

Article

Intro to C# (vs. Objective-C and Java)

Document version: 1.0.0

Last updated: 10/6/2011

CONTENTS

OVERVIEW.....	3
OBJECTIVES.....	3
INTRODUCTION TO C#.....	3
Hello C#!+	6
.NET Runtime	9
.NET and C# Program Basics	10
C# Classes	12
C# Structs	16
C# Delegates & Events	17
Garbage Collection	19
Java Programmer Point of View	21
Objective-C Programmer Point of View	22
SUMMARY.....	23

Overview

C# (pronounced "C sharp") is a simple, modern, object-oriented, and type-safe programming language. It will immediately be familiar to C/C++ and Java programmers. C# combines the high productivity of Rapid Application Development (RAD) languages and the raw power of C++.

This article introduces C#, and compares it to other modern object-oriented languages like Objective-C and Java.

Objectives

When you've finished the article, you will have:

- A high-level understanding of the C# programming language.
 - A high-level understanding of the .NET framework.
 - A clear view of the differences and similarities in the implementation of basic scenarios in C# versus Java and Objective-C.
-

Introduction to C#

C# is an object-oriented language that enables programmers to build quickly a wide range of applications for the Microsoft® .NET framework. The goal of C# and the .NET framework is to shorten development time by freeing the developer from worrying about several low-level programming tasks such as memory management, type safety issues, standard libraries, array bounds checking, and so on. This allows developers to spend the majority of development time working on their applications and business logic instead.

C# is a simple, modern object-oriented and type-safe programming language. Let us see why.

Simple

- Being a part of the Microsoft .NET platform, C# applications receive support from the runtime, and they use services provided by the runtime. The .NET framework is a managed environment: the runtime manages memory allocation—this frees the developer from that task. Programmers do not have to deal with the burden of memory management and eliminating memory leaks.
- C# supports operator overloading, similarly to C++; Java and Objective-C do not support it.
- Inheritance-related features are clearer and more precise than in C++:
 - An **abstract** keyword is used instead of declaring pure virtual functions.
 - Abstract classes are declared as such directly by using the **abstract** keyword, not indirectly by lack of method/function implementation ("pure virtual functions"). An abstract class may be completely implemented and yet be declared abstract for architectural reasons.
 - An **override** keyword is used for overriding a virtual method.
- New class member types, such as properties and indexers, are introduced to simplify common tasks such as list iteration and property getting and setting.

Less Error Prone

C# is less error-prone than C++ due to the following characteristics:

- Unlike C++, where it is possible to declare any type of pointer and assign it to any allocated address or cast it to any other type, C# allows only type-safe casting. Casting errors are detected at either compile time or at run time. However, in either situation, you will not miss any errors.
- The use of Generic types allows template classes to remain type-safe at compile time. This allows the programmer to detect errors during compile, rather than at run time.
- Running under the managed CLR, bounds checking is performed for managed arrays automatically while accessing the allocated array member object. This prevents accidental memory corruption.

Modern

C# is designed to meet modern application requirements, as well as integrating modern software concepts:

- C#, as a .NET language, provides a larger set of basic types, including for example the Decimal type used to handle financial applications correctly.
- Known concepts, such as enumerated types, bit flags, first-class functions and others, become first-class citizen types with a rich set of functionalities that represent type-specific usage requirements.

- It contains a built-in exception-handling mechanism.
- C# provides a built-in event handler for simpler handling of integrated (for example, GUI and Web) application requirements.
- C# provides a complete reflection mechanism, allowing object metadata to be explored **during run time**.

Object Oriented

C# supports the three pillars of Object-Oriented Programming:

- **Encapsulation**
A class has tools to hide its implementation details and be responsible for its objects' state
- **Inheritance**
The "is a" relationship can be defined between classes, enabling extensibility and reusability
- **Polymorphism**
A program can handle different types of objects identically, sharing the same base, having polymorphic behavior, based on specific object types

Everything in C# is part of a type (for example, class, interface, and struct). There are no "global" (non-member) functions or variables.

Every type instance "is an **object**." All types are derived from a common base class: **object**.

While these concepts are very familiar to Java developers, they are fundamentally different (and in some cases even completely new) for C/C++/Objective-C developers.

A few words about Java

Java is a programming language originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications typically are compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is general-purpose, concurrent, class-based, and object-oriented. It is designed specifically to have as few implementation dependencies as possible.

A few words about Objective-C

The Objective-C language is a simple computer language designed to enable sophisticated object-oriented programming. Objective-C is defined as a small but powerful set of extensions to the standard ANSI C language. It is possible to compile any C program with an Objective-C compiler, and to freely include C code within an Objective-C class. Its additions to C are mostly inspired by Smalltalk, one of the first object-oriented programming languages. All of the syntax for non-object-oriented operations (including primitive variables, preprocessing, expressions, function declarations, and function calls) is

identical to C, while the syntax for object-oriented features is an implementation of Smalltalk-style messaging. Today, Objective-C is used primarily on Apple's Mac OS X and iOS: two environments based on the OpenStep standard, though not compliant with it. Objective-C is the primary language used for Apple's Cocoa API, and it was originally the main language on NeXT's NeXTSTEP OS.

“Hello C#!”

Historically, all articles about programming languages start with a “Hello World!” sample. This article will follow this tradition—here is a C# version of this popular program:

```
C#
using System;
class HelloApp
{
    static void Main()
    {
        Console.WriteLine("Hello C#!");
    }
}
```

This code references the "System" namespace, which provides the "Console" class, declares a class and enclosed "Main" static method, which, in turn, calls the Console class's "WriteLine" static method to output a string.

Note: Running the above sample directly under Windows Phone 7 will not produce any tangible output, as the Windows Phone 7 console is not accessible. The code provided is only for comparison.

Let us see the same program written in Java:

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

And now in Objective-C:

```
Objective-C
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    printf( "Hello Objective-C!\n" );
    return 0;
}
```

```
}
```

The first difference is in code organization: while both C# and Java have a class with one method, Objective-C has just a method.

The second difference lies in the signature of the method: in C#, the “Main” method must start with capital “M”, while in Java and Objective-C, it is not a requirement. The “String[]” argument for the “Main” method in C# is optional.

The third difference is in the “Main” method’s return type: in C# (and Java), it returns void while in Objective-C it returns “int”. In C#, the Main method could also return an “int” value.

The more we examine the code the more differences we will discover.

Defining a Class

Much of object-oriented programming consists of writing code for new objects—defining new classes. The class definitions in C# and Java are fairly similar:

C# (and Java)

```
class Fraction
{
    public int numerator;
    public int denominator;

    public void Print()
    {
        //C#
        Console.WriteLine("{0}/{1}", numerator, denominator);
        //Java
        //System.out.println(new PrintfFormat("%d/%d").sprintf(numerator,
denominator));
    }
}
```

In Objective-C, classes are defined in two parts:

1. An interface that declares the methods and instance variables of the class and names its superclass
2. An implementation that actually defines the class (contains the code that implements its methods)

Typically, these are split between two files, though sometimes a class definition may span several files by using a feature called a “category.” The same class/interface written in Objective-C:

Objective-C

```
#import <Foundation/NSObject.h>

@interface Fraction: NSObject {
```

```

    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
@end

```

Objective-C

```

#import "Fraction.h"
#import <stdio.h>

@implementation Fraction
-(void) print {
    printf( "%i/%i", numerator, denominator );
}

-(void) setNumerator: (int) n {
    numerator = n;
}

-(void) setDenominator: (int) d {
    denominator = d;
}

-(int) denominator {
    return denominator;
}

-(int) numerator {
    return numerator;
}
@end

```

The usage of the class is as follows:

C#

```

// Create a new instance
Fraction fraction = new Fraction();
// Set the values
fraction.numerator = 5;
fraction.denominator = 10;
// Print it
fraction.Print();

```

Java

```
// Create a new instance
final Fraction fraction = new Fraction();
// Set the values
fraction.numerator = 5;
fraction.denominator = 10;
// Print it
fraction.Print();
```

Objective-C

```
// Create a new instance
Fraction *frac = [[Fraction alloc] init];

// Set the values
[frac setNumerator: 1];
[frac setDenominator: 3];

// Print it
printf( "The fraction is: " );
[frac print];
printf( "\n" );

// Free memory
[frac release];
```

Freeing memory is vital in Objective-C programming since the memory is managed by the application, while in both C# and Java it managed by the runtime (that will be discussed later).

The whole concept of method calls (as it exists in C# and Java) is different in the Objective-C world when comparing the language syntax.

To get an object to do something, you send it a message telling it to apply a method. In Objective-C, message statements are enclosed in brackets:

Objective-C

```
[receiver message]
```

The receiver is an object, and the message tells it what to do. In source code, the message is simply the name of a method and any arguments that are passed to it. When a message is sent, the run-time system selects the appropriate method from the receiver's repertoire and invokes it.

.NET Runtime

Much like Java applications, C# applications work within a managed run-time environment called the common language runtime (CLR), which provides a number of services to the application.

Compilers and tools expose the common language runtime's functionality and enable you to write code that benefits from this managed execution environment. Code developed with a language compiler that targets the CLR is called managed code. It benefits from features such as cross-language integration,

cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services.

To enable the runtime to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in the compiled code. Metadata is stored with the code. Every loadable CLR portable executable (PE) file contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

The runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used (this is called garbage collection). Objects managed by the runtime are called managed data. Garbage collection eliminates memory leaks as well as some other common programming errors. You can use managed data, unmanaged data, or both managed and unmanaged data in your .NET Framework application.

The CLR makes it easy to design components and applications, whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be integrated tightly.

.NET and C# Program Basics

A C# program file should end with the **.cs** extension (in Java the file should end with the **.java** extension and in Objective-C with the **.m** extension). All the code in C# is written within class definitions—there are no “global entities” in the .NET object-oriented world. All class code is written within the class definition – there is no separation between class definition and implementation. All methods are written “inline”.

.NET classes are defined within namespaces. Internally, the name of the namespace in which a class is defined becomes part of the class name. Thus, different namespaces can include identical class names since the “full name” of the class is different.

The namespace name usually describes the general functionality topic, while the class name describes the class specific role in that general area.

Namespace members are used by prefixing the member name with the namespace name followed by the dot (.) operator. For example, to use the **Console** class, which is defined in the **System** namespace, we can write: **System.Console.WriteLine(...)**.

A more abbreviated way is to bring the required namespace into the program’s namespace, using the **using** keyword:

```
C#  
using System;
```

With this statement, **System** namespace members can be directly used, with no namespace related prefix additions: **Console.WriteLine(...)**.

We can define our own namespaces to serve as an organizational package for a few related types. You can declare your own namespace using the **namespace** keyword:

```
C#
namespace Graphics
{
    class Rectangle{...}
    class Circle{...}
}
```

.NET Common Type System

The .NET framework defines a type system, common to all languages that target the .NET environment, known as the Common Type System (CTS). The CTS defines a rich set of types that can be hosted by the CLR (for example, structs, classes, enumerators, interfaces, and delegates) and their basic definition rules. The CTS also provides a rich set of basic types, which will be presented soon. Based on this extensive type system, the .NET Framework defines a set of rules that are termed the Common Language Specification (CLS). The CLS is the minimum specification that any .NET compliant language (a language implemented on top of the CLR) must implement.

The CTS provides a rich set of pre-defined basic types for all .NET languages to use. This set contains primitive types, such as **System.Int32** and **System.Double**, as well as other commonly used types, such as **System.Decimal** and **System.String**.

The base of all .NET types, including the .NET primitive types, is **System.Object**. By defining a common base, .NET allows treating all types in common terms, such as defining collection types to store any type of elements, defining a generic method that can accept any argument type, and so on. **System.Object** contains a set of members that are applicable to any .NET type. Some of these members are virtual. Therefore, they can be overridden by deriving types to better represent their characteristics. In the .NET environment, types are divided into two different categories:

- Reference Types—Reference type objects are allocated from the heap. A reference type variable contains only a reference to the actual data, which resides on the heap.
 - A class instance is an example of a reference type. Consider the following class:

```
C#
class Employee
{
    public Employee(string name)//...
    private string name;
}
```

- The instance of such a class could be created as follows:

```
C#
```

```
Employee e = new Employee("John Doe");
```

Note: This statement allocates space for an **Employee** object on the heap, assigns it to the required data, and returns a reference to this object, to be stored in the reference variable "e".

- Value Types—Value-typed objects are allocated inline, either on the stack or as part of a containing object.
 - Primitive type variables, such as "int", are an example of a value type:

C#

```
int i = 32;
```

Note: This statement allocates 32-bits of space on the stack and places a 32-bit value into this space.

Note: In C#, the object's location (stack vs. heap) is not determined by the object's creation method (assignment operator vs. new). The variable type (value-type vs. reference-type) determines where the variable will be created, except for boxed value types¹.

From the .NET framework class hierarchy's perspective, reference types directly derive from **System.Object**; value types, however, derive from **System.ValueType**, which derives from **System.Object**. The **ValueType** class adjusts the **Object** class implementation (overrides some of its methods) to better suit a value-type variable's behavior.

As part of the safety concept prevalent in .NET, uninitialized variable usage is impossible: variables are either automatically initialized with default values (such as with class members and array elements), or the compiler prevents their usage (such as local method variables). When automatically assigned to their defaults, reference type variables default to **null**, numeric value type variables default to zero, and Boolean variables default to **false**. This process also does a recursive initialization of **struct** and **class** members.

C# Classes

C# classes support information hiding by encapsulating functionality in properties and methods and by enabling several types of polymorphism, including subtyping polymorphism via inheritance and parametric polymorphism via generics. You define classes by using the keyword "class" followed by an

¹ Boxing is the process of converting a [value type](#) to the type **object** or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a **System.Object** and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system, in which a value of any type can be treated as an object.

identifier—the name of the class. Next, you can create instances of the class by using the "new" keyword followed by the name of the class. As previously discussed, classes are reference-type variables.

Before we get deeper into the C# classes, let us introduce C# access modifiers. There are four main access modifiers:

Accessibility	Meaning
Public	Access is not restricted
Protected	Access is limited to the containing class or types derived from the containing class
Internal	Access is limited to the current assembly (project)
Private	Access is limited to the containing type

In addition, the **protected internal** access modifier combination can be used. This means that access is limited to the current assembly or types derived from the containing class.

Note: You can also apply the previous access modifiers to class members.

A class may contain a rich set of member types such as fields, constants, methods, constructors, destructor, properties, indexers, operators, and events.

You can define a class as static by using the **static** keyword in the class definition. It is not possible to create instances of a static class—static classes are loaded automatically by the .NET runtime when the program or namespace containing the class is loaded and used. The only constructor permitted in a static class is the static constructor. A static constructor will be called once at most, and will be called before the first time a static member of the type is accessed. An instance constructor (if one exists) will always run after a static constructor.

C# Class Members

Fields (known in C/C++ as data members) are initialized automatically with a default value according to their type category (reference-type vs. value-type). By default, a field is defined to be an instance field—a field that is contained in every instance of the class, holding an instance's specific value. However, a field can be made static, in which case its value is shared by all class instances. You define static fields using the **static** keyword. Public static fields can only be accessed through the class name using the dot (.) operator.

Another class member, the method, is a functional member of the class. In C#, all methods are defined inside the class declaration. Class methods can access other class members:

```
C#  
public void raiseSalary(int percentage)  
{  
    // Update the salary field
```

```
    salary += salary * percentage/100;
}
```

The object on which the method was activated is available inside the method using the keyword **this**. Observe the following sample where class fields are updated with the value of identically-named parameters:

```
C#
public Employee(string name, int salary)
{
    this.name = name;
    this.salary = salary;
}
```

Methods that manipulate static fields only should be defined as static methods. They can be activated using the class name alone and have no access to any non-static field in the class.

By default, method parameters in C# are passed by value.

- When the parameter is of a value-type, a local copy of the argument's value is created for the method. Working on this local independent value's copy, the method has no influence on the original argument's value.
- When the parameter is of a reference-type, a local copy of the argument's reference is created for the method. Through this reference copy, the method can alter the original argument's value, though it cannot change the reference itself.

Sometimes we would like the method to work directly on the argument passed, be it a value or reference type, instead of working on an independent local copy. To get such behavior, C# provides us with a special kind of parameter—reference parameters.

Note: This is the same as passing the address of the variable in C, or a reference in C++.

You define reference parameters by adding the **ref** keyword before the parameter type in the method's signature, and again in the method call. The **ref** keyword must be specified in both.

```
C#
// Function definition
public void Swap(ref int i1, ref int i2)
{
    int tmp = i1;
    i1 = i2;
    i2 = tmp;
}
// Function call
int i1 = 5, i2 = 10;
...
```

```
Swap(ref i1, ref i2);
```

Reference parameters should be initialized before they are passed to the function. In some cases, this is inconvenient, especially when their value is computed by the function. For such cases, C# offers another kind of parameter: output parameters. Output parameters allow us to pass non-initialized variables to a method. The method can then define them using the **out** keyword. Out parameters must be assigned inside the method. Failing to assign out parameters generates a compiler error. As with the **ref** keyword, both the function definition and call must include the **out** keyword to use output parameters.

C#

```
// Function definition
public void getSize(out int width, out int height)
{
    width = this.width;
    height = this.height;
}
//Function call
int width, height;
...
getSize(out width, out height);
```

To enable users to access class fields in an elegant yet controlled manner, C# presents properties. Properties can be considered as **smart fields**. They enable defining special get and set methods, termed **getter** and **setter**, which are activated using simple field access syntax. This way, we can provide the users with an intuitive field access service, yet still perform validation (and any other required) operations. You define properties as follows:

C#

```
//Field to store the property value
private int day;
// Property
public int Day
{
    // Getter
    get
    {
        // Any functionality could be executed here in order to alter
        // the return value
        return day;
    }
    // Setter
    set
    {
        // Any functionality could be executed here in order to validate
        // a value before it being assigned to the private field
        // Note: new property value is always provided by the keyword "value"
        if (isDayOK (value))
            day = value;
    }
}
```

```
}  
}
```

By default, property getter and setter access is provided by the property definition (in the previous example they both are public). You can explicitly change an accessor's visibility to differ from that of the property, though the accessibility modifier of the accessor must be more restrictive than that of the property. Only one of the accessors may have its visibility altered.

C#

```
public FieldType PropertyName  
{  
    get  
    {  
        return field;  
    }  
    protected set  
    {  
        field = value;  
    }  
}
```

C# Structs

While a class is a very general type, the .NET CTS defines additional, more specific, types that may better suit specific application requirements. Though we would usually use classes to define our types, classes might sometimes be too "heavy" for us. Class objects are reference type variables and their manipulation involves a lot of overhead. We may require a "lightweight" type, for which we would like more efficient object manipulation.

For such cases, C# enables the programmer to declare data structures, using the keyword- "struct".

Structures are value-type user defined types and as such are derived implicitly from `System.ValueType`. They are allocated and manipulated directly on the stack. You define structures using the **struct** keyword. Structures, like classes, can have fields and methods (as well as properties and events, which we have not covered yet). Structures can define constructors, and can override virtual methods derived from **System.ValueType**. However, they have some limitations:

- Structures cannot be part of an inheritance tree—they cannot inherit from other classes (except for their default base as value types: **System.ValueType**). They are sealed, which means they cannot serve as base classes for other classes.
- Though using structures saves the program from having to do heap allocation and reduces GC costs (as value types do not have to be collected), careless use of this data type may incur significant copying costs, as value types are copied whenever transferred—as a method argument, to another variable, and so on.

- As value-type objects, structure objects cannot have a **null** value unless they are wrapped as a `Nullable<T>` types.
- You cannot define your own parameterless constructor for a structure, as it already has a default (non-overrideable) parameterless constructor that initializes all fields and their default values.
- You cannot define a destructor for a structure.

Being more limited and activated differently from regular classes, structures should be used for lightweight, value-type objects. Possible examples are `Point`, the `Complex` number type, the `System`'s `decimal` and `DateTime` types, and so on. Such types are actually a simple combination of value-types, and users expect them to behave as primitive value-types (that is, zero instead of null, sealed).

C# Delegates & Events

During the course of development, we frequently call functions that expect a **function** as one of their arguments. These functions will call back our provided function during their execution. They are usually termed **callback functions**. The “traditional” way to pass functions as parameters was by sending a function pointer that is merely an address with no type information attached (such as the number and type of parameters, the return value, and so on). Therefore, passing function pointers is not type safe.

In the .NET framework, callback methods are used for several purposes: events, threading, item filtering and more. To enhance callback method support, the .NET framework comes with a new, type-safe mechanism termed **delegate**. **Delegate** objects refer internally to a method callback. However, compared to function pointers, they provide richer functionality in a more encapsulated and type-safe manner.

In C#, when defining a method that accepts a callback function as a parameter, we should do the following:

- Define a delegate type that identifies the required callback function prototype.
- Define our method as taking a parameter of the delegate type defined.

Users of our method will:

- Create an object of the specified delegate type, providing it with their callback function as a parameter. This will work only if the callback function follows the delegate's required prototype.
- Supply the newly created delegate object, which actually wraps their callback function, as a parameter to the method.

C# allows you to define a special callback wrapper type, using the **delegate** keyword. As a wrapper of a callback, the type dictates a specific signature, for example:

```
C#
public delegate void CarOperation (Car c);
```

To use a specified delegate, you create an object of that delegate type and pass to it the required callback target (a method). This callback method should match the delegate's defined signature.

Using the delegate defined above as an example, we will show some functions that could be used as callback functions by assigning them to instances of the above delegate type:

```
C#  
public void WashCar(Car c)  
{  
...  
}  
  
public void FixEngine(Car c)  
{  
...  
}
```

You can then pass a delegate instance to a general method that expects to get such a delegate object as a parameter:

```
C#  
public void processCars (CarOperation operation);
```

C# also offers the `Func<T1,T2,T3,..>` and `Action<T1[,T2,T3,..]>` generic delegates, providing the programmer with general purpose delegates and eliminating the need to declare specific ones for every case.

Events are a programmable logical concept. They allow an object to notify observers of internal status changes, without having any other object poll or access its internal information.

For this purpose, events overload the "+" and "-" operators, allowing the programmer to add or remove event affected code (known as "event handlers"), respectively, by defining delegates to handle the event.

You may see events as a special case of delegates, which can be invoked only from within the declaring type, as opposed to standard delegates.

For example, when a user clicks a button using the mouse, several objects in the application would like to be notified so that they can react accordingly. By defining an event member in our class, we enable external objects to receive a notification whenever the event occurs. Specifically, a type defining an event offers the following abilities:

- The ability for external objects to register (and unregister) their interest in the event
- The ability for the defining object to maintain the set of registered objects and notify them when the specific event occurs

When an object registers an interest in an event, it provides the event object with a callback method to be called when the event occurs. The ability to store and invoke a callback is implemented using the delegate mechanism described before. In its callback method, the registered object can respond to the event's occurrence. Generally, events in .NET use delegates of the following signature:

C#

```
public delegate void DelegateName(object sender, EventArgs args);
```

Where: "DelegateName" would be the delegate name definition and "args" would be a parameter of type "EventArgs" or a derivative.

For example, a delegate for an event notifying that a button has been clicked would look like this:

C#

```
public delegate void ClickHandler(object sender, ClickEventArgs args);
```

To complete the above example, define the event itself by adding an event member to the class:

C#

```
public event ClickHandler Click;
```

Note: Event members are used like delegates, instructing the receivers to provide a callback method matching the **ClickHandler** (in the above example) prototype.

To raise the event defined in the class (this can be done at any time according to the class's needs), use the following code (which is a best practice for firing events):

C#

```
void OnClick(ClickEventArgs e)
{
    // Make sure there are subscribers registered to the event
    if(Click != null)
        Click(this, e); //Raise the event (call the method)
}
```

Listeners should register their interest in the specific event, providing their callback method:

C#

```
classInstanceWithEventDefined.Click +=
    new ClassWithEventDefinition.ClickHandler(this.OnClick);
```

Note: In this code snippet, "classInstanceWithEventDefined" is an instance of the class in which the event was defined, and "OnClick" is a callback function that will be executed when the event is raised.

Garbage Collection

Every program uses resources of one sort or another—memory buffers, network connections, database resources, and so on. In fact, in an object-oriented environment, every type counts as a resource

available for a program's use. To use any of these resources, memory must be allocated to represent the type. The steps required to access a resource are as follows:

1. Allocate memory for the type that represents the resource.
2. Initialize the memory to set the initial state of the resource and to make the resource usable.
3. Use the resource by accessing the instance members of the type (repeat as necessary).
4. Tear down the state of the resource to clean up.
5. Free the memory.

The .NET runtime requires that all memory be allocated from the managed heap. The application never frees objects from the managed heap – the runtime frees objects automatically when they are no longer needed by the application. The .NET garbage collector (GC) completely absolves the developer from tracking memory usage and knowing when to free memory. The garbage collector's optimizing engine determines the best time to reclaim available memory ("perform a collection"), based on the allocations the application makes. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory. Garbage collection (not having to manage application memory) will be new to Objective-C developers (Java has a garbage collection mechanism similar to .NET's).

The .NET GC is a tracing garbage collector, meaning that unused objects are collected without having to maintain referencing information during the application's normal operation, and that unused objects are freed when the GC dictates (on the Windows Phone 7 GC, this occurs once for every megabyte of allocations, approximately). This model makes no promise on an object's collection timing. Therefore, it is a less-than-optimal solution when dealing with memory objects that need to be released in a timely manner and in a predictable order.

The "full .NET" memory allocator works by maintaining a block of free space in which a single pointer ("next object pointer") determines where to allocate the next object. This model works well with compaction—the process in which used memory blocks are joined together to create a compact, defragmented free region. However, on Windows Phone 7, the garbage collector does not have the luxury of performing compaction. Therefore, the memory allocator uses a free list of available memory blocks, similarly to classical memory allocators. Furthermore, on the "full .NET" GC, the heuristics for determining when a collection occurs are fairly sophisticated, accounting for object lifetime, collection frequency, working set size, and other criteria. The Windows Phone 7 GC, on the other hand, has a very simple trigger for performing a GC: whenever a megabyte of memory has been allocated.

When a garbage collection occurs, the GC constructs a graph of all referenced objects. To construct this graph, the GC starts from a set of roots. Roots are references that either refer to objects on the heap or are set to null, and include static fields, local variables in currently active methods, and GC handles.

The GC starts by traversing the set of active roots, building a graph of all objects reachable from the roots. This is the "Mark" phase. At the end of this process, the GC has a graph of all objects that are somehow accessible from the application—all objects that are not in the graph are considered garbage

and may be reclaimed. Next, the GC frees all memory objects for which no roots were discovered in the "Mark" phase. This is the "Sweep" phase.

At the next stage, which occurs occasionally, the GC compacts the memory objects within the allocated space to create a defragmented memory space. This stage occurs whenever fragmentation level exceeds a defined threshold. During this process, the GC moves objects in memory. Therefore, it updates all references to them.

Asynchronously from the GC's standard operation, a dedicated finalizer thread runs in the background and executes finalization code for objects that require it (a class requests finalization by declaring a finalizer, overriding the `Object.Finalize` protected method; in C#, you declare a finalizer by writing a method that has the name of the class prefixed by a ~ sign). Note that an object's finalizer runs only after the object has become unreachable by the application's code; referenced objects are not enqueued into the finalizer thread's queue. When the finalizer thread is done with an object, the last reference to it is removed, and it can be collected during the next GC cycle.

Even though finalization provides an easy way to reclaim unmanaged resources at some point after an object becomes unreachable, you should usually prefer the safer, faster, and deterministic "dispose pattern" (for more information about the Dispose pattern, see <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx>).

Java Programmer Point of View

From a Java developer's point of view, the C# language is very familiar. Both languages are object-oriented languages. Both are strongly typed, and both have a garbage collector that manages memory. C# differs by its ability to define stack-based objects using the **struct** keyword, its ability to pass primitive variables by reference, its simple event-based response mechanism based on delegates, and the existence of properties. To the Java developer, some of these differences can be circumvented. Other differences have no analog in Java.

The following partial list represents notable differences between C# and Java:

- In Java, types are declared in their own files. There can be only one type in a single Java source file, and the file name must be the same as the type name.
- C# namespaces are mostly equivalent to Java packages. Every type is contained in a package, which is defined at the top of the source file (where the type is declared). Usually, the whole application defines a root package such as "com.mycompany.appname", and all types exist in this package or sub-packages (such as "com.mycompany.appname.subpackage"). The package definitions may also reflect the folder hierarchy in the project. However, there is no aliasing or nesting of packages.
- In Java, inheritance is done using the "extends" keyword and interface implementation is done using the "implements" keyword (for example, "class A extends B implements IC, ID")
- There is no "readonly" keyword (or equivalent) in Java.

- All user-defined types are reference types and only primitive types are value types. Reference types are always passed by reference and value types are always passed by value (copied). There's no equivalent to the "ref" or "out" keyword in Java.
- There is no delegate concept in Java. The only way to simulate this is using reflection.
- Java's "synchronized" keyword ensures thread safety before entering a method as does the [MethodImpl(MethodImplOptions.Synchronized)] attribute in C#.
- In Java, a programmer cannot implement two interfaces that declare the same method.
- Java doesn't allow static constructors. Instead, there's a static block.
- In Java, there is no property construct—properties need to be defined explicitly (declaring set/get methods in the containing class).
- In Java, the "protected" access modifier allows access from any other type in the same package, in addition to derived types.
- In Java, all methods are virtual by default, and there are no "virtual" or "override" keywords.
- In Java, a programmer cannot further restrict access to an override (over the access of the overridden member).
- There are no indexers in Java.

Objective-C Programmer Point of View

Other than the fact that both languages are object-oriented in nature, Objective-C syntax is very different from C#. There are a differences in coding approach (like sending messages to objects in Objective-C rather than calling functions in C#), memory management (Objective-C has no memory management mechanism), the type system (.NET's strong type system as opposed to pointers to loosely typed memory locations in Objective-C), etc.

The following partial list represents notable differences between C# and Objective-C:

- Objective-C types are defined within two separate files—header and implementation files (.h and .m, respectively).
- In Objective-C, there are two types of methods in classes: instance and class methods, denoted by a prepended "+" or "-" in the method prototype. Class methods are just like *static* methods in C#.
- In Objective-C, all methods are public, while C# defines 4 access modifier types.
- NSObject in Objective-C is an equivalent of System.Object.

- In Objective-C, what we typically think of as constructors are “init” methods. Destructors/Finalizers are “dealloc” methods. These methods are just a programming convention and are not special class members.
- In Objective-C, properties are denoted with @property lines in the header file. Corresponding @synthesize statements in the implementation file generate the property code. Special directives in the @property declaration determine the specifics of the implementation (such as nonatomic, retain).
- Objective-C’s *Protocols* are equivalent to *Interfaces* in C#.
- The *Self* (in Objective-C) pointer is the equivalent of *this* in C#.
- *#import* is Objective-C’s version of C#’s *using* statements.
- The *super* keyword in Objective-C is termed *base* in C#.
- In Objective-C, sending a message to an object is equivalent to calling method in C#.
- Objective-C’s *@selector()* is equivalent to C# delegates.

Summary

C# is a simple, modern, object-oriented, and type-safe programming language derived from C and C++, that aims to combine high productivity and efficiency.

C# introduces clear syntax, type safety, and automatic memory management. It enables a relatively easy migration path for Java developers and greatly simplifies the development effort for Objective-C developers while preserving the key concepts of object-oriented programming and its programming power.