Main

Some variations of the Main method.

```
static void Main() { }
static int Main() { return 0; }
static int Main(string[] args) { return 0; }
```

Command-line args can also be retrieved using `System.Environment.GetCommandLineArgs()`.

Console I/O

```
Console.WriteLine();
Console.ReadLine();
```

Strings

Formatting

```
string.Format("{0} {1} {2}", a, b, c);
string.Format("{0:d9}", someNumber);
```

The second example shows how specific formatting can be applied. That is, limiting decimal place precision, padding, etc.

Verbatim Strings

With "verbatim" strings, no escaping is done (\n, \t, etc).

```
string v = @"This is a ""verbatim"" string. Backslashes, \, are not escaped.";
```

Interpolated Strings

.NET 4.6+

With "interpolated" strings, variables can be referenced in the string and filtered.

```
string p = $"myVariable has value {myVariable | expression}."
```

Arrays

Various ways to initialize.

```
int[] a = {1, 2, 3};
int[] b = new int[] {1, 2, 3};
int[] c = new int[3];
```

Implicitly-typed

```
var a = new[] {1, 2, 3};
```

Matrix

```
int[,] mat = new int[3, 4];
```

Jagged Array

```
int[][] jag = new int[3][];
for (int i = 0; i < 3; ++i)
{
  jag[i] = new int[4];
}
```

Parameters

Modifiers

- `out` - method *must* assign a value to the argument before returning. Not necessary for caller to initialize variable.
- `ref` - method *may* assign a value to the argument. Necessary for caller to initialize variable.
- `params` - list of parameters.

Example 1 (out)

Definition:

```
void Add(int a, int b, out int sum)
{
  sum = a + b;
}
```

Invocation:

```
int sum;
Add(4, 5, out sum);
```

Example 2 (ref)

Definition:

```
void Swap(ref Card a, ref Card b)
{
  Card tmp = a;
  a = b;
  b = tmp;
}
```

Invocation:

```
Card a = new Card("KD");
Card b = new Card("9D");
Swap (ref a, ref b);
```

Example 3 (params)

Definition:

```
int Sum(params int[] nums)
```

```
{
  int sum = 0;
  foreach (int i in nums)
  {
    sum += i;
  }
  return sum;
}
```

Invocation:

```
int result = Sum(1, 2, 3, 4);
```

Default Parameters

```
void LogEvent(string msg, string tag = "Default", Color color = Color.blue);
```

You can use **named parameters** to feed arguments out of order.

```
LogEvent(tag: "Timing");
```

Unnamed, ordered parameters must come first though.

```
LogEvent(msg, color: Color.red);
```

You can also use default parameters in constructors above .NET 4.0.

Enums

```
enum MyEnum { Yo }
```

```
enum MyEnum
{
  Three = 3, // Will start with value 3.
  Four,
  Five
}
```

By default, enums use the `System.Int32` (C# `int`) type. You can change this as follows.

```
enum MyEnum : byte
{
  MyByte
}
```

You can get the name as a string or raw value like this.

```
MyEnum.Three.ToString(); // "Three"
(int)MyEnum.Three; // 3
```

Value Types

- ValueTypes are extended from `Object`.
- ValueTypes declare their memory on the stack instead of the heap. This means that a ValueType will be deallocated on exiting its scope in which it was defined. In contrast to the managed heap where it would have to wait to be garbage collected.
- Primitive types, enums, and structs are implemented using ValueTypes.
- ValueTypes are copied member-by-member when assigned to a variable or passed as an argument. If a member is a ReferenceType (what classes use) then only its reference will be copied unless `IClonable` is implemented by the ReferenceType.

Structs

Structs cannot extend classes, or override the default constructor.

Quirk: If you have private fields and properties in a struct with a custom constructor, then you need to call the default constructor in all custom constructors to initialize the fields with default values.

```
struct Point
{
  private int X { get; set; }
  private int Y { get; set; }
  private string label;

  public Point(string label) : this()
  {
    this.label = label;
  }
}
```

Nullable

ValueTypes cannot be assigned null. They must be wrapped in Nullable.

```
System.Nullable<int> i = null;
```

There is a convenient shorthand for this (?).

```
int? i = null;
bool? b = null;
float? t = 0.1f;
```

Null Operators

Null Coalescing Operator (??)

Returns an alternate value on null.

```
Card cardB = cardA ?? new Card("2C"); // If cardA is null, return new Card.
```

Null Conditional Operator (?.)

Provides safe method access. Also known as "Elvis operator."

```
Card cardA = deckA?.Get(1);
```

If `deckA` turns out to be null, it will return null and not call `Get()`.

Constructors

Chaining

```
public Card(char rank, char suit) { } // Master constructor.
public Card() : this('2', 'C') { } // Use master constructor.
```

Static

You can initialize static variables with the static constructor.

```
static MyStaticConstructor()
{
  // No access modifiers allowed.
}
```

This is called only once on first instantiation of the class, or on static member access.

Static Import

You can import static members with `using static`.

```
using static ClassWithStaticMembers;
```

Access Modifiers

- `public` - Open to everyone.
- `private` - Only members of the class.
- `protected` - private + children.
- `internal` - Only within the assembly.
- `protected internal` - Protected and within the assembly (no external children).

- Types are implicitly internal.
- Members are implicitly private.

Properties

A basic .NET property:

```
int myInt;

public int SomeInt
{
  get
  {
    return myInt;
  }
  set
  {
    myInt = value;
  }
}
```

Auto-Properties

```
public int SomeInt { get; set; }
```

This uses a private field internally and sets that to a default value.

For ReferenceTypes, the default value is null; which can be a problem. In many cases, the default value will not be what you want. The solution to this is to assign a default value in the constructor. Alternatively, C# 6.0 includes a special syntax.

```
public string SomeString { get; set; } = "Default value.";
```

Object Initialization Syntax

You can use object-literal notation in constructing an object.

```
Card c = new Card { Rank = '5', Suit = 'D' };
Card c = new Card() { Rank = '5', Suit = 'D' }; // Equivalent to above.
Card c = new Card(suit: 'D') { Rank = '5' }; // You can get crazy with this.
```

The names in the brackets are public fields of the Card class, not the names of the parameters in the constructor.

Read-Only Fields

- A `const` field must be assigned at the time of the declaration.
- A `readonly` field can be assigned at declaration or in the constructor.

`const` is implicitly static.

```
public const int PI = 3.1415f;
```

`readonly` is an instance variable.

```
public readonly DateTime INITIALIZED;

public MyBrand()
{
  INITIALIZED = DateTime.Now;
}
```

`static` can be used with `readonly`, which is almost like `const`

```
public static readonly DateTime FIRST_INITIALIZATION;

static MyBrand()
{ // The difference is you can use a static constructor to initialize it.
  FIRST_INITIALIZATION = DateTime.Now;
}
```

Partial classes

The `partial` keyword allows classes to be split among multiple files. The file names don't matter, only that the class name and namespace are the same.

This is one way you could split things up:

```
// Cookie.cs
partial class Cookie
{
  // Methods
  // Properties
}
```

```
// Cookie.Boilerplate.cs
partial class Cookie
{
  // Field data
  // Constructors
}
```

The 'Sealed' Keyword

This keyword prevents a class from being extended.

```
sealed class MySealedClass { }
```

The 'Base' Keyword

Just like Java's `super`, C# has the `base` keyword to call a parent ctor.

```
public MyClass(int a, string b) : base(a)
{
  this.b = b;
}
```

You of course can use `base` to access members of the base class as well.

Polymorphism

You can mark a method as overrideable with the `virtual` keyword.

```
public virtual void OverrideableMethod() { }

// Then, in the child class:
public override void OverrideableMethod() { }
```

You can call the parent method with:

```
base.OverrideableMethod();
```

You can prevent a method from further being overridden by sealing it.

```
public override sealed void OverrideableMethod() { }
```

Abstract Classes

```
abstract class MyAbstractClass
{
  public abstract void AbstractMethod();
}

class AnotherClass : MyAbstractClass
{
  public override void AbstractMethod() { /* ... */ }
}
```

Class Cast Checking

The 'AS' Keyword

Null if not type.

```
Deck d = someObject as Deck;
if (d != null) d.Shuffle();
```

The 'IS' Keyword

False if not type.

```
if (someObject is Deck)
{
  ((Deck)someObject).Shuffle();
}
```

Shadowing

Instead of overriding, you can completely replace (shadow) the parent method, property or field.

```
public new int MyProperty { get; set; }
public new void MyMethod() { }
```

You can still access the parent's original member by explicit casting.

```
((Parent)child).MyMethod();
```

Overflow Checking

You can use the `checked` keyword to carry out a checked cast, which checks for numeric overflow when narrowing a scope.

Block form:

```
checked
{
  byte a = (byte)someIntegerA;
  byte b = (byte)someIntegerB;
}
```

Inline form:

```
checked(byte a = (byte)someIntegerA);
```

If you can enabled overflow checking project-wide, you can use the `unchecked` keyword in a similar fashion to ignore checking.

The project-wide setting is usually in "Build Settings > Check for arithmetic overflow/underflow".