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);
```

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()`.

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".