in java can use the notation `super()` as the first line in constructor. C# uses the `base` keyword and C# syntax for calling the base class constructor is reminiscent of the C++ initializer list syntax.

Java code:

```
abstract class SortAlgorithm
{
        public SortControl parentpanel;
        public SortAlgorithm()
        {
            // calling the base class constructor
            super();
         }
}
```

C# code:

```
abstract class SortAlgorithm
{
        public SortControl parentpanel;
        public SortAlgorithm() :base()// calling the base class constructor
        { }
}
```

2.10 Summary

In this chapter, a quite few similar features in Java and C# are presented, which makes programmers come to the thought that "C# seems is a Java-like language". The syntax and semantics of these features in both languages are similar: Object-Oriented, Namespace vs Package, Garbage collection, Exception Handling, Abstract Classes, Arrays, Constants, and Constructors and Calling Base Class Constructors. There are also some differences among these similar features such as declaration of array and constants.

Chapter 3 Difference of C# and Java

C#'s most intriguing facets are its differences from Java, not its similarities. There are a few significant differences between these two languages. Since the developers of C# had the advantage of carefully examining Java while developing their language, it is not surprising that some of the differences attempt to address significant problems that are difficult to deal with in Java. This section covers features of C# that Java implements differently.

3.1 Data Types

Java has *primitive* and *reference* types; C# refers to them as *value* and *reference* types. *Value* types include *byte*, *int*, *long*, *float*, and *double*. *Reference* types include *class*, *interface*, and *array*. In Java, the benefit of value types stems from their simplicity relative to objects, resulting in performance benefits and memory savings compared with the alternative of implementing everything as object. C# has a third category, the infamous *pointer*. Java doesn't provide a *pointer* type, primary because of the complexity of these types and the dangers they pose to application stability when used incorrectly.

C# has wider variety of data types than Java. C# has simple type consisting of integer type, boolean type, char type, floating-point type, decimal type, struct type and enumeration type. C# reference type consists of object type, class type, interfaces, delegates (which is introduced in section 4.4), string type, and arrays. Unlike Java, C# has unsigned types

(*byte, ushort, uint, ulong*). Furthermore, every type in C# is an object, because every data type in C# directly of indirectly derived from *System.Object*, and object is the ultimate base class of all types. This gives all value types, including primitives such as *int* and *long*, *object* capabilities. In C#, the primitive types are treated as objects as and when necessary. C#'s type system is unified such that value of any type can be treated as an object. Any value type can be converted to reference type and vice versa using *boxing* and *unboxing*, which will be discussed in section 4.7.

Although C# maintains strong parallels with the data types offered by Java, the designers of C# have also drawn heavily on the features of C and C++. For the Java programmer, C# data types include many subtle and confusing difference as well as some new features. Java provides a set of data types suitable for the resolution of most contemporary business computing problems---complexity, incompatibility and security. However, the unified type system provided by the .NET Framework, as well as the extended selection of C# data and member types, offers greater flexibility and more control to the programmer.

3.2 Switch Statement

The *switch* statement is used to conditionally execute statement blocks. The value of an expression defines which *case* is executed. Both Java and C# allow the control flow to implicitly fall through different cases. For example, if a *case* is not explicitly left, the switch statement keeps executing through all *cases* up to the end of the statement.

In C#, the *switch* statement is an enhanced version of switch statement in Java. Java accepts only integral values in switch statement, but C# supports a broader range of data

types for the `switch` expression including enum and built in integer types. Most usefully, C# supports the use of strings as `switch` expression.

C# code:

```
string direction, str;
switch (direction){
    case "first" : str = "North";
        break;
    case "second" : str = "South";
        break;
    case "third" : str = "East";
        break;
    case "forth" : str = "West";
        break;
    default:
        break;
}
```

Non-empty cases within `switch` statements must end with a statement that causes a control-flow transfer. Usually cases end with a `break`; although they can end with `goto case X`; where `X` is another case in the `switch` statement.

3.3 Intermediate Language

Both C# and Java compile initially to an intermediate language: Java code runs as Java Virtual Machine (JVM) bytecodes that are either interpreted in the JVM or JIT (Just-in-Time) compiled. C#, as part of the .NET framework, is compiled to Microsoft Intermediate Language (MSIL), and is run as JIT compiled bytecodes or compiled entirely into native code. The MSIL is run within the CLR (common language runtime) which is a

stack-based virtual machine as JVM. The CLR, in turn, converts the MSIL into commands or code that will run on a particular operating system. Moreover, since the other languages that make up the .NET platform (including VB) compile to MSIL, it is possible for classes to be inherited across languages.

Microsoft is very flexible about choosing when MSIL is compiled to the native machine code. The company takes care to say that MSIL is not interpreted, but compiled to machine code. It also understands that many -- if not most -- programmers accept the idea that Java programs are inherently slower than anything written in C. The implication is that MSIL-based programs (written in C#, Visual Basic, "Managed C++" -- a version of C++ that conforms to the CLS -- and so on) will outperform "interpreted" Java byte code. Of course, this has yet to be demonstrated, since C# and other MSIL-producing compilers have not yet been released. However, the ubiquity of JIT compiler (Just-In-Time compiler) for Java make Java and C# relatively equal in terms of performance. Java byte code and MSIL are very similar languages. Java byte code and MSIL are both intermediate assembly-like languages that are compiled to machine code for execution, at runtime or otherwise.

3.4 Platform Interoperability

Java has the strength in platform interoperability because it is designed to be platform independent. Java source code is compiled into intermediate byte-codes, which are then interpreted at run-time by a platform-specific Java Virtual Machine (JVM). This allows developers to use any compiler they want on any platform to compile code, with the assumption that this compiled byte-code will run on any supported operating system. Since

a JVM can be installed on any platform before the java code could be run, this makes Java is truly supported on every OS with JVM.

C#, as part of the .NET framework, is compiled to an intermediate language--Microsoft Intermediated Language (MSIL), hence .NET is currently only fully available on Windows platforms. However, if developers are creating applications for Windows platform, it's more sensible to use C# over Java, because naturally Java doesn't ship with as many Windows-rich features. It's not that C# is better than Java, but Microsoft has created all the classes to handle Windows-based tasks and made them available through wizards, drag and drop, and point and click.

.NET takes care of this is by creating a managed execution environment based on a Common Language Runtime (CLR), where every aspect of code execution is controlled, regardless of source development language. The CRL handles memory management, provides a secure runtime environment, ensures object location transparency and provides concurrency management. C# code runs in a managed execution environment, which is the most important technological step to making C# run on different operating systems. However, some of the .NET libraries are based on Windows, particularly the `WinForms` library which depends on the nitty-gritty details of the Windows API. There is a project to port the Windows API to Unix systems, but this isn't here now and Microsoft have not given any firm indication of their intentions in this area.

3.5 Language Interoperability

C# has strength in language interoperability, any language targeted to the CLR in Visual

Studio .NET can use, subclass, and call functions only on managed CLR classes built in other languages. C# demands the standardization of code so that the code written in one language can be reused in another language. While this is possible, it is no doubt more awkward in Java, as in other programming languages not supported yet in .NET. Java, by contrast, was one programming language and execution machinery for that one programming language.

It is very straightforward to create Java code for a Windows-based operating system that can access COM objects. However, CLR would be language neutral, that it would support multiple programming languages, has interoperability with DLLs, COM, OLE automation—all of these technologies that any piece of code was written in before .NET. Using Visual Studio .NET, libraries can be easily built in J#, Visual Basic .NET and managed C++ (with other languages to come) and subclassed or directly used in C#. Just looking at the libraries available in the on-line documentation, the developer has no idea whether the classes were built in C#, C++ or another.

Both the Java Virtual Machine and the Common Language Runtime allow programmers to write code in many different languages, so long as they compile to byte code or IL (Intermediated Language) code respectively. However, the .NET platform has done much more than just allows other languages to be compiled to IL code. NET allows multiple languages to freely share and extend each others libraries to a great extent. For instance, an Eiffel or Visual Basic programmer could import a C# class, override a virtual method of that class. With VB.NET, the VB language finally completes the transition to a fully OO language.

Microsoft is opening up a channel to developers in other programming languages, because .NET is enfranchising Perl, Eiffel, COBOL, and other programmers by allowing cross-language component interactions. By using XML and SOAP in their component messaging layer, it is opening up a channel to non-.NET components. Languages written for .NET will generally plug into the Visual Studio.NET environment and use the same RAD (Rapid Application Development) frameworks if needed, thus overcoming the "second class citizen" effect of using another language.

3.6 Threading

Threads are a powerful abstraction for allowing parallelized operations: graphical updates can happen while another thread is performing computations, two threads can handle two simultaneous network requests from a single process, and so on. Java and C# threading capabilities are quite different on the syntax and usage.

3.6.1 Create and Run a Thread

Java provides most of its threading functionality in the `java.lang.Thread` and `java.lang.Runnable` classes. Creating a thread is as simple as extending the `Thread` class and calling `start();` a `Thread` may also be defined by authoring a class that implements `Runnable` and having that class passed into a `Thread`.

Where Java allows the `java.lang.Thread` class to be extended and the `java.lang.Runnable` interface to be implemented, C# does not provide these facilities. In C#, a thread can be created using `System.Threading` namespace which contains two classes: `Thread` and `ThreadStart`. The `ThreadStart` method has

both its parameters and return value as void -- this simply means that instead of using the inner class pattern, an object will need to be created and one of the object's methods must be passed to the thread for execution.

Java code:

```
// Thread class extends the java.lang.Runnable interface
class SortPanel extends javax.swing.JPanel implements Runnable {
    //declaration of a thread
    private java.lang.Thread animateTread = new Thread(this);

    // run() method is the thread's running behavior
    public final void run() {
        // make sure the current thread is the one we want
        while (Thread.currentThread() == animateTread) {
        try {
            if (!stopRequest && sortAlgorithm != null) {
                sortAlgorithm.sort(dataArray);
                stopRequest = true;
                System.out.print( "\n" + "result  ");
                for(int i=0;i<dataArray.length;++i)
                    System.out.print(dataArray[i]+" ");
                System.out.print("\n");
            }
            // the tread sleeps for one second (1000 milliseconds)
           Thread.sleep(1000);
            } catch ( InterruptedException e) {
                System.out.println("Thread was interrupted in SortPanel.run()");
            } //end of catch
        }//end of while
    }
}
```

C# code:

```
class SortControl {
    // Decalration of a thread
    private Thread t;
    private void ClickEvent(object sender, System.EventArgs e)
    {
        // Instantiate the thread, PerformSort() method is the thread running behavior
        t = new Thread(new ThreadStart(PerformSort));
        // Start the thread
        t.Start();
    }
    public void PerformSort()
    {
        // make sure the current thread is the one we want
        while (Thread.CurrentThread == t)
        {
            try
            {
                if (!stopRequest && sortAlgorithm != null)
                {
                    sortAlgorithm.sort(dataArray);
                    stopRequest = true;
                    Console.WriteLine( "\n" + "result  ");
                    for(int i=0;i<dataArray.Length;++i)
                        Console.WriteLine(dataArray[i]+" ");
                    Console.WriteLine("\n");
                }
            // the thread sleeps for one second, the same method as in Java
                Thread.Sleep(1000);
            } catch ( ThreadInterruptedException e) {
            Console.WriteLine("Thread was interrupted in SortPanel.PerformStart()");
            }
        }
} }
```

3.6.2 Thread Usage

There are lots of standard operations that a thread can perform: test if the thread is alive, what is the current thread, put a thread into sleep and kill a thread, etc. The following table describes the mapping between Java's *java.lang.Thread* methods and C#'s *System.Threading.Thread* object.

Java	C#	Description
isAlive() method	*IsAlive* get property	Returns true if the thread is alive
interrupt() method	*Interrupt()* method	In Java, this sets the interrupted status of the thread and can be checked to see whether a thread has been interrupted; in C#, it returns to the started state if the sleeping or waiting thread's Interrupted method is called by another thread in the program
isInterrupted() method	N/A	Returns true is this thread has been interrupted
sleep() method	*Sleep()* method	A method paused the thread of execution for a given amount of time or until the thread is interrupted
join() method	*Join()* method	Java join method is simply timeout; in C#, it returns a boolean upon termination to signify whether the thread died(true) or the timeout expired(false)
suspend() method	*Suspend()* method	Calls the running thread into a suspend status
resume() method	*Resume()* method	Resumes a suspended thread

Table 3.1: Comparison of Thread Usage Methods in Java and C#

Java code:

```
class SortPanel extends javax.swing.JPanel implements Runnable {
    // Create the thread and instantiate it
     private java.lang.Thread animateTread = new Thread(this);
    // start the thread
     public void start()
     {
          stopRequest = false;
          // Starts thread that controls the timing of the sorting
          if ((animateTread == null) || (!animateTread.isAlive())) {
               animateTread = new Thread(this);
          }
          // Test whether the thread is alive
          if (!animateTread.isAlive()) {
               animateTread.start();
          }
     }
}
```

C# code:

```
class SortControl {
     // Create a thread
     private System.Threading.Thread animateThread;
     public void start() {
          stopRequest = false;
          // Instantiate the thread
          animateThread = new Thread(new ThreadStart(PerformSort));
          // Test whether the thread is alive
          if (!animateThread.IsAlive){
               // Start the thread
               animateThread.Start();
          }
     }
}
```

3.6.3 Stop a Thread

In Java, `Thread.suspend()` and `Thread.stop()` provide asynchronous methods of stopping a thread. However, these methods put the running program into an inconsistent state. Using them often results in deadlocks and incorrect resource cleanup. Upon a `stop()` method call, an unchecked `java.lang.ThreadDeath` error will propagate up the running thread's stack, unlocking any locked monitors as it goes along. [5] The proper way to stop a running thread in Java is to set a variable that the thread checks occasionally. For a thread to terminate cleanly the `run()` method must complete; therefore, when the thread detects that the variable is set, it should return from the `run()` method.

In C#, a thread is destroyed with a call to the `Thread.Abort()` method, which begins the process of terminating the thread. Once the thread terminates, it cannot be restarted by calling the function `Thread.Start()` again. The runtime system forces a thread to abort by throwing an uncatchable `System.Threading.ThreadAbortException`. The `Thread.Abort()` method lets the system quietly stop the thread without informing the user. As this method does not say that the thread will abort immediately, hence to be sure that the thread has terminated, `Thread.Join()` can be called to wait on the thread. Join is a blocking call that does not return until the thread has actually stopped executing. [3]

3.6.4 Thread Synchronization

Both languages allow developers to acquire a lock to coordinate the actions of the two threads in a certain behavior on any reference object—once the lock is acquired, the

program block can be assured that it is the only block that has acquired it. Also by using that lock, a program can wait until a signal comes through the variable causing the thread to wake up.

Java	C#	Description
Synchronized	*Lock*	C# provides the lock statement which is semantically identical to the synchronized statement in Java
Object.wait()	*Monitor.Wait*(object obj)	In C#, to wait for a signal, the *System.Threading.Monitor()* class needs to be used instead of a wait method inherent to an Object in Java. Both of these methods need to be executed in synchronized blocks.
Object.notify()	*Monitor.Pulse*(object obj)	The same concept as *Monitor.Wait()*

Table 3.2: Comparison of Thread Synchronization Methods in Java and C#

Java code:

```
public static Object synchronizeObj = "locking variable";
public static void count() {
    synchronized( synchronizeObj ) {
        for( int count=1;count<=5;count++ ) {
            System.out.print( count + " " );
```

```
                synchronizeObj.notifyAll();
                if( count < 5 )
                        try {
                                synchronizeObj.wait();
                        } catch( InterruptedException error ) { }
                }
        }
}
```

C# code:

```
public static Object synchronizeObj = "locking variable";
public static void Count() {
    lock( synchronizeObj ) {
        for( int count=1;count<=5;count++ ) {
            Console.WriteLine( count + " " );nbsp;
            Monitor.PulseAll( synchronizeObj );
            if( count < 5 )
                Monitor.Wait( synchronizeObj );
        }
    }
}
```

3.7 Nested Classes

Java supports both nested classes and inner classes. A nested class is syntactically nested in another but otherwise has no special relationship. In Java, nested classes are declared by using the `static` keyword on a class declaration that is syntactically nested in a class declaration. When a class declaration that is lexically nested in a class declaration omits the `static` keyword, the class is called inner class. An inner class takes an implicit `this`

pointer from the outer class, which means it can access the instance members of the outer class using *this*.

C# only has nested classes, which are more like C++ than Java. C#'s nested classes have few limitations than Java's inner classes. In fact, C#'s nested classes are akin to nested classes of Java. The methods of a nested class may access all the members (fields or methods) of the nested class but they can access only static members (fields or methods) of the outer class. Private members of outer class are accessible from the nested class but the enclosing instance must be passed to the nested class.

3.8 Access Modifiers

C# has all the access modifiers as Java has (*private, public, protected*). *Private* and *public* have the same meanings as in Java, noting that in C# the default access level is *private*. However, the C# *protected* access modifier has the same semantics as the C++ version, and the scope of the *protected* is slightly different from that of the *protected* access modifier in Java. A protected entity (method or data) can only be accessed by the same class or its sub classes. The *internal* access modifier in C# is equivalent to the *protected* access modifier in Java. By default access modifiers of all methods in C# are *private* but that of Java are *protected*. C# also has the *internal protected* access modifier, which means that a member can be accessed from classes that are in the same assembly or from derived classes. [9]

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

Table 3.3: Comparison of Access Modifers in Java and C#

3.9 Serialization

Object serialization is the process of rendering an object into a state that can be stored persistently. In Java, for an object to be serialized, it must be an instance of a class that implements either the `java.io.Serializable` or `java.io.Externalizable` interface. [5] Java classes are not serializable by default and are required to implement the `java.io.Serializable` interface before they can be persisted. C# classes require annotation with the `[Serializable]` attribute (attributes will be discussed in the next chapter) before they can be processed by the formatter which is an object handles message serialization; the classes are marked as serializable with that `[Serializable]` compiler attribute, which is similar in spirit to the Java implements `java.io.Serializable`. Serialization of object can also be customized using the `System.Runtime.Serialization.ISerializable` interface.

The `transient` keyword in Java is used to specify members that are not to be serialized. Serializable objects in C# are specified with the `[Serializable]` attribute and

members that should not be serialized are specified using the *[Nonserialized]* attribute. The default serializable format in Java is binary, but it allows customization of format. In C#, however, there are two formatters control the way the serialized data is stored: *BinaryFormatter* and *SoapFormatter*, which produce two serializable formats: Binary format and XML. [3]

3.10 Documentation

Documentation is a key resource in development on any platform. Java's javadoc is the tool with standard format makes it easy to produce excellent HTML documentation of APIs (Application Program Interfaces). C# implements a code documentation has similar functionality to the *javadoc* utility. [9] Like *javadoc*, this works only if the developer places markup in appropriately formatted code comments. Unlike *javadoc*, this markup is extracted at compile time and is output to a separated file as XML. Being XML, this information can be easily processed to produce documentation in different formats---not just HTML, but any format that is supported by a valid XSLT (Extensible Stylesheet Language Transformation). The most common approach, however, is to utilize an XSLT to generate HTML tags.

3.11 Importing Libraries

A language is pretty much useless without libraries. Java and C# both define an extensive set of standard libraries that implement critical functionality, in particular: multithreading, locking, interprocess communication, and network access. However, a Java developer knows that Java provides built in class libraries or packages support through which

developers can achieves their task more efficiently. C# does not have its own class libraries, but it shares the .NET framework class libraries known as namespaces that can be used in other .NET languages such as VB.NET or JScript.NET. The .NET framework class library provides a set of classes that provide essential functionality for applications build within the .NET environment. Web functionality, XML support, database support, threading and distributed computing support is provided by the .NET framework class library. .NET allows an entire family of programming languages such as C# and VB.NET to share the same library functions, the same web functionality and the same database access methodology.

3.12 Summary

In this chapter, some different features in Java and C# are presented. Java has the strength of platform interoperability and C# has the strength of language interoperability. C# has more data types and access modifiers than that in Java, and supports boarder data types in `switch` statement. The threading in Java and C# has differences with syntax and implementation. C# only supports nested class while Java supports nested classes and inner classes. In C#, objects have `[Seriliazable]` and `[Nonseriliazed]` attributes. C# has code documentation tool which outputs a XML file. C# doesn't have its own library like Java but it shares the .NET framework class library with other .NET languages.

Chapter 4 The Features in C# Beyond Java

C# is a modern and innovative programming language, one which carefully incorporates features found in the most common industry and research languages. It has many useful features as Java does; in addition, many new features which are completely new to Java have been added into that language.

4.1 Enumeration

In C#, an enumeration (enum) is a special form of value type, which inherits from *System.Enum* and supplies alternate names for the values of an underlying primitive type. An enumeration type has a name, an underlying type, and a set of fields. The members of an enumeration are the constants declared in the enumeration and the members inherited from class object.

Although Java can add *static final* variables to interfaces, they don't have much other functionality than their values. In contrast, C# can convert *enums* between their strongly typed *enum* value, their underlying integer type, or a string representation of the *enum* itself.

Using enumeration types is the best way to store data such as days, months, account types, genders, and so on. In the following example, an enumeration, Days, is declared. Two enumerators are explicitly converted to *int* and assigned to *int* variables.

C# code:

```
using System;
public class EnumTest
{
    enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
  public static void Main()
  {
    int x = (int) Days.Sun;
    int y = (int) Days.Fri;
    Console.WriteLine("Sun = {0}", x);
    Console.WriteLine("Fri = {0}", y);
  }
}
```

output:

```
Sun = 2
Fri = 7
```

Notice that if removes the initializer from Sat=1, the result will be:

```
Sun = 1
Fri = 6
```

4.2 For-each statement

Borrowed from Visual Basic, the C# `foreach` loop is a less verbose way to iterate through an array or collection class that implements the `System.Collections.IEumerable` interface.

C# code:

```
using System;
using Threading;
```

```
foreach (Thread t in threadHolder.Values)
{
        if ( t != null && t.IsAlive)
        t.Abort();
}
```

C# 2.0 adds a new semantic to the *foreach* keyword that makes writing the iterators much more concise. The original version of Iterators required a foreach method be declared in the class. This syntax limited iterators to one per class and did not support arguments. The new syntax uses the same function syntax as other methods, allows for multiple types of iterators per class and also for any number of function arguments. The only requirement is that a *yield* instruction exists in the function and that the function returns *IEnumerator*, *IEnumerable* or generic versions. The default iterator used for *foreach* must be given the name *GetEnumerator*. The *yield* syntax has been updated to yield return value to return the next value or yield break to indicate completion. [12]

C# code:

```
public class List
{
     public object foreach ()
     {
      int i=0;
     while (i < theList.Length ())
    yield theList[i++];
      }
}
```

4.3 Jump Statement---Goto

In Java `goto` is an unimplemented keyword which cannot be used in Java progeams. C# supports `goto` statement; it's possible to jump to any position inside the program with help from `goto`. It transfers control to a labeled statement. The label must exist and must be in the scope of the `goto` statement. More than one *goto* statement can transfer control to the same label.

C# code:

```
using System;
class  Hello
{
   public static void Main()
   {
      Console.WriteLine("DemoGoto");
      goto nobodyknows;
      Console.WriteLine("End of DemoGoto");
nobodyknows:
      Console.WriteLine("Goto jumps to here");
   }
}
```

The `goto` statement can transfer control out of a block, but it can never transfer control into a block. The purpose of this restriction is to avoid the possibility of jumping past an initialization. The same rule exists in C++ and other languages as well.

The `goto` statement and the targeted label statement can be very far apart in the code. This distance can easily obscure the control-flow logic, also makes the programs complex and

painful to maintain. Therefore, most programming guidelines recommend that avoiding use `goto` statements.

4.4 Delegates and Events

A delegate in C# allows programmers to pass methods of one class to objects of other classes that can call those methods. This concept is familiar to C++ developers who have used function pointers to pass functions as parameters to other methods in the same class or in another class. Delegates are a type introduced by C# that has no direct analogue in Java.

Delegates provide an object-oriented type-safe mechanism for passing method references as parameters without using function pointers. Delegates are primarily used for event handling and asynchronous callbacks. C#'s delegate is used where Java developers would use an interface with a single method. A delegate instance encapsulates a static or an instance method. The use of appropriate design patterns and interfaces in Java can provide equivalent capabilities; however, delegates are an elegant and powerful feature.

The design of delegates in C# makes simplifies for event creation and handling. Event-driven programming is referred to as the publish-subscribe design pattern and also known as Observer/Observable to Java programmers. Events are published and interested parties can subscribe to a given event. An event listener registers for some event with an event broadcaster, and when something happens, the broadcaster "fires" an event to all listeners of the event. This event model not only makes it easier to both trigger and responds to events, but also it uses a system that makes it a lot less likely to introduce bugs when setting up a messaging system.

Java event handling adheres to this publish-subscribe approach, so an event can be published to all the subscribers of that event. When an event occurs, the parties that have subscribed are notified. The publisher has register or unregister methods which are used to register/unregister subscribes. Usually the subscribe implements an event interface and can add itself to the particular event and take appropriate action on its occurrence (which is conveyed to all the subscribers simultaneously).

C# uses delegates to provide an explicit mechanism for creating a publish-subscribe model to handle events. Delegates are a general-purpose mechanism for calling methods indirectly and their principal uses in .NET framework are for implementing events and call-back methods. An event is a subclass of the `System.EventArgs` class which inherits all its methods from `System.Object`. In C#, any object can publish a set of events to which other classes can subscribe. When the publishing class raises an event, all the subscribed classes are notified. The publisher and the subscribers are decoupled by the delegates. The delegates are passed as parameters of the protected methods which are invoked when an event occurs. The subscribe is a method subscribe to the event that accept the delegate parameters and returns the same type as the even delegate. The event handler usually returns void and take two parameters: the first is the source of the event which is the publishing object; the second parameter is an object derived from `System.EventArgs` subclass. The C# delegates are implemented in the .NET framework as a class derived from `System.Delegate`. An event handler in C# is a delegate with a special signature, which is given below:

```
public delegate void MyEventHandler(object sender, MyEventArgs e);
```

C# has class members called events that accept one or more delegates. The `event` keyword is used in the declaration of methods that are to be the delegates called when an event occurs. When an event is raised through the event variable the target method is called through a predefined delegate. Since publisher and the subscribers are decoupled by the delegate which makes for more flexible and robust code. In contrast, Java doesn't have an event class member, nor does it have delegates. It commonly implements event behavior via inner classes and methods.

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);
    }
}
```

C# code:

```
using System;
```

```
class EvenNumberEvent: EventArgs{
    // 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 100 random numbers between 1 and 200 then causes and
        //event to occur if the current number is even.
        public void scramble (){
            Random r = new Random((int) DateTime.Now.Ticks);
             for(int i=0; i < 100; i++){
                int current = (int) r.Next(200);
                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. scramble();
        //unregister the callback/subscriber
        pub.EvenNumHandler -= new
        Publisher.EvenNumberSeenHandler(EventHandler);
        }
}
```

4.5 Properties

Properties are named members of classes, structs and interfaces. [11] In C#, properties are natural extension of data fields. Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. In Java, this is often accomplished by programs implementing getter and setter methods. C# property is similar to Visual Basic property. C# property is a built in mechanism which has accessors that specify the statements to access the property just like it was a field. The accessor of a property contains the executable statements associated with getting (reading or computering) or setting (writing) the property. [11] A property should have at least one accessor, either set or get. The set accessor has a free variable available in it called value, which gets created automatically by the compiler. It is illegal to use the implicit parameter name (value) for a local variable declaration in a set accessor. Since normal data fields and properties are stored in the same memory space, in C#, it is not possible to declare a field and property with the same name.

Java code:

```
public class SortPanel extends JPanel{
```

```
    private boolean stopRequest;
    public void setStopRequest(boolean stop)
    {
            stopRequest = stop;
    }
    public boolean getStopRequest()
    {
            return stopRequest;
    }

}
```

In C#, properties are defined using property declaration syntax. The first part of the syntax looks quite similar to a field declaration. The second part includes a get accessor and/or a set accessor. The code would be more like:

```
public class SortControl:UserControl{
    protected static bool stopRequest = true;
    public bool stopRequest
    {
            get
            {   return stopRequest;   }
            set
            {   stopRequest = value;   }
    }
}
```

Properties that can be read and written, like `stopRequest` in `SortControl` class, including both get and set accessors. The get accessor is called when the property's value is read; the set accessor is called when the property's value is written. In a set accessor, the new value for the property is given in an implicit value parameter. It is then possible to just

assign to `stopRequest`. This is a tiny syntactic change from Java code, but it makes the implementation more elegant.

A property can be read and written in the same way that fields can be read and written. For example, the following code instantiates the `SortControl` class and writes and reads its `stopRequest` property.

C# code:

```
SortControl sortControl = new SortControl();
sortControl.stopRequest = true; // set
bool stop = sortControl.stopRequest; // get
```

4.6 Indexer

One of C#'s most interesting features is the class indexer. An indexer is a special kind of property that makes it possible to access value in a class with array like syntax. Obviously, this ability becomes useful when creating a collection class, but giving a class array-like behavior can be useful in other situations, such as when dealing with a large file or abstracting a set of finite resources. Here's a sample class with an indexer that returns a string:

C# code:

```
public class ListBox: Control
{
        private string[] items;
        //indexer that indexes by number
        public string this[int index] {
        get
        {
```

```
            return items[index];
        }
        set
        {
            items[index] = value;
        Repaint();
        }
    }
}
```

Notice that the property name is `this`, which refers back to the current instance of the class, and that the parameter list is enclosed in square brackets instead of parentheses. When defining an indexer, the parameters may be of any type, although `int` usually makes the most sense. It's also possible to have more than one indexer in the same class.

4.7 Boxing and Unboxing

Boxing and unboxing is an essential concept in C#'s type system. It enables a unified view of the type system where a value of any type can be treated as an object. Conversion of a value type to a reference type is called Boxing. It is possible that any value types such as `int`, `structs` or enumerations are compatible with `object`. When a variable of a value type needs to be converted to a reference type, an object box is allocated to hold the value, and the value is copied into the box.

C# code:

```
class BoxingExample
{
    static void Main() {
    int i = 1;
```

```
        //boxing
        //which automatically wraps up the value 3 in a heap object.
        object obj = i;
    }
}
```

The runtime system implements boxing by instantiating a container object of the appropriate type and copying the data from the value type into it. It's important to understand that the boxed instance contains a copy of the source value. Any changes made to the original value are not reflected in the boxed instance.

Unboxing is the reverse of boxing. An unboxing conversion permits an explicit conversion from type object to any value-type or from any interface-type to any value-type that implements the interface-type. It takes an object representing a previously boxed value and re-creates a value type from it. The runtime system checks that the boxed instance is being unboxed as the correct type. If the source argument is null or a reference to an incompatible object, it throws a *System.InvalidCastException*. It isn't possible to create a value type representation of any reference type using unboxing; unboxing works only on objects that contains previously boxed values.

C# code:

```
class UnboxingExample
{
    static void Main() {
    int i = 1;
    //boxing
    //which automatically wraps up the value 1 in a heap object.
    object obj = i;
```

```
        //unboxing a previous boxed value
        //Automatically unwraps the wrapped int value.
        int j = (int) obj;
    }
}
```

With boxing and unboxing, primitive types can be used just as efficiently as any primitive type in any language, hence, this primitive type can be converted to an object automatically when it needs to be treated like an object. This type system unification provides value types with the benefits of object-ness without introducing unnecessary overhead.

4.8 Struct

A `struct` is very similar to a class. A struct can implement interfaces and have the same kinds of members as a class, but a struct does not support inheritance. Unlike class, a struct cannot have a constructor nor have a destructor. While a class is created in the heap as a reference type, a struct is a value type that is stored on the stack. Therefore, used with care, structs are faster than classes. It is helpful to view C# structs as a construct which makes the C# type system work elegantly.

However, simply replacing a class with a struct can be disastrous. Since a struct is passed by value, a "fat" struct is slower to pass around because values must be copied to a new place. In the case of a class, only the reference to the class is passed around. The following is an example of a struct. Note how similar it is with a class. Substituting the word `class` for `struct`, it becomes a class.

C# code:

```
struct Point
{
        public int x, y;
        public Point(int x, int y) {
        this.x = x;
        this.y = y;
        }
}
```

C# simply allows programmers to extend the primitive set of types built in to the language. In fact, C# implements all the primitive types as structs. For instance, the `int` type merely aliases `System.Int32` struct, the `long` type aliases `System.Int64` struct. These primitive types are of course able to be treated specially by the compiler, but the language itself does not make such a distinction.

4.9 Operator Overloading

Unlike Java, C# has the useful feature that various operators can be overloaded. Operator overloading allows programmers to build types which feel as natural to use as simple types. Operator overloading is pretty useful concept derived from C++ by c#, but C# implements a stricter version of operator overloading in C++.

Operator overloading provides a way to define and use operators such as +, -, and / for user-defined classes or structs. It allows define/redefine the way operators work with classes and structs. This allows programmers to make their custom types look and feel like simple types such as `int` and `String`. It consists of nothing more than a method declared

by the keyword operator and followed by an operator. These method called operator functions and must be public and static. They can take only value arguments; the `ref` and `out` parameters are not allowed as arguments to operator functions.

There are three types of overloadable operators called unary, binary, and conversion. It is impossible that all operators of each type can be overloaded. Following is a table illustrating these operators along with their overloadabilities.

Operators	Overloadability
+, -, *, /, %, &, |, <<, >>	All C# binary operators can be overloaded.
+, -, !, ~, ++, --	All C# unary operators can be overloaded.
==, !=, <, >, <= , >=	All relational operators can be overloaded
&&, ||	They can't be overloaded.
[] (Array index operator)	They can't be overloaded.
( ) (Conversion operator)	They can't be overloaded.
+=, -=, *=, /=, %=	These compound assignment operators can be overloaded. However in C#, these operators are automatically overloaded when the respective binary operator is overloaded.
=, . , ?:, ->, new, is, as, sizeof	These operators can't be overloaded in C#.

Table 4.1: Illustration of Operators Overloadbility

4.9.1 Overloading Unary Operators

Unary operators are those that require only a single operand/parameter for the operation.

The class or struct involved in the operation must contain the operator declaration. Unary operators are +, -, !, ~, ++, --. When overloading the unary operators, +, -, !, or ~ must take a parameter of the defining type and can return any type; ++ or - must take and return the defining type.

C# code:

```
class Complex
{
    private int realPart;
    private int imgPart;
    public Complex()
    { }
    public Complex(int i, int j)
    {
        realPart = i;
        imgPart = j;
    }
    public static Complex operator -(Complex c)
    {
        Complex temp = new Complex();
        temp.realPart = -c.realPart;
        temp.imgPart = -c.imgPart;
        return temp;
    }
}
```

4.9.2 Overloading Binary Operators

Binary operators are those that require two operands/parameters for the operation. One of the parameters has to be of a type in which the operator is declared. They include +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, and <=.

C# code:

```
class Complex
{
    private int realPart;
    private int imgPart;
    public Complex()
    {
    }

    public Complex(int i, int j)
    {
     realPart = i;
     imgPart = j;
 }
    public static Complex operator +(Complex c1, Complex c2)
    {
        Complex temp = new Complex();
        temp.realPart = c1.realPart+c2.realPart;
        temp.imgPart = c1.imgPart+c2.imgPart;
        return temp;
    }
}
```

4.9.3 Overloading Conversion Operators

Conversion operators are those that involve converting from one data type to another through assignment. There are implicit and explicit conversions. Implicit conversions are those that involve direct assignment of one type to another. Explicit conversions are conversions that require one type to be casted as another type in order to perform the conversion. Conversions that may cause exceptions or result in loss of data as the type is converted should be handled as explicit conversions.

C# code:

```
//example of overloading explicit conversion operator
string dbFileType = "C";    // Assume this is from a database
FileType fileType = new FileType();
fileType = (FileType)dbFileType;
```

A few rules greatly simplify the design and use of operator overloading: each operator declaration must be *Public* and *Shared*, and must be declared in the same type as one of its operands. The operator overload methods can be overloaded just like any other methods in C#, but it can't return void. The overloaded methods should differ in their type of arguments and/or number of arguments and/or order of arguments. The operator overloading is one of the key concepts of C# and is a very interesting aspect of the language.

4.10 Pointers

In the C++ world, the use of pointers is an integral part of the language. This gives C++ a tremendous amount of power and flexibility, which are both a strength and a weakness. It's a strength because C++ gives programmers lots of freedom to do exactly what they want, but the same freedom also makes it easy to write code with subtle bugs.

Java, on the other hand, views pointers as something that is too much trouble to deal with and hides any direct pointer manipulation in its language and the programmer doesn't have the power to access or manipulate pointer references. This means that programmers never have to deal with any of the bugs that are common in code involving pointers. However, programmers still have to be aware that some parameters are passed by reference (any object variable) and other parameters are passed by value (any primitive type).

C# takes a view of pointers that's between the C++ and Java views. For the majority of the code that people write, the disadvantages of pointers outweigh the advantages; C# doesn't support pointers when producing managed code. From a pragmatic viewpoint, however, there are some cases where pointers are very useful, so C# provides a way to bypass the safety net when pointer use is required. That gives developers the best of both worlds—the safety that the Java model provides, along with most of the flexibility of the C++ model when it is needed.

A pointer variable holds a memory address that references another data item. Pointers are similar to Java and C# reference types except that pointers give the developer access to both the memory address referenced by the pointer and the data at that address. Pointers are unique in that they do not derive from `System.Object`. A major problem with using pointers in C# is that there is a background automatic garbage collection process in operation. Trying to free memory, garbage collection could change the memory location of a current object without the programmer's knowledge. So any pointer which previously pointed to that object will no longer do so. This could compromise the running of the C# program and could affect the integrity of other programs. Therefore the code using pointers has to be explicitly marked by the programmer as `unsafe`, which can deal directly with pointer types, and fix objects to temporarily prevent the garbage collector from moving them. This `unsafe` code feature is in fact a "safe" feature from the perspective of both programmers and users. Unsafe code must be clearly marked in the code with the modifier unsafe, so programmers can't possibly use unsafe features accidentally, and the C# compiler and the execution engine work together to ensure that unsafe code cannot masquerade as safe code.

C# code:

```
unsafe{
    //Declare and initialize an int variable
    int myInt = 100;
    //Declare a pointer to an int
    int* myPtr;
    //Use address-of (&) operator to get the address of myInt and assign to myPtr
    myPtr = &myInt;
    //Use indirection (*) to get value of the int referenced by myPtr
    System.Console.WriteLine("value of a = " + *myPtr);
    //Use indirection to set value of int pointed to by myPtr
    *myPtr = 300;
}
```

output:

```
value of a = 500;
```

Unsafe code is used in C# when speed is extremely important or when the object needs to interface within existing software, such as COM objects or native C code in DLLs.

4.11 Attributes

An attribute is a powerful C# language feature that is attached to a target programming element to customize behaviors or extract organizational information of the target at design, compile, or runtime. There are two types of attributes: *intrinsic* and *custom*. Intrinsic attributes are those provided by the .NET Framework. Serializable is an example of intrinsic attribute. It's clear that custom attributes are very useful, but missed in Java features. For a Java developer new to C#, attributes may well be one of the hardest language features to appreciate; however, while attributes may seem like a relatively insignificant element of C# grammar, they can dramatically ease the development.

Attributes provide a powerful method of associating declarative information with C# code (types, methods, properties, and so forth). Once associated with a program entity, the attribute can be queried at run time and used in any number of ways.

Custom attribute can be created by defining an attribute class, a class that derives directly or indirectly from `System.Attribute`. Custom attributes can be used to store information that can be stored at compile time and retrieved at run time. This information can be related to any aspect/feature of target element depending upon the design of an application. There are four sections of any custom attribute `AttributeUsage`, Class declaration, Constructor and Properties.

Attribute usage is defined by `System.AttributeUsageAttribute`, which is another attribute. `AttributeUsage` has three members; `AttributeTarget`, `Inherited` and `AllowMultiple`. `AttributeTarget` specifies the target elements to which custom attribute can be applied.

Target name	Description
assembly	The attribute applies to the entire assembly.
module	The attribute applies to the entire module.
return	The attribute applies to the return value of the method, property set branch or indexer set branch.
value	The attribute applies to the implicit `value` parameter of the property or indexer.

Table 4.2: Attributes targets and their description

An attribute class is declared as any other class in C#, but it must be declared as public classes. By convention, the name of the attribute class ends with the word Attribute. Constructors are needed for custom attribute class. Constructors can have required and optional parameters the same way as in a regular class. When required Properties should be declared, for access and setting the information (as per design requirement) for the attribute class. The following code is an example of using an attributes to provide information about the name of a developer and then using property to access the information.

C# code:

```
using System;
namespace CustomCode
{
   [CodeInfo("Concordia")]
   class CustomAttrDemo
   {
         static void Main(string[] args)
         {
             CustomAttrDemo cattDemo = new CustomAttrDemo();
             DisplayCustomAttribute(cattDemo);
         }
         static void DisplayCustomAttribute(CustomAttrDemo cattDemo)
         {
             Type type = cattDemo.GetType();
             Object obj = type.GetCustomAttributes(false)[0];
             if(obj is CodeInfoAttribute)
             {
                 System.Console.Write("Developer - ");
                 System.Console.WriteLine(((CodeInfoAttribute)obj).Devloper);
             }
```

```
            else
            System.Console.WriteLine("Attribute not found");
        }
    }
    // define the AttributeUsage for CodeInfoAttribute class
    [AttributeUsage(AttributeTarget.All,Inherited=true,AllowMultiple=true)]
    // custom attribute class
    public class CodeInfoAttribute : Attribute
    {
        // Private Data
        private string developerName;
        // Constructor
        public CodeInfoAttribute(string developerName)
        {
        this.developer_name = developerName;
        }
        //Declare a property to get developer's name
        public string Devloper
        {
        get
        { return developerName; }
        }
    }
}
```

4.12 Rich Parameter Passing

Compared to Java, C# has much richer syntax for expressing parameter passing, including the ability to specify that a parameter is in, `out` or `ref`. By default, a parameter passed into a method in C# is passed by value. However, if the value of a parameter can be changed inside a method, then it should be marked as `ref`. This allows any assignments that occur inside a method to be reflected outside the method after the method call has

terminated. The `ref` parameters must have been initialized before the method is executed. A parameter marked as out is exactly the same as a `ref` parameter, but does not require the parameter to have been initialized before the method is called. Such a rich syntax is in line with other high level languages and therefore simplifies the integration of C# with these other environments. They also represent best practice software engineering principles.

4.13 Summary

In this chapter, some features that lack in Java but are present in C# has been discussed. Enumeration, delegates, properties, indexer, boxing, unboxing, operator overloading, rich parameter passing are all powerful features in C# language. For-each, goto statement and struct are also useful while developing. Pointers marked as `unsafe` code because programmers need directly access the memory.

Chapter 5 GUI Comparison---Swing vs Windows Form

So far, what I discussed is all language level of these two languages. I also write applications building in both Java and C#: an animation of quicksort algorithm. I use Borland JBuilder 6.0 for creating Java applet and Visual Studio .NET Beta 1 for creating C# windows application. Although while writing the code, I have the feeling that no big difference from the language syntax; however, no doubt there are some differences about the GUI part.

Swing is more powerful than Windows Form in some ways. For example, border styles can be plugged into any component using the strategy design pattern. In Windows Forms, components are responsible for supplying and drawing their own borders, but not all components support borders, and some only supply a few borders. This is because Windows form is a thin layer about native windows control and doesn't seem to have any improvements over WFC in J++. Moreover, there is no LayoutManager in C#, and all the components are aligned to a fixed coordinate system which is relative to the top left of the Form control. Although this enables programmers to have fine grain control over the exact placement of their controls, but it comes very difficult for any programmer to imagine the exact position where the component should be placed during design time. Furthermore, since Swing used the model-view-controller (MVC) design pattern, all the components use separated data models which can all be extended. All the layout and rendering elements also can be extended and overridden. None of this true for Windows Forms.

However, Windows Form for sure has some advantages over Swing. For Swing, speed used to be a problem, although it has become a minor issue latterly. Windows Form is much faster than Swing. In addition, Event Handlers are able to be added by clicking buttons and selecting menu items in Visual Studio .NET. For example, instead of complex inner classes in Java, C# Windows Forms uses a VB-like syntax for event handler signatures.

C# code:

```
private void ClickEvent(object sender, System.EventArgs e)
{
    //if the Create button is clicked
    if (sender == cmdCreate)
    {
        this.sortControl.Show();
        dataArray.Initialize();
        scramble();
        initSort();
    }
    //if the Sort button is clicked
    if (sender == cmdSort)
    {

        sortControl = new SortControl(dataArray);
        sortControl.StopRequest = false;
        sortControl.PausePeriod = pausePeriodMilliSeconds;
        sortAlgorithm = (SortAlgorithm) new QSortAlgorithm();
        sortControl.setSortAlgorithm(sortAlgorithm);
        sortControl.start();

    }
    else
```

```
        //if the Exit button is clicked
        if (sender == cmdStop)
        {
            sortControl.stop();
            foreach(Thread t in threadHolder.Values)
            {
                if(t != null && t.IsAlive)
                t.Abort();
            }
            Application.Exit();
        }
}
```

A nice thing to notice is that operator overload is used to add event handlers with .NET showing in the tool-generated code which makes the code more elegant than inner class in Java:

```
this.cmdSort.Click += new System.EventHandler(this.ClickEvent);
```

Moreover, Windows Form has powerful support for DataBinding and XML, although I did not have the chance to touch these two parts in my application. Data binding provides a way for developers to create a read/write link between the controls on a form and the data in their application. Clearly, data binding was used within applications to take advantage of data stored in databases. Windows Forms data binding allows programmers to access data from databases as well as data in other structures, such as arrays and collections. A new aspect of Visual Studio is XML Web services, which provides the ability to exchange messages in a loosely coupled environment using standard protocols such as HTTP, XML, XSD, SOAP, and WSDL.

The question comes out that which GUI framework is better? On one hand, Java Swing has been improving over the years and there are excellent Java GUI component vendors produce better GUI components. Java Swing uses the MVC pattern much more extensively than Windows Form, and some GUI features such as LayoutManager and border style are more convenience in Java Swing than in Windows Form. On the other hand, Windows Form is faster than Java Swing and it is the best way to develop windows application. Both Java Swing and .NET Windows Form have their own advantages and disadvantages. Programmers have the choice to choose the better tool for their own needs.

Chapter 6 Conclusion

Java became very popular among many developers for several years and has made phenomenal inroads into the world of system, business, internet and education programming. C# derived from C and C++ is Microsoft's answer to Sun's popular Java programming language and a part of Microsoft's grand .NET initiative. The similarity between Java and C# is a great benefit to any organization with an existing training investment in Java or C++. Programmers accustomed to Java will have no trouble understanding C#. Moreover, Windows programmers who invested time learning Java (often in the form of Visual J++) will come up to speed on C# even more quickly than C++ programmers. It's worth noting just how many of these features are "Java-like," as they were introduced in Java and will thus be comfortable and familiar to existing Java developers. Garbage collection, interfaces, and type-safe variables are part of Java's feature set. After all, C# is not a Java clone; it is designed much closer to C++ and it adds some very handy features of its own. C# borrows most of its operators, keywords, and statements directly from C++. It provides greater expressiveness and is more suited to writing performance-critical code than Java, while sharing Java's elegance and simplicity, which makes both much more appealing than C++.

Java has many years of head start, so it is more mature and has a strong user base. Java will run on many different operating systems without recompiling code is always being a big advantage. However, C# has many more built-in language features, and it provides direct

access to operating system services in the Windows platform. [11] No doubt that C#'s designer had the benefit of knowing about Java's weaknesses and this allowed them to fix many of them in the C# language along with providing extra features. However, adding new features makes the C# language more complex and difficult for novice programmers. The great thing about Java is its simplicity and is currently the only choice for cross platform development. In fact, it has the advantages that it is popular used among programmers and wildly supported by many different vendors. Overall, I think that these two languages share enough in common and it is possible for programmers finding plenty of interesting work with both languages.

References:

[1] *Object Oriented Programming*, http://www.tutorgig.com/encyclopedia

[2] *Programming Microsoft Windows with C#* by *Charles Petzold*, Microsoft Press, 2002

[3] *Inside C#* by *Tom Archer*, Microsoft Corporation, 2002

[4] *Professional C# (2nd Edition)* by *Simon Robinson, Burt Harvey, Christian Nagel, Ollie Cornes, Karli Watson, Morgan Skinner, Jay Glynn, Zach Greenvoss & Scott Allen*, Wrox Press Inc, March 2002

[5] *The Java™ Tutorial* by *Sun Microsystems*, http://java.sun.com/docs/books/tutorial/

[6] *C# and Java: Comparing Programming Languages* by *Kirk Radeck*, Microsoft, February 2003

[7] *Multithreading in* C# By *Paul Deitel, Harvey Deitel, Jeffrey Listfield, Tem Nieto, Cheryl Yaeger, Marina Zlatkina* www.informit.com February 28, 2003

[8] *C#: A Thread to Java?* by *Setul Verma* www.csharphelp.com

[9] *A Comparison of Microsoft's C# Programming Language to Sun Microsystems' Java Programming Language* by *Dare Obasanjo,* Whitepaper

[10] *A Comparative Overview of C#* by *Ben Albahari,* Genamics, http://genamics.com August 10, 2000

[11] *Microsoft MSDN online library* by *Microsoft*, http://msdn.microsoft.com

[12] *What's New in C# 2.0* by *Bill Wagner*, http://www.c-sharppro.com/features/2003/10/cs200310bw_f/cs200310bw_f.asp