C# for Java Developers

At first glance, Java developers might not get particularly excited about C# code, because of the syntactical similarity between it and Java. However, look more closely and you will see subtle yet significant differences: features such as operator overloading, indexers, delegates, properties, and type safe enumerations in C#.

This appendix focuses on applying much-loved Java programming tricks to C# code, highlighting features that C# adds to the picture, and pointing out tricks that C# cannot do (although you won't find many of those). Of course, we assume that as a reader of this appendix, you are a professional Java developer; so we will not go into too much detail when describing the Java language.

Starting Out

Let's take a look at the infamous "Hello World!" example in Java:

```
public class Hello {
public static void main(String args []) {
System.out.println("Hello world! This is Java Code!");
}
}
```

The corresponding C# code for this is as follows:

```
using System;
public class Hello
{
public static void Main(string [] args)
{
System.Console.WriteLine("Hello world! This is C# code!");
}
}
```

The first thing that you'll notice is that the two appear to be very similar syntactically and both languages are case-sensitive. C# is object-oriented like Java, and all functionality must be placed inside a class (declared by the keyword `class`). These classes can contain methods, constructors, and fields just as Java classes can, and a C# class can inherit methods and fields from another class or interface as in Java. The implementation of classes and methods is similar in both languages.

C# code blocks are enclosed by braces just as in Java. The entry point to a C# application is the static `Main()` method, as required by the compiler (similar to Java but note the uppercase "M"). Also note that only one class in the application can have a `Main()` method. Similar to Java, the static keyword allows for the method to be called without creating an instance of the class first. For the `Main()` method in C# you have the choice of either a `void` or int return type. `void` specifies that the method does not return a value and `int` specifies that it returns an integer type.

The `using` keyword in C# corresponds to the import keyword in Java. Therefore, in the C# code above, we are essentially importing the C# equivalent of a class package called `System`. In C#, a class package is called a *namespace*, and we will look more closely at these in the next section.

Note that although we have written it with a lowercase *s* here, in C# the string type can also be written with a capital *S* as String. You will also notice that the array rank specifier (`[]`) has been shifted from in front of the `args` variable in the Java example, to between the string type and `args` variable in the C# sample. In fact, this specifier can occur before or after the variable in Java. However, in C#, the array rank specifier must appear before the variable name because an array is actually a type of its own indicated by type `[]`. We'll discuss arrays in more depth a bit later.

Finally, as you might expect, the names of methods tend to differ between the languages. For example, in Java we would use `System.out.println()` to display text in the command console. The equivalent to this method in C# is `System.Console.WriteLine()`.

Compiling and Running C# Code

In Chapter 2, we noted that like Java code, C# source code is compiled in two stages: first to Intermediate Language (IL), and then to native code. To run the previous C# code, you need to save it with an appropriate filename (for example, HelloWorld) and file extension .cs, and then compile it to IL using the csc command:

```
csc HelloWorld.cs
```

The next step is to compile the IL to native code and run the example. To do this, just type the name of the file, without the extension (as we would with Java code):

```
HelloWorld
Hello world! This is C# code!
```

Namespaces

While Java classes reside in logical divisions referred to as *packages*, C# (and other managed) classes are grouped together into *namespaces*. Packages and namespaces differ significantly in their implementation. A Java class that you want to make part of the `com.samples` package, for example, must have package `com.samples;` as the first line of code in the file. This is, of course, excluding any comments. Any code

within that file automatically becomes a part of the specified package. Also, a Java package name is associated with the folder containing the class file in that they must have the same name. The `com.samples` package must therefore be in class files that exist in the com/samples folder. Let's take a look at some examples of how packages work in Java:

```
package java2csharp.javasamples;
public class Hello {
public static void main(String args []) {
System.out.println("Hello world! This is Java Code!");
}
}
```

The following list provides examples of how the previous code could be referenced or executed. This list assumes that the class file has been made available to the JRE:

- From the command line:

```
java java2csharp.javasamples.Hello
```

- As a direct reference in the code:

```
public class Referencer {
java2csharp.javasamples.Hello myHello = new java2csharp.samples.Hello();
```

- By utilizing the `import` directive one could omit fully qualified package names, so `Referencer` could also be written as:

```
import java2csharp.javasamples.*;
public class Referencer {
Hello myHello = new Hello();
}
```

Wrapping a class in a namespace is achieved in C# by using the `namespace` keyword with an identifier, and enveloping the target class in brackets. Here is an example:

```
namespace java2csharp.csharpsamples
{
using System;
public class Hello
{
public static void Main(string [] args)
{
System.Console.WriteLine("Hello world! This is C# code!");
}
}
}
```

As you can see, we delimit layers of namespaces using the `.` operator, as in Java. Note that C# does not require an asterisk (*) needed in C#—applying the `using` directive implicitly imports all elements of the specified namespace. You will also have noticed the major difference from Java here: the use of namespace parentheses in which we place classes associated with the namespace. The advantage of using the parentheses like this is that we then disassociate package names from directory structures: feasibly we

could place a file containing this namespace anywhere within the directory as long as the CLR recognizes it. Therefore, it also enables us to call the file containing these classes anything we wish (it doesn't have to be the same name as the class as in Java); we can have more than one public class defined per file; and we can split the classes defined in this namespace into different files in different parts of the directory structure, as long as the namespace declaration appears in each of the files.

We can also introduce multiple namespaces in the same file with no restriction. We could, for example, add the definition of a new class and place it in a new namespace in the same file and still not be outside the bounds of the language:

```
namespace java2csharp.csharpsamples
{
using System;
public class Hello
{
public static void Main(string [] args)
{
System.Console.WriteLine("Hello world! This is C# code!");
}
}
}

namespace java2csharp.morecsharpsamples
{
using System;
public class AnotherHello
{
public static void Main(string [] args)
{
System.Console.WriteLine("Hello again! This is more C# code!");
}
}
}
```

As we pointed out in the previous section, classes from a particular namespace can be imported into another namespace with the `using` keyword. We can see that we import classes from the System namespace (the top level .NET Base Class namespace) into both namespaces above. We can also import classes from other namespaces directly into our classes by referring to the imported class using its full name (namespace included), in a similar way to using direct referencing of classes in Java code.

Namespaces may also be defined within other namespaces. This type of flexibility is impossible in Java without having to create a subdirectory. We could change the previous example so that the `AnotherHello` class is in the `java2csharp.csharpsamples.hellosamples` namespace:

```
namespace java2csharp.csharpsamples
{
namespace hellosamples
{
using System;
public class AnotherHello
{
public static void Main(string [] args)
```

```
{
System.Console.WriteLine("Hello again! This is more C# code!");
}
}
}
}
```

Java classes are part of a package; all classes created are part of the default package. C# mimics this functionality. Even if you do not declare one, a default namespace is created for you. It is present in every file and available for use in named namespaces. Just as in Java you cannot change package information, in C# namespaces cannot be modified either. Packages can span multiple files in the same folder; namespaces can span multiple files in any number of folders, and even multiple assemblies (the name given to code libraries in .NET), as discussed in Chapter 13.

Note that the default accessibility for types inside a namespace is internal. You must specify types as `public` if you want them available without full qualification; however, we strongly advise against this practice. No other access modifiers are allowed. In Java, internal package types may also be marked as `final` or `abstract` or not marked at all (this default access makes them available only to consumers inside the package). Access modifiers are discussed later in this appendix.

One final feature of namespaces not available to Java packages is that they may be given a `using` alias. `using` aliases make it very easy to qualify an identifier to a namespace or class. The syntax is simple. Suppose you had a namespace `Very.Very.Long.NameSpace.Name`. You could define and use a `using` alias (here VVLNN) for the namespace as follows:

```
using VVLNN = Very.Very.Long.Namespace.Name;
```

Declaring Variables

C# follows a similar scheme of variable declaration to Java, where the declaration consists of a `datatype` keyword and followed by the name of the variable to hold that `datatype`. For example, to declare an integer (`int`) variable called `myInt`, we would use the following code:

```
int myInt;
```

Identifiers are the names we give to classes, objects, class members, and variables. Raw keywords, discussed in the next section, can neither be Java nor C# identifiers; however, in C# you can use keywords as variable names by prefixing the name with `@`. Note that this exception is only with keywords and does not allow the breaking of any other rules. Although identifiers may have letters and numbers, the first letter of the identifier in both C# and Java must not be a number. Here are some valid and invalid examples of variable declaration:

```
int 7x; //invalid, number cannot start identifier
int x7; //valid, number may be part of identifier
int x; //valid
int x$; //invalid, no symbols allowed
int @class; //valid, prefix @ allows it to be used as an identifier
int @7k; //invalid, prefix @ only works for keywords
```

Variable Naming Conventions

Java uses camel-case notation for methods, properties, and variables, meaning that they are lowercase for the first letter in the name and capital letter for the first letter of every other word in the name. The first letter of class and object names in Java are uppercase. The following snippet shows the general syntax most Java programmers use:

```
int id;
int idName;
int id_name; //practiced also
final int CONSTANT_NAME; //widely adopted
int reallyLongId;
public class ClassName //every first letter capitalized
public interface InterfaceName
public void method(){}
public void myMethodName(){}
```

Based on the C# library classes, it is safe to make certain assumptions about C# naming conventions. A documented naming guideline for C# was not provided at the time of this writing. Each first letter of all method and property identifier names is capitalized, as is each first letter of all class and namespace names. Interfaces are preceded with an `I`. Variables are camel-cased, as shown in the following examples:

```
int id;
int idName;
public class ClassName //every first letter capitalized
public interface IInterfaceName //interface name preceded by I
public void Method(){} // first letter always capitalized
public void MyMethodName(){} // first letter of all other words capitalized
```

Data Types

Types in Java and C# can be grouped into two main categories: *value* types and *reference* types. As you are probably aware, value type variables store their data on the stack, while reference types store data on the heap. Let's start by considering value types.

Value Types

There is only one category of value type in Java; all value types are by default the primitive data types of the language. C# offers a more robust assortment. Value types can be broken down into three main categories:

- Simple types
- Enumeration types
- Structures

Let's take a look at each of these in turn.

Simple types

The C# compiler recognizes a number of the usual predefined datatypes (defined in the System Base Class namespace), including integer, character, Boolean, and floating point types. Of course, the value ranges of the indicated types may be different from one language to another. Below we discuss the C# types and their Java counterparts.

Integer values

C# has eight predefined signed and unsigned integer types (as opposed to just four signed integer types in Java):

C# Type	Description	Equivalent in Java
`sbyte`	Signed 8-bit	`byte`
`short`	Signed 16-bit	`short`
`int`	Signed 32-bit	`int`
`long`	Signed 64-bit	`long`
`byte`	8-bit unsigned integer	n/a
`ushort`	16-bit unsigned integer	n/a
`uint 32-bit`	Unsigned integer	n/a
`ulong 64-bit`	Unsigned integer	n/a

When an integer has no suffix the type to which its value can be bound is evaluated in the order `int`, `uint`, `long`, `ulong`, `decimal`. Integer values may be represented as decimal or hexadecimal literals. In the following example the result is 52 for both values:

```
int dec = 52;
int hex = 0x34;
Console.WriteLine("decimal {0}, hexadecimal {1}",dec, hex);
```

Character values

`char` represents a single two-byte long Unicode character. C# extends the flexibility of character assignment by allowing assignment via the hexadecimal escape sequence prefixed by `\x` and Unicode representation via `\u`. You will also find that you will not be able to convert characters to integers implicitly. All other common Java language escape sequences are fully supported.

Boolean values

The `bool` type, as in Java, is used to represent the values `true` and `false` directly, or as the result of an equation as shown below:

```
bool first_time = true;
bool second_time = (counter < 0);
```

Decimal values

C# introduces the decimal type, which is a 128-bit data type that represents values ranging from approximately 1.0x10-28 to 7.9x1028. They are primarily intended for financial and monetary calculations where precision is important (for example, in foreign exchange calculations). When assigning the decimal type a value, `m` must be appended to the literal value. Otherwise, the compiler treats the value as a double. Because a double cannot be implicitly converted to a decimal, omitting the m requires an explicit cast:

```
decimal precise = 1.234m;
decimal precise = (decimal)1.234;
```

Floating-point values

The following table lists the C# floating type values and their Java equivalents.

C# Type	Description	Equivalent in Java
`Float`	Signed 32-bit floating point	`float`
`double`	Signed 64-bit floating point	`double`

Floating-point values can either be doubles or floats. A real numeric literal on the right-hand side of an assignment operator is treated as a double by default. Because there is no implicit conversion from float to double you may be taken aback when a compiler error occurs. The following example illustrates this problem:

```
float f = 5.6;
Console.WriteLine(f);
```

This example produces the following compiler error message.

```
Literal of type double cannot be implicitly converted to type 'float'; use an 'F'
suffix to create a literal of this type
```

There are two ways to solve this problem. We could cast our literal to `float`, but the compiler itself offers a more reasonable alternative. Using the suffix `F` tells the compiler this is a literal of type `float` and not `double`:

```
float f = 5.6F;
```

Although it is not necessary, you can use a D suffix to signify a `double` type literal.

Enumeration types

An *enumeration* is a distinct type consisting of a set of named constants. In Java you can achieve this by using `static` final variables. In this sense, the enumerations may actually be part of the class that is using them. Another alternative is to define the enumeration as an interface. The following example illustrates this concept:

```
interface Color {
static int RED = 0;
```

```
static int GREEN = 1;
static int BLUE = 2;
}
```

Of course, the problem with this approach is that it is not type-safe. Any integer read in or calculated can be used as a color. It is possible, however, to programmatically implement a type-safe enumeration in Java by utilizing a variation of the Singleton pattern, which limits the class to a predefined number of instances. The following Java code illustrates how this can be done:

```
final class Day { // final so it cannot be sub-classed
private String internal;
private Day(String Day) {internal = Day;} // private constructor
public static final Day MONDAY = new Day("MONDAY");
public static final Day TUESDAY = new Day("TUESDAY");
public static final Day WEDNESDAY = new Day("WEDNESDAY");
public static final Day THURDAY = new Day("THURSDAY");
public static final Day FRIDAY = new Day("FRIDAY");
}
```

As you can see from the above example, the enumerated constants are not tied to primitive types, but to object references. Also, because the class is defined as final, it can't be sub-classed, so no other classes can be created from it. The constructor is marked as `private`, so other methods can't use the class to create new objects. The only objects that will ever be created with this class are the static objects the class creates for itself the first time the class is referenced.

Although the concept is pretty simple, the workaround involves techniques that may not be immediately apparent to a novice after all, we just want a readily available list of constants. C#, in contrast, provides inbuilt enumeration support, which also ensures type safety. To declare an enumeration in C# the `enum` keyword is used. In its simple form an `enum` might look like this:

```
public enum Status
{
Working,
Complete,
BeforeBegin
}
```

In this example, the first value is 0 and the `enum` counts upward from there, `Complete` being 1 and so on. If for some reason you are interested in having `enum` represent different values you can do so by assigning them as follows:

```
public enum Status
{
Working = 131,
Complete = 129,
BeforeBegin = 132
}
```

You also have the choice of using a different numerical integral type by inheriting from `long`, `short`, or `byte`. `int` is always the default type, as demonstrated in this snippet:

```
public enum Status : int
{
Working,
Complete,
BeforeBegin
}
public enum SmallStatus : byte
{
Working,
Complete,
BeforeBegin
}
public enum BigStatus : long
{
Working,
Complete,
BeforeBegin
}
```

It might not be immediately apparent but there is a big difference between these three enumerations, tied directly to the size of the type they inherit from. The C# byte, for example, can contain one byte of memory. This means `SmallStatus` cannot have more than 255 constants; if you want more, set the value of any of its constants to more than 255. The following listing displays how we can use the `sizeof()` operator to identify the differences between the different versions of Status:

```
int x = sizeof(Status);
int y = sizeof(SmallStatus);
int z = sizeof(BigStatus);
Console.WriteLine("Regular size:\t{0}\nSmall size:\t{1}\nLarge size:\t{2}",
x, y, z);
```

Compiling the listing produces the following results:

```
Regular size: 4
Small size: 1
Large size: 8
```

Structures

One of the major differences between a C# *structure* (identified with the keyword `struct`) and an object is that, by default, the struct is passed by value, while an object is passed by reference. There is no analogue in Java to structures. Structures have constructors and, methods; they can have other members normally associated with a C# class too: indexers (for more on these members see Chapter 4), properties, operators, and even nested types. Structures can even implement interfaces. By using structs we can create types that behave in the same way as, and share similar benefits to, the built-in types. The following snippet demonstrates how a structure can be used:

```
public struct EmployeeInfo
{
```

```
public string firstName
public string lastName
public string jobTitle
public string dept
public long employeeID
}
```

Although we could have created a class to hold the same information, using a struct is a little more efficient here because it is easier to create and copy it. The following snippet shows how to copy values from one struct to another:

```
EmployeeInfo employee1;
EmployeeInfo employee2;
employee1 = new EmployeeInfo();
employee1.firstName = "Dawn";
employee1.lastName = "Lane";
employee1.jobTitle = "Secretary";
employee1.dept = "Admin";
employee1.employeeID = 203;
employee2 = employee1;
```

Structures are often used to tidy up function calls too: we can bundle up related data together in a struct and then pass the struct as a parameter to the method. However, the following limitations apply to using structures:

- A struct cannot inherit from another struct or from classes.
- A struct cannot act as the base for a class.
- Although a struct may declare constructors, those constructors *must* take at least one argument.
- The struct members cannot have initializers.

Structs and attributes

Attributes (or compiler directives, discussed in Chapter 10 and Appendix D) can be used with structures to add more power and flexibility to them. The `StructLayout` attribute in the `System.Runtime.InteropServices` namespace, for example, can be used to define the layout of fields in the struct. It is possible to use this feature to create a structure similar in functionality to a C/C++ union. A union is a data type whose members share the same memory block. It can be used to store values of different types in the same memory block. In the event that one does not know what type the values to be received will be, a union is a great way to go. Of course there is no actual conversion happening; in fact there are no underlying checks on the validity of the data. The same bit pattern is simply interpreted in a different way. The following snippet demonstrates how a union could be created using a struct:

```
[StructLayout(LayoutKind.Explicit)]
public struct Variant
{
[FieldOffset(0)]public int intVal;
[FieldOffset(0)]public string strinVal;
[FieldOffset(0)]public decimal decVal;
[FieldOffset(0)]public float floatVal;
[FieldOffset(0)]public char charVal;
}
```

The `FieldOffset` attribute applied to the fields is used to set the physical location of the specified field. Setting the starting point of each field to `0` ensures that any data store in one field will overwrite to a certain extent whatever data may have been stored there. It follows then that the total size of the fields will be the size of the largest field, in this case the decimal.

Reference Types

All a reference type variable stores is the reference to data that exists on the heap. Only the memory addresses of the stored objects are kept in the stack. The object type, arrays, and interfaces are all reference types. Objects, classes, and the relationship between the two do not differ much between Java and C#. You will also find that interfaces, and how they are used, are not very different in the two languages. We will look at classes and class inheritance in C# in more depth later in this document. Strings can also be used the same way in either C# or Java. C# also introduces a new type of reference type called a *delegate*. Delegates represent a type-safe version of C++ function pointers (references to methods) and are discussed in Chapter 6.

Arrays and collections

Array syntax in C# is very similar to that used in Java. However, C# supports "jagged" arrays, and adds multidimensional arrays (as opposed to the arrays of arrays supported by Java):

```
int[] x = new int[20]; //same as in Java except [] must be next to type
int[,] y = new int[12,3]; //same as int y[][] = new int[12][3];
int[][] z = new int[5][]; //same as int x[][] = new int[5][];
```

In C#, arrays are actual types, so they must be written syntactically as such. Unlike in Java, you cannot place the array rank specifier `[]` before or after the variable; it must come before the variable and after the data type. Since arrays are types, they have their own methods and properties. For example, we can get the length of array `x` using:

```
int xLength = x.Length;
```

We can also sort the array using the static `Sort()` method:

```
Array.Sort(x);
```

You should also note that although C# allows us to declare arrays without initializing them, we cannot leave the determination of the size of an array until runtime. If you need a dynamically sized array, you must use a `System.Collections.ArrayList` object (similar to the Java's `Arraylist` collection). We cover C# collection objects in depth in Chapter 9.

Type Conversion and Casting

Type conversion in Java consists of implicit or explicit narrow and wide casting, using the `()` operator as needed. It is generally possible to perform similar type conversions in C#. C# also introduces a number of powerful features built into the language. These include *boxing* and *unboxing*.

Because value types are nothing more than memory blocks of a certain size, they are great to use for speed reasons. Sometimes, however, the convenience of objects is good to have for a value type. Boxing and unboxing provide a mechanism that forms a binding link between value types and reference types by allowing them to be converted to and from the object type.

Boxing an object means implicitly converting any value type to type `Object`. An instance of `Object` is created and allocated, and the value in the value type is copied to the new object. Here is an example of how boxing works in C#:

```
int x = 10;
Object obj = x;
```

This type of functionality is not available in Java. The previous code would not compile because primitives cannot be converted to reference types.

Unboxing is simply the casting of the Object type containing the value back to the appropriate value type. Again, this functionality is not available in Java. We can modify the code above to illustrate this concept. You will immediately notice that while boxing is an implicit cast, unboxing requires an explicit one:

```
int x = 10;
Object obj = x;
int y = (int) obj;
```

Another powerful feature of C# dealing with casting is the ability to define custom conversion operators for our classes and structs. We deal with this issue in depth in Chapter 5.

Operators

The following table lists the C# operators.

Category	Operator
Arithmetic	`+ - * / %`
Logical	`& | ^ ~ && || !`
String concatenation	`+`
Increment and decrement	`++ —`
Bit shifting	`<< >>`
Comparison	`== != < > <= >=`
Assignment	`= += -= *= /= %= &= |= ^= <<= >>=`
Member access (for objects and structs)	`.`
Indexing (for arrays and indexers)	`[]`
Cast	`()`
Conditional (the Ternary Operator)	`?:`
Object Creation	`new`
Type information	`sizeof` (unsafe code only) `is typeof as`
Overflow exception control	`checked unchecked`
Indirection and Address	`* -> &` (unsafe code only) `[]`

Java developers will immediately spot that C# operators are very similar to Java's. However, there are a few significant differences.

To determine whether an object belongs to a given class or any of the parent classes Java uses the `instanceof` operator. The C# equivalent of `instanceof` is the `is` operator. It returns `true` if the run-time type of the given class is compatible with the specified type. Here's an example of its use:

```
string y = "a string";
object x = y;
if(x is System.String)
{
System.Console.WriteLine("x is a string");
}
```

Also, since Java has no value types other than the primitives whose size is always known, there is no real use for a `sizeof` operator. In C#, value types range from primitives to structs to enums. As with Java, the size of the primitives is known. There is a need, however, to know how much space a `struct` type or `enum` type occupies. This is what the `sizeof` operator is for. The syntax is quite simple: `sizeof(<ValueType>)`, where `<Value Type>` is the struct or enum. Note that `sizeof` may only be used in an unsafe context. The `sizeof` operator cannot be overloaded.

The `typeof` operator is used to get an instance of a type's `System.Type` object without having to create an instance of the type. In Java, every type has a public static class variable that returns a handle to the `Class` object associated with that class. The `typeof` operator provides this type of functionality. Just as we saw with `sizeof`, the syntax is very simple. The statement `typeof(<Type>)` where `<Type>` is any user-defined type will return you the type object of that type.

Flow Control and Iteration

Most of the flow control statements are conceptually and syntactically very similar to Java's. Here's a brief summary:

`if...else` `if...else`

```
if(option == 1)
{
//do something
}
else if(option == 2)
{
//do something else
}
else
{
//do this if none of other options are selected
}
```

```
switch

    switch(option)
    {
    case 1:
    //do something
    break;
    case 2:
    //do something else
    break;
    default:
    break;
    }
```

You should note that the C# version of `switch` (unlike Java's) all but prohibits fall-through. All case clauses must end with a break, unless the case clause is empty. To jump from one case clause to another you must use a `goto` statement.

```
for

    for (int i = 0; i <10; i++)
    {
    // iterates 10 times
    }

while

    bool condition = false;
    while (!condition)
    {
    // do something that may alter the value of the condition Boolean
    }

do...while

    bool condition;
    do
    {
    // do something that may alter the value of the condition Boolean
    // at least one iteration occurs whatever the initial value of condition
    } while (condition);

foreach
```

C# introduces a `foreach` statement, used specifically to iterate through, and not change collection or array entries to get the desired information. Changing the contents might have unpredictable side effects. The `foreach` statement usually takes the following form:

```
    foreach (ItemType item in TargetCollection)
```

`ItemType` represents the data type stored in the collection or array and `TargetCollection` represents the actual array or collection. There are two sets of requirements that a collection you want to iterate through using the `foreach` statement must meet. The first set has to do with the composition of the collection itself. They are as follows:

- The collection type must be an interface, class, or struct.
- The collection type must include a `GetEnumerator()` method for returning an enumerator type. An enumerator type is basically an object that allows you to step through a collection item by item.

The second set of requirements deal with the composition of the enumerator type returned by the `GetEnumerator()` method mentioned above. Here is the list of requirements:

- The enumerator should provide a Boolean method `MoveNext()`.
- `MoveNext()` should return true if there are more items in the collection.
- `MoveNext()` should step to the next item in the collection at each invocation.
- The enumerator type must provide a property named `Current` that returns an `ItemType` (or a type that can be converted to `ItemType`).
- The property accessor `Current` should return the current element of the collection.

The following snippet of C# code uses `foreach` to iterate through a `Hashtable` collection:

```
Hashtable t = new Hashtable();
t["a"] = "hello";
t["b"] = "world";
t["c"] = "of";
t["d"] = "c-sharp";
foreach(DictionaryEntry b in t)
{
Console.WriteLine( b.Value );
}
```

We talked about the `break` statement in our discussion of `switch`; this statement can be used to exit from any flow control or iterative statement. The `continue` statement forces the end of the current iteration of an iterative statement, while `return` is used in a method to return control to the caller of the method.

Classes

Conceptually, classes in both C# and Java are very similar. A class is the template for an object, which is a data type that can hold both data and functionality that acts upon that data. Instantiating an object means creating a specific occurrence of that object, based on the class template. C# classes contain members that include methods (including constructors) and fields, like Java classes. However, there are some important conceptual differences between C# and Java classes, and a few different keywords too (as we would expect).

Access Modifiers

As with Java, we can add the usual modifiers to the start of the class or member declaration to modify the behavior of the class or member. The following table list the C# modifiers and their Java equivalents.

Access Modifier	Java Equivalent	Description
`public`	`public`	No restrictions on access. Members of enum and interface, as well as namespaces, are public by default.
`private`	`private`	Accessible only to the declaring class. Members of class and struct are private by default.
`internal`	n/a	Accessible to files in the same assembly.
`protected`	n/a	Accessible to the declaring class, and any subclass of the declaring class. In C# `protected` is more restrictive than in Java. Protected access will not allow other files in the same assembly to access the member.
`protected internal`	`protected`	Accessible to assembly files and subclasses of declaring class.

The private keyword is used to make methods and variables accessible only from within the containing class. It serves the same function in both languages. The `public` modifier allows entities outside the package/namespace to access the members of the class. However, C# and Java differ in the way protected and default are handled. While in Java, protected makes the method or variable accessible to classes in the same package or subclasses of the class, in C# protected makes code only visible to that class and subclasses that inherit from it.

C# also introduces a new access modifier: `internal`. The `internal` keyword modifies data members so that they are visible to all code within the entire component but not clients of that component. The difference between the `no` modifier in Java (which signifies an element that is accessible only to elements within the package) and `internal` is that `internal` is accessible to all elements of the assembly, which can span multiple namespaces.

Class Members

As we have seen throughout this document, the differences in syntax between C# and Java when declaring and referring to classes and their members is minimal. However, there are marked differences in class member modifier syntax, as explained in the following table.

Member Modifiers	Java Equivalent	Description
`virtual`	n/a	Allows target members to be overridden by an inherited class (the default in Java).
`static`	`static`	Target member marked as `static` belongs to class and not instance of class. Hence, there is no need to instantiate the class in order to gain access to it.
`event`	n/a	Used to bind client code to events of the class, the event modifier allows you to specify a delegate that will be called when some event in your code occurs. Note that it is the job of the class programmer to define when and where the event is raised, and the job of the subscriber to choose how to handle it.
`abstract`	`abstract`	Indicates that the target member is implicitly virtual and has no implementation code. The derived class must provide this implementation and the implemented method must be marked as override.
`const`	`final`	Indicates that the target member cannot be modified. Java also has a `const` keyword, which at the time of this writing is simply a reserved word.
`readonly`	n/a	Indicates that the target member can only be assigned values in its declaration or in the constructor of its containing class.
`extern`	n/a	Indicates that the target member is implemented externally. This modifier is typically used with the `DllImport` attribute.
`override`	n/a	Indicates that the target member provides a new implementation of a member inherited from a base class.

For more information on delegates and events, refer to Chapter 6.

As with Java, defining abstract methods in C# mandates that the class be abstract.

C# does not have a native modifier, and there is also no C# version of transient, volatile, or synchronized at the time of writing. In Java, using native indicates that the method is implemented in a platform-dependent language. It requires that the method be abstract since the implementation is to be found elsewhere. The closest relative to this type of functionality is the `extern` modifier. Using `extern` implies that the code is implemented externally (by some native DLL for example). Unlike Java, however, there is no need to use the `abstract` keyword in association with it. In the following snippet, the `Flower` class displays an example of how `extern` can be used:

```
public class Flower
{
public Flower(){}
public extern int GetColor();
// rest of Flower class definition
}
```

This doesn't make much sense without using the `DllImport` attribute to specify the external implementation. The following code provides the appropriate modifications, assuming there is a `See()` function exported by the User32.dll resource:

```
public class Flower
{
public Flower(){}
[System.Runtime.InteropServices.DllImport ("User32.dll")]
public static extern int GetColor();
// rest of Flower class definition
}
```

Note that we have now marked `GetColor()` as `static`. The `DllImport` attribute requires this of the methods it is used on.

Passing as reference to methods

Java and C# differ extensively in syntax and ideology regarding the way methods are handled by an object. For one thing, in C# not all reference data type parameters are passed as references and not all simple data types have to be passed by value. You have the option to pass arguments by value as an `in` parameter (this is the default way parameters are passed) by reference as a `ref` parameter, or as an `out` parameter. This is illustrated by the following code:

```
public static void Main(string[] args)
{
int a = 10;
Console.WriteLine(a);
AddOne(a);
Console.WriteLine(a);
}
public static void AddOne(int a)
{
a++;
}
```

This produces the following output in both C# and Java:

```
10
10
```

Because `a` is passed by value, the value that is passed is not tied to the value a in `Main()`. Consequently, incrementing `a` in the `Add()` method does not affect `a` in `Main()`. This is probably not the behavior we want; we would like the changes made to `a` to be remembered after the method call. We can do this by passing by reference instead of by value, like this:

```
public static void Main(string[] args)
{
int a = 10;
Console.WriteLine(a);
AddOne(ref a);
```

```
Console.WriteLine(a);
}
public static void AddOne(ref int a)
{
a++;
}
```

This produces the following output:

```
10
11
```

So, to use a reference parameter, we precede the parameter type with the `ref` keyword. We can also pass values back from a method using the `out` parameter. Note that `out` parameters do not need to be initialized before they are passed as arguments. The following code displays 100:

```
public static void Main(string[] args)
{
int a;
Add(out a);
Console.WriteLine(a);
}
public static void Add(out int a)
{
a = 100;
}
```

Properties

Unlike Java, C# does not use `get` and `set` methods to access an object's internal attributes. Instead it combines these methods together into another kind of class member called a *property*. A property contains a `get` accessor, which allows reading of internal fields of an object, and a `set` accessor that allows you to change the value of an internal field. The value keyword represents the new value to the right of the equals sign at assignment time. Not including the appropriate accessor in the property declaration will make the property either read-only (no `set`), or write-only (no `get`). The following class, `Person`, contains a few properties, called `Age` and `Name`:

```
public class Person
{
private int age;
private string name;
public Person(string name)
{
this.name = name;
}
public int Age
{
get
{
return age;
}
set
{
```

```
age = value;
}
}
public string Name
{
get
{
return name;
}
}
}
```

In the previous example, the property `Age` has a `get` and `set` accessor so you can read or write to the property. `Name`, however, is created only after you create a new instance of the properties object; after this you can only read the value of the `Name` property. Properties are accessed as if they are `public` fields:

```
Person john = new Person("John Smith");
john.Age = 21;
Console.WriteLine("My name is {0}, and I am {1} years old.", john.Name,
john.Age);
```

The output of the previous code is:

```
My name is John Smith, and I am 21 years old.
```

Note that property names must be unique.

Destructors

C# uses destructors in a similar way to C++. They work similarly to finalizers in Java; their syntax, however, is very different. With destructors, a tilde (~) prefixes the class name:

```
~Sample()
{
}
```

A word of advice concerning code in the destructor: the garbage collector in .NET is not invoked immediately after a variable goes out of scope. Indeed, there are certain intervals or memory conditions that bring the thread to life. Since there is a possibility that it might be triggered in low memory situations, consider making code in the destructor as short as possible. It is also a good idea to call `close()` on resource-intensive objects before destroying the controllers that use them.

Class Inheritance

Class inheritance in C# is also implemented in a very similar way to Java. Both languages are based on single implementation inheritance (in other words a subclass is only allowed to inherit from one other class) and multiple interface inheritance (a class can implement as many interfaces as desired).

C# does not have Java's extends or implements modifiers. To derive from a class or implement an interface in C#, we use the `:` operator. When a class base list contains a base class and interfaces, the base

class comes first in the list. The `interface` keyword is used to declare an interface. The following code shows examples of how to use these concepts:

```
//declare a parent/base class
class MyBaseClass
{
//class members
}
// declare an interface IFirstInterface
interface IFirstInterface
{
// interface members
}
// declare a subclass of MyBaseClass that inherits from interfaces too
class MySubClass : MyBaseClass, IFirstInterface, ISecondInterface
{
// class members
}
```

Abstract classes

As with Java, in C# we can use the `abstract` modifier in a class declaration to indicate that the class should not (and cannot) be instantiated. Classes derived from abstract classes must implement all the abstract methods of the class, and the sealed (see below) modifier cannot be applied to these methods.

Preventing inheritance

In C# the sealed modifier is used to prevent accidental inheritance, because a class defined as sealed can not be inherited from. Declaring a class as final achieves the same goal. Declaring a method as final also seals it, making it impossible to override. Declaring a variable as final is essentially making it read-only; however, you can still set a final value to the value of a variable. (This is different from constants, where the value of constants must be known at compile time so constants may only be set equal to other constants.)

Using base class members and base constructors

The keyword `this` works the same in Java and C#. In Java the `super` reference variable is used to signify the immediate parent class. In C# the equivalent is base. Take a C# class `CalculateFor` that provides the ability to work out the value of integer x raised to a particular integer power (for example, x raised to the power of three is x multiplied by x multiplied by x), given x and the power (provided an overflow does not occur):

```
using System;
public class CalculateFor
{
internal int x;
public CalculateFor(int x)
{
this.x = x;
}
public int ToThePower(int power)
{
int total = 1;
for(int i = 0; i < power; i ++)
```

```
{
total *= x;
}
return total;
}
}
```

We could use this class in other code like this, given a value of x of 9 and a value of power of 3:

```
CalculateFor myNumber = new CalculateFor(9);
int result = myNumber.ToThePower(3);
```

Let's introduce a subclass of `CalculateFor`, `ExpCalculateFor`, which contains a member floating-point variable, and the method `ToTheExponent()` that multiplies the result of ten to a particular power by that floating point value:

```
using System;
public class ExpCalculateFor
{
internal float y;
public ExpCalculateFor(float y) : CalculateFor(10)
{
this.y = y;
}
public int ToTheExponent(int power)
{
int total = 1;
for(int i = 0; i < power; i ++)
{
total *= base.x;
}
total *= y;
return total;
}
}
```

Notice the syntax used when referring to a base constructor in a subclass's constructor declaration. Actually we could simplify the `ToTheExponent()` method to the following, reusing the functionality of the base class's `ToThePower()` method:

```
public int ToTheExponent(int power)
{
float total = (base.x).(base.ToThePower(power));
total *= y;
return total;
}
```

Method overriding and hiding

In C#, method overriding is a very explicit procedure. This is quite different from the Java approach, where overriding is the default behavior when the signature of a super class member is the same as the signature of its subclass. In C#, to provide method overriding functionality, the modifiers virtual and override are used in tandem. All methods in the base class that you expect will be overridden must use

the `virtual` keyword. To actually override them use the `override` keyword in the child class. The following code uses an example class and subclass to demonstrate the override functionality:

```
using System;
public class FruitPlant
{
public FruitPlant(){}
public virtual void BearFruit()
{
Console.WriteLine("Generic fruit plant");
}
}
class MangoTree : FruitPlant
{
public MangoTree(){}
public MangoTree(){}
public override void BearFruit()
{
Console.WriteLine("Tree fruit is:->Mango");
}
}
public class FruitPlantTest
{
public FruitPlantTest(){}
public static void Main(string[] args)
{
FruitPlant p = new FruitPlant();
p.BearFruit();
MangoTree t = new MangoTree();
t.BearFruit();
((FruitPlant)t).BearFruit();
}
}
```

Compiling and running this code produces the following output:

```
Generic fruit plant
Tree fruit is:->Mango
Tree fruit is:->Mango
```

As you can see the most derived `Fruit()` method is called, irrespective of our use of final cast of the `MangoTree` instance to the `Plant` instance. Indeed, the benefit of using method overriding is that you are guaranteed that the most derived method will always be called.

Although we cannot override a method in C# unless the method was originally declared as virtual, C# also introduces a new concept, *method hiding*. This allows developers to redefine super-class members in the child class and hide the base class implementation even if the base member is not declared virtual. C# uses the new modifier to accomplish this.

The benefit of hiding members from the base class rather than overriding them is that you can selectively determine which implementation to use. By modifying the previous code we can see this concept in action:

```
public class FruitPlant
{
public FruitPlant(){}
public void BearFruit()
{
Console.WriteLine("Generic fruit plant");
}
}
class MangoTree : FruitPlant
{
public MangoTree(){}
new public void BearFruit()
{
Console.WriteLine("Tree fruit is:->Mango");
}
}
// then FruitPlantTest implementation
```

Running this example produces this output:

```
Generic plant fruit
Tree fruit is:->Mango
Generic plant fruit
```

In other words, unlike overriding, when hiding methods, the method invoked depends on the object the method is called on. For the last line of output, we cast the `MangoTree` instance back to a `Plant` instance before calling the `BearFruit()` method. So the `Plant` class's method is called.

You should note that the new modifier can also be used to hide any other type of inherited members from base class members of a similar signature.

Input and Output

Being able to collect input from the command prompt and display output in the command console is an integral part of Java's input/output functionality. Usually in Java one would have to create an instance of a `java.io.BufferedReader` object, using the `System.in` field in order to retrieve an input from the command prompt. The following code shows a simple Java class, `JavaEcho`, which takes input from the console and echoes it back, to illustrate the use of the `Java.io` package to gather and format input and output:

```
import java.io.*;
public class JavaEcho {
public static void main(String[] args)throws IOException {
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
String userInput = stdin.readLine ();
System.out.println ("You said: " + userInput);
}
}
```

In C#, the `System.Console` class provides methods that can provide similar functionality for reading and writing from and to the command prompt. There is no need for any extra objects; the `Console` class provides methods that can read entire lines, read character by character, and even expose the underlying stream being read from. The members of `Console` are briefly described in the following tables.

Public Properties	Description
`Error`	Gets the system's standard error output stream as a `TextWriter` object.
`In`	Gets the system's standard input stream as a `TextReader` object.
`Out`	Gets the system's standard output stream as a `TextWriter` object.

Public Methods	Description
`OpenStandardError()`	Overloaded. Returns the standard error stream as a Stream object.
`OpenStandardInput()`	Overloaded. Returns the standard input stream as a Stream object.
`OpenStandardOutput()`	Overloaded. Returns the standard output stream as a Stream object.
`Read()`	Reads the next character from the standard input stream.
`ReadLine()`	Reads the next line of characters as a string from `Console`, which is set to the system's standard input stream by default.
`SetError()`	Redirects the `Error` property to use the specified `TextWriter` stream.
`SetIn()`	Redirects the `In` property to use the specified `TextReader` stream.
`SetOut()`	Redirects the `Out` property to use the specified `TextWriter` stream.
`Write()`	Overloaded. Writes the specified information to `Console.Out`.
`WriteLine()`	Overloaded. Writes information followed by a line terminator to `Console.Out`.

All of the Console members are static, so you don't need to (and can't) instantiate a `System.Console`.

```
object.
Using the powerful methods of the Console class we could write an equivalent of the
JavaEcho class
in C# as follows:
class CSEchoer
{
static void Main(string[] args)
{
string userInput = System.Console.ReadLine();
System.Console.WriteLine ("You said : " + userInput);
}
}
```

The previous code is much shorter and easier to digest in comparison to its Java counterpart. One useful thing you'll get with the `Console.WriteLine()` static method is the ability to use formatted strings. The flexibility of formatted strings can be illustrated by writing a simple game where user input is used to generate a story. Here is the code for this game, `EchoGame`:

```
class EchoGame
{
static void Main(string[] args)
{
System.Console.WriteLine("Name of a country?");
string userInput1 = System.Console.ReadLine();
System.Console.WriteLine("Name of a young prince?");
string userInput2 = System.Console.ReadLine();
System.Console.WriteLine("What was the prince doing?");
string userInput3 = System.Console.ReadLine();
System.Console.WriteLine("What did he find while doing this?");
string userInput4 = System.Console.ReadLine();
System.Console.WriteLine("Then what did he do?");
string userInput5 = System.Console.ReadLine();
System.Console.WriteLine("Once upon a time in"
+ " {0}, there was a young prince {1},\n" +
"who while {2}, came across a {3}, and then "
+ "{4} ! ", userInput1, userInput2,
userInput3, userInput4, userInput5 );
}
}
```

The insertion points are replaced by the supplied arguments starting from the index `{0}`, which corresponds to the leftmost variable (in this case `userInput1`). You are not limited to supplying only string variables, nor are you confined to using just variables, or even using variables of the same type. Any type that the method `WriteLine()` can display can be supplied as an argument, including string literals or actual values. There is also no limit to the number of insertion points that can be added to the string, as long as it is less than the overall number of arguments. Note that omitting insertion points from the string will cause the variable not to be displayed. You must, however, have an argument for each insertion point you specify whose index in the argument list corresponds to the index of the insertion point. In the following listing, for example, removing `{1}` is fine as long as there are still three arguments. In this case `{0}` matches up with `strA` and `{2}` matches up with `strC`:

```
Console.WriteLine("hello {0} {1} {2}", strA, strB, strC);
```

Summary

Microsoft describes C# as a simple, modern language derived from C and C++. Because Java is also a modernization of C++, much of the syntax and inbuilt features present in C# are also available in Java.

C# uses.NET Framework, and so offers built-in, type-safe, object-oriented code that is interoperable with any language that supports the Common Type System (CTS). Java does offer interoperability with C and C++, but it is not type-safe. Moreover, it is highly complex.

C# namespaces provide a much more flexible way of grouping related classes. C# filenames are not bound to the classes within them as they are in Java, nor are namespace names bound to folders as package names are in Java. C# also provides a rich set of built-in value types, including type-safe enumerations, structures, and the built-in primitives that offer a robust alternative to Java's primitives.

C# provides bi-directional conversion between reference and value types called *boxing* and *unboxing*. This functionality is not supported in Java. C# supports the use of classes, complete with fields, constructors, and methods, as a template for describing types, and provides the ability to define destructors, methods called just before the class is garbage collected. C# also provides three approaches to method parameters—`in` (default), `out`, or `ref`.

C# also introduces the concept of method hiding, as well as supporting explicit overriding with the `virtual` and `override` keywords, and C# provides properties as an alternative to `get()` and `set()` methods as a way to access safely internal fields.