C# Basics

A First C# Program

using System;

```
class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30));       // 360
    Console.WriteLine (FeetToInches (100));      // 1200
  }

  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Custom Type Examples

Just as we can build complex functions from simple functions, we can build complex types from primitive types. In this example, we will define a custom type named `UnitConverter`—a class that serves as a blueprint for unit conversions:

```
using System;

public class UnitConverter
{
  int ratio;
// Field
  public UnitConverter (int unitRatio) {ratio =
unitRatio; } // Constructor
  public int Convert   (int unit)    {return unit *
ratio; } // Method
```

```
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInchesConverter = new
UnitConverter (12);
    UnitConverter milesToFeetConverter  = new
UnitConverter (5280);

    Console.WriteLine
(feetToInchesConverter.Convert(30));    // 360
    Console.WriteLine
(feetToInchesConverter.Convert(100));   // 1200
    Console.WriteLine (feetToInchesConverter.Convert(

milesToFeetConverter.Convert(1)));   // 63360
  }
}
```

Constructors and instantiation

Data is created by *instantiating* a type. Predefined types can be instantiated simply by using a literal such as `12` or `"Hello world"`. The `new` operator creates instances of a custom type. We created and declared an instance of the `UnitConverter` type with this statement:

```
UnitConverter feetToInchesConverter = new UnitConverter
(12);
```

Immediately after the `new` operator instantiates an object, the object's *constructor* is called to perform initialization. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
public class UnitConverter
```

```
{

  ...

  public UnitConverter (int unitRatio) { ratio =
unitRatio; }
  ...
}
```

Instance versus static members

The data members and function members that operate on the *instance* of the type are called instance members.

The `UnitConverter`'s `Convert` method and the `int`'s `ToString` method are examples of instance members. By default, members are instance members.

Data members and function members that don't operate on the instance of the type, but rather on the type itself, must be marked as `static`. The `Test.Main` and `Console.WriteLine`methods are static methods. The `Console` class is actually a *static class*, which means *all* its members are static. You never actually create instances of a `Console`—one console is shared across the whole application.

Let's contrast instance from static members. In the following code, the instance field `Name` pertains to an instance of a particular `Panda`, whereas `Population` pertains to the set of all `Panda` instances:

```
public class Panda
{
  public string Name;              // Instance field
  public static int Population;    // Static field

  public Panda (string n)          // Constructor
  {
    Name = n;                      // Assign the
instance field
    Population = Population + 1;   // Increment the
static Population field
  }
}
```

The following code creates two instances of the `Panda`, prints their names, and then prints the total population:

```
using System;

class Test
{
  static void Main()
  {
    Panda p1 = new Panda ("Pan Dee");
    Panda p2 = new Panda ("Pan Dah");

    Console.WriteLine (p1.Name);      // Pan Dee
    Console.WriteLine (p2.Name);      // Pan Dah

    Console.WriteLine (Panda.Population);   // 2
  }
}
```

Attempting to evaluate `p1.Population` or `Panda.Name` will generate a compile-time error.

Conversions

C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either *implicit* or *explicit*: implicit conversions happen automatically, and explicit conversions require a *cast*. In the following example, we *implicitly* convert an `int` to a `long` type (which has twice the bitwise capacity of an `int`) and *explicitly* cast an `int` to a `short` type (which has half the capacity of an `int`):

```
int x = 12345;      // int is a 32-bit integer

long y = x;           // Implicit conversion to 64-bit
integer

short z = (short)x;  // Explicit conversion to 16-bit
integer
```

Implicit conversions are allowed when both of the following are true:

- The compiler can guarantee they will always succeed.
- No information is lost in conversion.[1]

Conversely, *explicit* conversions are required when one of the following is true:

- The compiler cannot guarantee they will always succeed.
- Information may be lost during conversion.

Value Types Versus Reference Types

All C# types fall into the following categories:

- Value types
- Reference types
- Generic type parameters
- Pointer types

Value types comprise most built-in types (specifically, all numeric types, the `char` type, and the `bool` type) as well as
custom `struct` and `enum` types.
Reference types comprise all class, array, delegate, and interface types. (This includes the predefined `string` type.)
The fundamental difference between value types and reference types is how they are handled in memory.

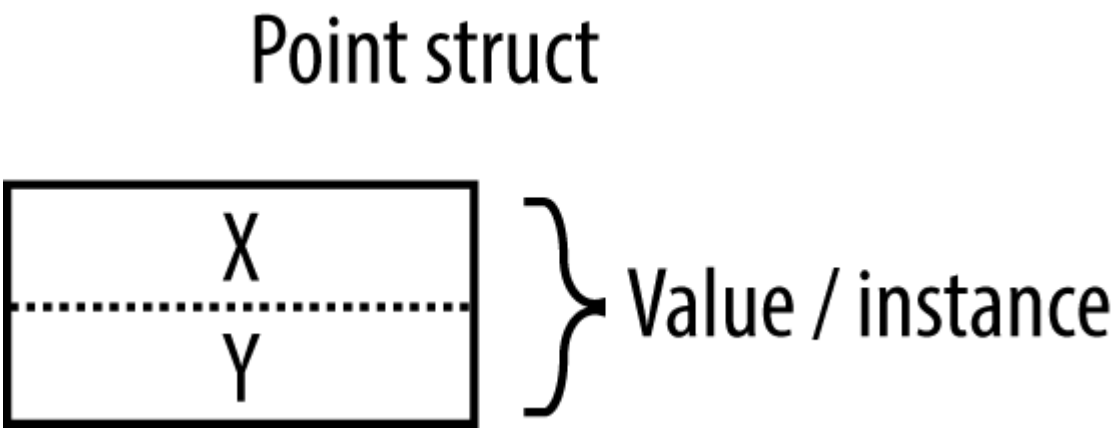
Value types

The content of a *value type* variable or constant is simply a value. For example, the content of the built-in value type, `int`, is 32 bits of data. You can define a custom value type with the `struct` keyword (see Figure 2-1):

```
public struct Point { public int X; public int Y; }
```

or more tersely:

```
public struct Point { public int X, Y; }
```



The assignment of a value-type instance always *copies* the instance. For example:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;             // Assignment causes copy

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                  // Change p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 7
}
```

Reference types

A reference type is more complex than a value type, having two parts: an *object* and the *reference* to that object. The content of a reference-type variable or constant is a reference to an object that contains the value. Here is the `Point` type from our previous example rewritten as a class, rather than a `struct` .

```
public class Point { public int X, Y; }
```

Assigning a reference-type variable copies the reference, not the object instance. This allows multiple variables to refer to the same object—something not ordinarily possible with value types. If we repeat the previous example, but with `Point` now a class, an operation to `p1` affects `p2`:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;             // Copies p1 reference

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                  // Change p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 9

}
```

Null

A reference can be assigned the literal `null`, indicating that the reference points to no object:

```
class Point {...}
...

Point p = null;
Console.WriteLine (p == null);   // True

// The following line generates a runtime error
// (a NullReferenceException is thrown):
Console.WriteLine (p.X);
```

In contrast, a value type cannot ordinarily have a null value:

```
struct Point {...}
...
```

```
Point p = null;  // Compile-time error
int x = null;    // Compile-time error
```

Numeric Types

C# has the predefined numeric types shown in Table 2-1.

C# type	System type	Suffix	Size	Range
Integral—signed				
`sbyte`	`SByte`		8 bits	-2^{7} to $2^{7}-1$
`short`	`Int16`		16 bits	-2^{15} to $2^{15}-1$
`int`	`Int32`		32 bits	-2^{31} to $2^{31}-1$
`long`	`Int64`	`L`	64 bits	-2^{63} to $2^{63}-1$
Integral—unsigned				
`byte`	`Byte`		8 bits	0 to $2^{8}-1$
`ushort`	`UInt16`		16 bits	0 to $2^{16}-1$
`uint`	`UInt32`	`U`	32 bits	0 to $2^{32}-1$
`ulong`	`UInt64`	`UL`	64 bits	0 to $2^{64}-1$
Real				
`float`	`Single`	`F`	32 bits	$\pm$ (~10^{-45} to 10^{38})
`double`	`Double`	`D`	64 bits	$\pm$ (~10^{-324} to 10^{308})
`decimal`	`Decimal`	`M`	128 bits	$\pm$ (~10^{-28} to 10^{28})

Table 2-1. Predefined numeric types in C#

Of the *integral* types, `int` and `long` are first-class citizens and are favored by both C# and the runtime. The other integral types are typically used for interoperability or when space efficiency is paramount.

Of the *real* number types, `float` and `double` are called *floating-point types*[2] and are typically used for scientific and graphical calculations.

The `decimal` type is typically used for financial calculations, where base-10-accurate arithmetic and high precision are required.

Overflow

At runtime, arithmetic operations on integral types can overflow. By default, this happens silently—no exception is thrown, and the result exhibits "wraparound" behavior, as though the computation was done on a larger integer type and the extra significant bits discarded. For example, decrementing the minimum possible `int` value results in the maximum possible `int`value:

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // True
```

Overflow check operators

The `checked` operator tells the runtime to generate an `OverflowException` rather than overflowing silently when an integral-type expression or statement exceeds the arithmetic limits of that type. The `checked` operator affects expressions with the ++, −−, +, − (binary and unary), `*`, `/`, and explicit conversion operators between integral types.
`checked` can be used around either an expression or a statement block. For example:

```
int a = 1000000;
int b = 1000000;

int c = checked (a * b);       // Checks just the
expression.

checked                        // Checks all expressions
{                              // in statement block.
   ...
   c = a * b;
   ...
}
```

Arrays

An array represents a fixed number of variables (called *elements*) of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly efficient access.
An array is denoted with square brackets after the element type. For example:

```
char[] vowels = new char[5];    // Declare an array of
5 characters
```

Square brackets also *index* the array, accessing a particular element by position:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine (vowels[1]);      // e
```

This prints “e” because array indexes start at 0. We can use a `for` loop statement to iterate through each element in the array. The `for` loop in this example cycles the integer `i` from `0` to `4`:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels[i]);            // aeiou
```

The `Length` property of an array returns the number of elements in the array. Once an array has been created, its length cannot be changed.
The `System.Collection` namespace and subnamespaces provide higher-level data structures, such as dynamically sized arrays and dictionaries.
An *array initialization expression* lets you declare and populate an array in a single step:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

or simply:

```
char[] vowels = {'a','e','i','o','u'};
```

All arrays inherit from the `System.Array` class, providing common services for all arrays. These members include methods to get and set elements regardless of the array type, and are described in Chapter 7.

Default Element Initialization

Creating an array always preinitializes the elements with default values. The default value for a type is the result of a bitwise zeroing of memory. For example, consider creating an array of integers. Since `int` is a value type, this allocates 1,000 integers in one contiguous block of memory. The default value for each element will be 0:

```
int[] a = new int[1000];
Console.Write (a[123]);            // 0
```

Value types versus reference types

Whether an array element type is a value type or a reference type has important performance implications. When the element type is a value type, each element value is allocated as part of the array. For example:

```
public struct Point { public int X, Y; }
...

Point[] a = new Point[1000];
int x = a[500].X;                  // 0
```

Had `Point` been a class, creating the array would have merely allocated 1,000 null references:

```
public class Point { public int X, Y; }

...
Point[] a = new Point[1000];
int x = a[500].X;                  // Runtime error,
NullReferenceException
```

To avoid this error, we must explicitly instantiate 1,000 `Points` after instantiating the array:

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++) // Iterate i from 0
to 999
   a[i] = new Point();             // Set array element
i with new point
```

An array *itself* is always a reference type object, regardless of the element type. For instance, the following is legal:

```
int[] a = null;
```

Multidimensional Arrays

Multidimensional arrays come in two varieties: *rectangular* and *jagged*. Rectangular arrays represent an *n*-dimensional block of memory, and jagged arrays are arrays of arrays.

Rectangular arrays

Rectangular arrays are declared using commas to separate each dimension. The following declares a rectangular two-dimensional array, where the dimensions are 3 by 3:

```
int[,] matrix = new int[3,3];
```

The `GetLength` method of an array returns the length for a given dimension (starting at 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)//rows
  for (int j = 0; j < matrix.GetLength(1); j++)//colums
    matrix[i,j] = i * 3 + j;
```

A rectangular array can be initialized as follows (to create an array identical to the previous example):

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

Jagged arrays

Jagged arrays are declared using successive square brackets to represent each dimension. Here is an example of declaring a jagged two-dimensional array, where the outermost dimension is 3:

```
int[][] matrix = new int[3][];
```

The inner dimensions aren't specified in the declaration because, unlike a rectangular array, each inner array can be an arbitrary length. Each inner array is implicitly initialized to null rather than an empty array. Each inner array must be created manually:

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int[3];                    // Create
inner array
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

A jagged array can be initialized as follows (to create an array identical to the previous example with an additional element at the end):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}

};
```

Simplified Array Initialization Expressions

There are two ways to shorten array initialization expressions. The first is to omit the `new` operator and type qualifications:

```
char[] vowels = {'a','e','i','o','u'};

int[,] rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
```

```
    {6,7,8}
};

int[][] jaggedMatrix =
{
    new int[] {0,1,2},
    new int[] {3,4,5},
    new int[] {6,7,8}
};
```

Default Values

All type instances have a default value. The default value for the predefined types is the result of a bitwise zeroing of memory:

Type	Default value
All reference types	`null`
All numeric and enum types	`0`
`char` type	`'\0'`
`bool` type	`false`

You can obtain the default value for any type with the `default` .

```
decimal d = default (decimal);
```

The default value in a custom value type (i.e., `struct`) is the same as the default value for each field defined by the custom type.

Parameters

A method has a sequence of parameters. Parameters define the set of arguments that must be provided for that method. In this example, the method `Foo` has a single parameter named `p`, of type `int`:

```
static void Foo (int p)
{
    p = p + 1;                  // Increment p by 1
    Console.WriteLine (p);      // Write p to screen
```

```
  }

  static void Main()
  {
    Foo (8);                    // Call Foo with an
  argument of 8
  }
```

You can control how parameters are passed with the `ref` and `out` modifiers:

Parameter modifier	Passed by	Variable must be definitely assigned
(None)	Value	Going *in*
`ref`	Reference	Going *in*
`out`	Reference	Going *out*

Passing arguments by value

By default, arguments in C# are *passed by value*, which is by far the most common case. This means a copy of the value is created when passed to the method:

```
  class Test
  {
    static void Foo (int p)
    {
      p = p + 1;                // Increment p by 1
      Console.WriteLine (p);    // Write p to screen
    }

    static void Main()
    {
      int x = 8;
      Foo (x);                  // Make a copy of x
      Console.WriteLine (x);    // x will still be 8
    }
  }
```

Assigning p a new value does not change the contents of x, since p and x reside in different memory locations.

Passing a reference-type argument by value copies the *reference*, but not the object. In the following example, Foo sees the same StringBuilder object that Main instantiated, but has an independent *reference* to it. In other words, sb and fooSB are separate variables that reference the same StringBuilder object:

```
class Test
{
  static void Foo (StringBuilder fooSB)
  {
    fooSB.Append ("test");
    fooSB = null;
  }

  static void Main()
  {
    StringBuilder sb = new StringBuilder();
    Foo (sb);
    Console.WriteLine (sb.ToString());    // test
  }
}
```

Because fooSB is a *copy* of a reference, setting it to null doesn't make sb null. (If, however, fooSB was declared and called with the ref modifier, sb *would* become null.)

The ref modifier

To *pass by reference*, C# provides the ref parameter modifier. In the following example, p and x refer to the same memory locations:

```
class Test
{
  static void Foo (ref int p)
  {
    p = p + 1;                // Increment p by 1
    Console.WriteLine (p);    // Write p to screen
  }
```

```
  static void Main()
    {
      int x = 8;
      Foo (ref  x);             // Ask Foo to deal directly
  with x
      Console.WriteLine (x);    // x is now 9
    }
  }
```

Now assigning `p` a new value changes the contents of `x`. Notice how the `ref` modifier is required both when writing and when calling the method.[4] This makes it very clear what's going on.

The `ref` modifier is essential in implementing a swap method

```
  class Test
  {
    static void Swap (ref string a, ref string b)
    {
      string temp = a;
      a = b;
      b = temp;
    }

    static void Main()
    {
      string x = "Penn";
      string y = "Teller";
      Swap (ref x, ref y);
      Console.WriteLine (x);    // Teller
      Console.WriteLine (y);    // Penn
    }
  }
```

The out modifier

An `out` argument is like a `ref` argument, except it:

- Need not be assigned before going into the function
- Must be assigned before it comes *out* of the function

The `out` modifier is most commonly used to get multiple return values back from a method. For example:

```
class Test
{
  static void Split (string name, out string
firstNames, out string lastName)
  {
     int i = name.LastIndexOf (' ');
     firstNames = name.Substring (0, i);
     lastName   = name.Substring (i + 1);
  }

  static void Main()
  {
    string a, b;
    Split ("Stevie Ray Vaughan", out a, out b);
    Console.WriteLine (a);                        //
Stevie Ray
    Console.WriteLine (b);                        //
Vaughan
  }
}
```

Like a `ref` parameter, an `out` parameter is passed by reference.

Out variables and discards (C# 7)

From C# 7, you can declare variables on the fly when calling methods with `out` parameters. We can shorten the `Main` method in our preceding example as follows:

```
static void Main()
{
  Split ("Stevie Ray Vaughan", out string a, out string
b);
  Console.WriteLine (a);                          // Stevie
Ray
  Console.WriteLine (b);                          //
Vaughan
}
```

When calling methods with multiple `out` parameters, sometimes you're not interested in receiving values from all the parameters. In such cases, you can "discard" the ones you're uninterested in with an underscore:

```
Split ("Stevie Ray Vaughan", out string a, out _);   //
Discard the 2nd param
Console.WriteLine (a);
```

In this case, the compiler treats the underscore as a special symbol, called a *discard*. You can include multiple discards in a single call. Assuming `SomeBigMethod` has been defined with seven **`out`** parameters, we can ignore all but the fourth as follows:

```
SomeBigMethod (out _, out _, out _, out int x, out _,
out _, out _);
```

For backward compatibility, this language feature will not take effect if a real underscore variable is in scope:

```
string _;
Split ("Stevie Ray Vaughan", out string a, _);   //
Will not compile
```

Implications of passing by reference

When you pass an argument by reference, you alias the storage location of an existing variable rather than create a new storage location. In the following example, the variables `x` and `y` represent the same instance:

```
class Test
{
  static int x;

  static void Main() { Foo (out x); }

  static void Foo (out int y)
  {
    Console.WriteLine (x);                // x is 0
    y = 1;                                // Mutate y
    Console.WriteLine (x);                // x is 1
  }
}
```

The params modifier

The `params` parameter modifier may be specified on the last parameter of a method so that the method accepts any number of arguments of a

particular type. The parameter type must be declared as an array. For example:

```
class Test
{
  static int Sum (params int[] ints)
  {
    int sum = 0;
    for (int i = 0; i < ints.Length; i++)
      sum += ints[i];                          // Increase
sum by ints[i]
    return sum;
  }

  static void Main()
  {
    int total = Sum (1, 2, 3, 4);
    Console.WriteLine (total);                 // 10
  }
}
```

You can also supply a `params` argument as an ordinary array. The first line in `Main` is semantically equivalent to this:

```
int total = Sum (new int[] { 1, 2, 3, 4 } );
```

Optional parameters

From C# 4.0, methods, constructors, and indexers (Chapter 3) can declare *optional parameters*. A parameter is optional if it specifies a *default value* in its declaration:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

Optional parameters may be omitted when calling the method:

```
Foo();      // 23
```

The *default argument* of `23` is actually *passed* to the optional parameter `x`—the compiler bakes the value 23 into the compiled code at the *calling* side. The preceding call to `Foo` is semantically identical to:

```
Foo (23);
```

because the compiler simply substitutes the default value of an optional parameter wherever it is used.

The default value of an optional parameter must be specified by a constant expression, or a parameterless constructor of a value type. Optional parameters cannot be marked with `ref` or `out`.
Mandatory parameters must occur *before* optional parameters in both the method declaration and the method call (the exception is with `params` arguments, which still always come last). In the following example, the explicit value of `1` is passed to `x`, and the default value of `0` is passed to `y`:

```
void Foo (int x = 0, int y = 0) { Console.WriteLine (x
+ ", " + y); }

void Test()
{
  Foo(1);    // 1, 0
}
```

To do the converse (pass a default value to `x` and an explicit value to `y`) you must combine optional parameters with *named arguments*.

Named arguments

Rather than identifying an argument by position, you can identify an argument by name. For example:

```
void Foo (int x, int y) { Console.WriteLine (x + ", " +
y); }

void Test()
{
  Foo (x:1, y:2);  // 1, 2
}
```

Named arguments can occur in any order. The following calls to `Foo` are semantically identical:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

You can mix named and positional arguments:

```
Foo (1, y:2);
```

However, there is a restriction: positional arguments must come before named arguments. So we couldn't call `Foo` like this:

```
Foo (x:1, 2);          // Compile-time error
```

Named arguments are particularly useful in conjunction with optional parameters. For instance, consider the following method:

```
void Bar (int a = 0, int b = 0, int c = 0, int d = 0) {
... }
```

We can call this supplying only a value for `d` as follows:

```
Bar (d:3);
```

This is particularly useful when calling COM APIs.

Ref Locals (C# 7)

C# 7 adds an esoteric feature, whereby you can define a local variable that *references* an element in an array or field in an object:

```
int[] numbers = { 0, 1, 2, 3, 4 };

ref int numRef = ref numbers [2];
```

In this example, `numRef` is a *reference* to the `numbers[2]`. When we modify `numRef`, we modify the array element:

```
numRef *= 10;
```

```
Console.WriteLine (numRef);         // 20
Console.WriteLine (numbers [2]);    // 20
```

The target for a ref local must be an array element, field, or local variable; it cannot be a *property* . *Ref locals* are intended for specialized micro-optimization scenarios, and are typically used in conjunction with *ref returns*.