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)

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

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

Example 2 (ref)

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

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

Example 3 (params)

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

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

```
static MyStaticConstructor() { }
```

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

Properties are an implementation of the accessor/mutator (getter/setter) pattern.

A basic .NET property:

```
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() { /* ... */ }
}
```

Interfaces

C# interfaces are similar to Java interfaces.

```
public interface IClashing
{
  public void ClashingMethod();
}
```

If by chance you have methods in different interfaces that clash, you can define them explicitly.

```
class SomeClass : IClashing, IAlsoClashing
{
  public void IClashing.ClashingMethod() { }
  public void IAlsoClashing.ClashingMethod() { }
}
```

You need to explicitly cast to the specific interface type to use a clashing method.

```
(someClass as IClashing)?.ClashingMethod();

IClashing ic = ((IClashing)someClass);
if (ic) ic.ClashingMethod();
```

If you just define the unqualified clashing method, it will override all of them.

ICloneable

C#'s object class defines a protected `MemberwiseClone()` method that copies an object by copying the members.

To perform a deep copy you can utilize `ICloneable`.

```
public interface ICloneable
```

```
{
  object Clone(); // Just create a new object and copy the values yourself.
}
```

IComparable and IComparer

`IComparable` is notable utilized by `Arrays.Sort`.

```
public interface IComparable
{
  int CompareTo(object o); // -1 (lt), 0 (eq), 1 (gt).
}
```

`IComparer` is typically defined on a helper class, (e.g., ISBNComparer.Compare(Book a, Book b)). `Arrays.Sort` also can use this: `Arrays.Sort(books, new ISBNComparer())`.

```
public interface IComparer
{
  int Compare(object a, object b);
}
```

IEnumerable, IEnumerator, Iterators, and Yield

Note: these examples don't use the generic versions. Use the generic versions to prevent penalties incurred by auto-boxing.

```
public interface IEnumerable
{
  IEnumerator GetEnumerator(); // Used by the foreach loop.
}

public interface IEnumerator
{
  bool MoveNext(); // Advance the cursor position.
  object Current() { get; } // Get the current item.
  void Reset(); // Reset the cursor.
}
```

```
public class SomeClass : IEnumerable
{
  // ...
  // This is a special "iterator" method.
  public IEnumerator GetEnumerator()
  {
    for (var i = 0; i < items.Length; ++i)
    {
         // The compiler automatically generates a nested IEnumerator that
         // keeps track of execution, and returns items[i] for Current.
         yield return items[i];
         // You cannot call Reset on an instance of this generated IEnumerator,
         // you'll have to get a new instance of it.
    }
  }

  // This is a "named iterator" method.
  public IEnumerator GetReverseEnumerator(string msg)
  {
    Console.WriteLine("Named iterators can receive parameters: " + msg);
    for (var i = items.Length-1; i >= 0; --i)
    {
      yield return items[i];
    }
  }
}
```

You can use the iterators in a `foreach` loop.

```
foreach (Item it in someClass) {}
foreach (Item it in someClass.GetReverseEnumerator()) {}
```

The 'Yield' Keyword

When you use the yield keyword in a statement, you indicate that the method, operator, or "get" accessor in which it appears is an iterator. Using yield to define an iterator removes the need for an explicit extra class (the class that holds the state for an enumeration, see IEnumerator)

- `yield return` - return each element one at a time.
- `yield break` - end iteration.

You consume an iterator by using a foreach statement or LINQ query. Each iteration of the foreach loop calls the iterator. When a `yield return` statement is reached in the iterator method, the expression is returned, and the current location in code is retained. Execution is restarted from that location the next

time that the iterator function is called.

`yield` is not a reserved word and has special meaning only when it is used before a `return` or `break` keyword.

An iterator method cannot have any `ref` or `out` parameters.

An implicit conversion must exist from the expression type in the `yield return` statement to the return type of the iterator.

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();
}
```

Boxing

Every time you assign a `ValueType` to a `ReferenceType`, e.g.,

```
int f = 5;
object oa = f;
```

C# does a 'box' operation: allocates an object on the heap and moves the value from the stack to the new

object.

C# does an 'unbox' operation to do the inverse, and requires casting:

```
int g = (int)oa;
```

This is very inefficient speed-wise and memory-wise.

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();
```

Exceptions

Fatal system exceptions extend `SystemException` and are thrown by the CLR. Extend `ApplicationException`, or just `Exception`, for app-related exceptions.

You don't have to specify an exception in try-catch.

```
try { }
catch { }
```

If you do, the syntax is the same as in Java.

```
try { }
catch (Exception e) { }
```

You can rethrow an exception by just calling throw in the catch block.

```
try {}
catch
{
  // Stuff.
  throw;
}
```

Throwing exceptions is the same as in Java too.

```
throw new SomeException();
```

If an exception happens while handling another, the proper practice is to shove the inner exception in a new one of the parent type.

```
try { }
catch (CustomException ce)
{
  try { }
  catch (Exception ie)
  {
    throw new CustomException(ce.Message, ie);
  }
}
```

Exception Filters

Use `when` to conditionally run code in a catch block.

```
try {}
catch (Exception e) when (isDebuggingOn) {}
```

Generics

Generics provide better performance because they prevent boxing or unboxing penalties when storing ValueTypes.

```
// Generic method.
void AddToList<T>(T item)
```

```
{
  Console.Log("Adding item of type " + typeof(T));
}

// Generic struct.
public struct Point<T>
{
  public void ResetPoint()
  {
    x = default(T); // Sets the default value w.r.t. the generic type.
    y = default(T);
  }
}
```

Generic Constraints

Generic class using constraints on the generic type.

```
public class MyGenericClass<T> where T : class, IDrawable, new()
{ }
```

Possible Constraints:

- `struct` - ValueTypes
- `class` - ReferenceTypes
- `new()` - Must have a default ctor. Must be last in the list.
- `MyCustomType`
- `IMyInterface` - Can have multiple interface constraints.
- `U` - Depends on generic type 'U' (generic param. U must also specified). Does not have to be letter 'U' of course.

```
class MyGeneric<T, U>
  where T : Set<U>, ISomeInterface
  where U : struct
  { }
```

Notice the `where TypeParam : [constraints]` sections are *not* separated by commas.

The System.Collection.Generics Namespace

Generic Interfaces

- `ICollection<T>` - Defines general characteristics (e.g., size, enumeration, and thread saftey) for all generic collection types.
- `IComparer<T>`
- `IDictionary<T>` - Allows a generic collection object to represent its contents using key-value pairs.
- `IEnumerable<T>`
- `IEnumerator<T>`
- `IList<T>`
- `ISet<T>`

Generic Collections

- `Dictionary<TKey, TValue>`
- `LinkedList<T>`
- `List<T>`
- `Queue<T>`
- `SortedDictionary<TKey, TValue>`
- `SortedSet<T>`
- `Stack<T>`

The System.Collections.ObjectModel Namespace

This contains a couple of important collection objects that notify listeners when the collection is modified.

- `ObservableCollection<T>`
- `ReadOnlyObservableCollection<T>`

Delegates

```
// Declaring a delegate type.
public delegate string MyDelegate(bool b, int n);

// Declaring and initializing a delegate variable.
MyDelegate mdel = new MyDelegate(SomeMethod);

// Alternatively, you can just supply the method name.
// This is called "method group conversion syntax".
MyDelegate mdel = SomeMethod;

// Delegates can be combined to call multiple methods.
mdel += AnotherMethod;
```

```
mdel += OrAnotherDelegate;

// Invoking the delegate.
mdel();
```

Generic Delegates

```
public delegate void GenericDel<T>(T arg);

public void MyFunc(string a) { }
GenericDel<string> strTarget = MyFunc;
```

Action<...> and Func<...>

These two are out-of-the-box, generic delegate types for simplifying the definition of a delegate.

Action<...>

- Takes up to 16 arguments.
- Can only point to functions with a `void` return type.

```
static void DisplayMessage(string msg, ConsoleColor col) { }
Action<string, ConsoleColor> target = DisplayMessage;
target("hello", ConsoleColor.Blue);
```

As you can see in the example, the types in the generic parameter list define the types of the parameters the delegate methods take in.

Func<...>

- Takes up to 16 arguments.
- The last generic parameter is the return type.

```
static string IntToString(int a) { }
Func<int, string> target = IntToString;
string s = target(5);
```

Predicate

Defined in the System as such:

```
public delegate bool Predicate<T>(T obj);
```

This is used in places like `List.FindAll()`.

Events

Events are a special kind of multicast delegate that can only be invoked from within the class or struct where they are declared (the publisher class).

```
public delegate string MyDelegate(int a);
public event MyDelegate MyEvent;
```

EventArgs

Microsoft's recommended pattern for how to publish data with a delegate.

```
public delegate void MyDelegate(object sender, MyEventArgs e);
public event MyDelegate MyEvent;

public class MyEventArgs
{
  public MyEventArgs(string s) { Text = s; }
  public string Text { get; private set; } // readonly
}

// Variation
public class MyEventArgs
{
  public MyEventArgs(string s) { text = s; }
  public readonly string text;
}

// Now when you raise an event to listeners. (In the publisher class...)
MyEvent(this, new MyEventArgs("Hey, this is some event data."));
```

EventHandler

Given the recommended (object sender, EventArgs e) pattern, there is a generic event type Microsoft provides to streamline this.

```
public event EventHandler<MyEventArgs> MyEvent;
```

You don't need to define the delegate in this case.

Delegates: Behind the Scenes

The compiler generates a class for a delegate type.

```
sealed class MyDelegate : System.MulticastDelegate
{
  // Basic synchronous invocation.
    public string Invoke(bool b, int n);

    // The other two are for asynchronous use.
    public IAsyncResult BeginInvoke(bool b, int b, AsyncCallback cb, object state);
    public string EndInvoke(IAsyncResult result);
}
```

The `MulticastDelegate` class is defined as such:

```
public abstract class MulticastDelegate : Delegate
{
    // Returns the list of methods "pointed to"
    public sealed override Delegate[] GetInvocationList();

    // Overloaded operators.
    public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
    public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);

    // Used internally to manage the list of methods maintained by the delegate.
    private IntPtr _invocationCount;
    private object _invocationList;
}
```

The `Delegate` class is defined as such:

```
public abstract class Delegate : ICloneable, ISerializable
{
    // Methods to interact with the list of functions.
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
```

```
    public static Delegate RemoveAll(Delegate source, Delegate value);

    // Overloaded operators.
    public static bool operator ==(Delegate d1, Delegate d2);
    public static bool operator !=(Delegate d1, Delegate d2);

    // Properties that expose the delegate target.
    public MethodInfo Method { get; }
    public object Target { get; }
}
```

- The '+=' operator is syntax sugar for the Combine() method.
- The '-=' operator is syntax sugar for the Remove() method.
- 'Method' gets target method details.
- 'Target' gets object details if the target method belongs to an instance.

Anonymous Methods

```
someDelegateOrEvent += delegate {
  Console.WriteLine("Anonymous method which ignores arguments.");
}

someDelegateOrEvent += delegate(int a, string b) {
  Console.WriteLine("Anonymous method which accepts arguments.");
}
```

- Cannot access `ref` or `out` parameters in the outer method.
- Cannot have local variables that are the same as in the outer method.
- Stuff in the outer class scope works as expected.

Lambda Expressions

Various forms of lambda expressions.

```
() => Console.WriteLine("Hello.");

i => (i % 2) == 0

(i) => (i % 2) == 0

(int i) => (i % 2) == 0
```

```
(a, b) => a + b

(a) => {
  Console.WriteLine(a);
  return a + 1;
}
```

As of .NET 4.6, you can use lambdas for member methods too.

```
class MyClass
{
  public int Add(int x, int y) => x + y;
}
```

Indexer Methods

Basic implementation.

```
public ReturnType this[int i]
{
  get { return (ReturnType)internalList[i]; }
  set { internalList[i] = value; }
}
```

Can use indexing by non-integer types too.

```
public ReturnType this[string s] {
  /* ... */
}
```

Multidimensional indexing.

```
public ReturnType this[string a, string b] {
  /* ... */
}
```

These can be overloaded so you can use multiple methods of indexing with your custom type.

You can also specify them in an interface.

```
public interface IMyContainer
{
  string this[int index] { get; set; }
}
```

Operator Overloading

Various operators in C# can be overloaded.

```
public static Point operator + (Point p1, Point p2)
{
  return new Point(p1.x + p2.x, p1.y + p2.y);
}
```

With binary operators, you get the corresponding assignment operator for free. For example, with the above example you also get '+='.

Both operands don't have to be the same type.

```
public static Point operator * (Point p1, float scalar)
{
  return new Point(p1.x * scalar, p1.y * scalar);
}
```

If you do this though, do it both ways to support commutativity.

```
public static Point operator * (float scalar, Point p1)
{
  return new Point(p1.x * scalar, p1.y * scalar);
}
```

You can override some unary operators, such as ++.

```
public static Point operator ++ (Point p)
{
  return new Point(p.x + 1, p.y + 1);
}
```

In C++ you can override the pre/post increment/decrement operators separately. In C# you can't, but you get the correct behavior for free.

Conversion Methods

Explicit.

```
public static explicit operator Square(Rectangle r)
{
  return new Square(r.Height);
}
```

You can now explicitly cast from Rectangle to Square.

```
Rectangle myRectangle = new Rectangle(4, 4);
Square mySquare = (Square)myRectangle;
```

Implicit.

```
public static implicit operator Rectangle(Square s)
{
  return new Rectangle(s.Length, s.Length);
}
```

You can not implicitly cast from Square to Rectangle.

```
Square mySquare =  new Square(4);
Rectangle myRectangle = mySquare;
```

In the implicit case, you also get the explicit cast for free. This makes sense because if it happens automatically, you should also be allowed to be verbose about it.

```
Rectangle myRectangle2 = (Rectangle)mySquare;
```

Conversion methods can be defined in structs as well.

Extension Methods

- Extension methods must be defined in a static class, and must be static.
- These add methods to existing types.
- The target type is specified as the first parameter.
- Additional parameters are allowed.

```
using System.Reflection;

static class MyExtensions
{
  // Targeting the Rectangle type.
  public static bool IsSquare(this Rectangle r)
  {
    return r.Height == r.Width;
  }

  // Targeting the int type.
  public static int Add(this int a, int b)
  {
    return a + b;
  }

  // You can also define extension methods targeting interfaces.
  public static void PrintAll(this System.Collections.IEnumerable iterator)
  {
    foreach (var item in iterator)
    {
      Console.WriteLine(item);
    }
  }
}
```

```
int myInt = 30;
int result = myInt.Add(2); // Using an extension method from MyExtensions.
```

LINQ (to Objects)

- A C# query language that allows unified data access to various types of data.
- LINQ (Language Integrated Query) appears similar to SQL.

- Based on what data is being targeted, people use different phrases:
  - LINQ to Objects (arrays and collections)
  - LINQ to XML
  - LINQ to DataSet (ADO.NET type)
  - LINQ to Entities (ADO.NET Entity Framework [EF])
  - Parallel LINQ (aka PLINQ) (parallel processing of data)
- Available in .NET 3.5 and up.
- Must include 'System.Linq' for core LINQ usage (in System.Core.dll).

```
string[] games = {"Skyrim", "Fallout 4", "League of Legends"};

IEnumerable<string> subset =
  from game in games
  where game.Contains(" ")
  orderby game
  select game;
```

It's recommended to use implicit typing instead of `IEnumerator<T>`.

```
var subset =
  from game in games
  where //...
```

More query examples:

```
// Most basic.
var res = from n in numList select n;

// Using many of the operators.
var res = from n in numList where n>0 && n<10 orderby n descending select n;

// Projection: return a subset type. In this case as an anonymous type.
var res = from book in bookList select new {book.Title, book.Author};
```

- LINQ queries return various types which all implement `IEnumerator<T>`.
- *Deferred execution*: LINQ results are not evaluated until you actually iterate over the sequence.
- *Immediate execution*: You can return a snapshot of the result sequence with extension methods provided by LINQ. Some are: `ToArray<T>()`, `ToDictionary<TSource, TKey>()`, and `ToList<T>()`.
- Other operators are (`join`, `on`, `equals`, `into`, `group`, `by`).
- There's also aggregation and set extension methods too:

    - `Count()`, `Reverse()`, `Intersect(otherRes)`, `Union(otherRes)`, `Concat(otherRes)`, `Distinct()`, `Max()`, `Min()`, `Average()`, `Sum()`.
- LINQ queries are shorthand and actually implemented by a bunch of extension methods (i.e., `Where(Func<T> fn)`, `Select(Func<T> fn)`, etc).

Also, if you have a non-generic collection (like `ArrayList`) you can use `OfType<T>()` to convert (and filter) it to a generic collection.

```
ArrayList myList = new ArrayList();
myList.AddRange(new object[] { 4, "g", new Pineapple(), 88 });

// Create List<T> from integers in myList.
List<int> myList2 = myList.OfType<int>();
```

Anonymous Types

```
var anon = new { Title = "Hello", Author = "Unknown" };
```

- Anonymous types override `ToString()`, `GetHashCode()`, and `Equals()` to perform value-based equality checking.
- The `==` operator compares by reference though.
- Properties of an anonymous type are read-only.

Attributes

```
[Serializable]
public class MyClass
{
  private int myField;

  [NotSerialized]
  private string myBrand;
}
```

C# Attribute Shorthand

By convention, attribute classes are suffixed with 'Attribute'. For example, `SerializableAttribute`. However, C# allows you to leave out the word 'Attribute' when referring to it with the `[...]` syntax.

Object Lifetime Stuff

- `Finalize()` is called when an object is garbage collected.
  - In C# you define this like a C++ destructor `~MyClass()` instead of overriding `Finalize()`.
  - This is not usually implemented.
- `IDisposable` defines the `Dispose()` method.
  - This is called manually by the client when finished to release resources.
- `Lazy<T>` is a wrapper class that implements lazy initialization.

Pointer Types

Though rarely used in C# developement, you can actually use pointers.

To use you need to:

- Define the `/unsafe` flag on compilation: `csc /unsafe *.cs`
- Enable the 'Allow unsafe code' option in Visual Studio.
- Use the unsafe keyword for blocks and methods.

'unsafe'

```
class Program
{
  static void Main(string[] args)
  {
    unsafe
    {
      int myInt = 5;
      Square(&myInt);
    }
  }

  unsafe static void Square(int *n)
  {
    return *n *= *n;
  }
}
```

```
unsafe struct Node { /* ... */ }
```

```
struct Node
{
  public int Value;
  public unsafe Node* next; // Unsafe field.
}
```

Unlike in C and C++, the * operator must be next to the type for pointer declaration, e.g.,

```
public unsafe Node* left, right; // C#
```

instead of,

```
public Node *left, *right; // C, C++
```

'stackalloc'

```
char* p = stackalloc char[256]; // Declare memory on the stack.
```

'fixed'

To prevent a `ReferenceType` from being swept or moved by the GC while you are using its address, use the `fixed` keyword.

```
fixed (MyRefType* p = &refVar)
{
  Console.Write(p->ToString());
}
```

'sizeof'

```
Console.Write(sizeof(int));

// Must be in an 'unsafe' block for custom types.
```

```
unsafe { Console.Write(sizeof(Point)); }
```

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".