```
// Single-line comments start with //
/*
Multi-line comments look like this
*/
/// <summary>
/// This is an XML documentation comment which can be used to
    generate external
/// documentation or provide context help within an IDE
/// </summary>
/// <param name="firstParam">This is some parameter documentation
    for firstParam</param>
/// <returns>Information on the returned value of a function</
    returns>
//public void MethodOrClassOrOtherWithParsableHelp(string firstParam
    ) {}

// Specify the namespaces this source code will be using
// The namespaces below are all part of the standard .NET Framework
    Class Library
using System;
using System.Collections.Generic;
using System.Dynamic;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using System.IO;

// But this one is not:
using System.Data.Entity;
// In order to be able to use it, you need to add a dll reference
// This can be done with the NuGet package manager: `Install-Package
     EntityFramework`

// Namespaces define scope to organize code into "packages" or "
    modules"
// Using this code from another source file: using Learning.CSharp;
namespace Learning.CSharp
{
    // Each .cs file should at least contain a class with the same
        name as the file.
    // You're allowed to do otherwise, but shouldn't for sanity.
    public class LearnCSharp
    {
        // BASIC SYNTAX - skip to INTERESTING FEATURES if you have
            used Java or C++ before
```

```
public static void Syntax()
{
    // Use Console.WriteLine to print lines
    Console.WriteLine("Hello World");
    Console.WriteLine(
        "Integer: " + 10 +
        " Double: " + 3.14 +
        " Boolean: " + true);

    // To print without a new line, use Console.Write
    Console.Write("Hello ");
    Console.Write("World");

    ////////////////////////////////////////////////
    // Types & Variables
    //
    // Declare a variable using <type> <name>
    ////////////////////////////////////////////////

    // Sbyte - Signed 8-bit integer
    // (-128 <= sbyte <= 127)
    sbyte fooSbyte = 100;

    // Byte - Unsigned 8-bit integer
    // (0 <= byte <= 255)
    byte fooByte = 100;

    // Short - 16-bit integer
    // Signed - (-32,768 <= short <= 32,767)
    // Unsigned - (0 <= ushort <= 65,535)
    short fooShort = 10000;
    ushort fooUshort = 10000;

    // Integer - 32-bit integer
    int fooInt = 1; // (-2,147,483,648 <= int <=
       2,147,483,647)
    uint fooUint = 1; // (0 <= uint <= 4,294,967,295)

    // Long - 64-bit integer
    long fooLong = 100000L; // (-9,223,372,036,854,775,808
       <= long <= 9,223,372,036,854,775,807)
    ulong fooUlong = 100000L; // (0 <= ulong <=
       18,446,744,073,709,551,615)
    // Numbers default to being int or uint depending on
       size.
    // L is used to denote that this variable value is of
```

```
    type long or ulong

// Double - Double-precision 64-bit IEEE 754 Floating
    Point
double fooDouble = 123.4; // Precision: 15-16 digits

// Float - Single-precision 32-bit IEEE 754 Floating
    Point
float fooFloat = 234.5f; // Precision: 7 digits
// f is used to denote that this variable value is of
    type float

// Decimal - a 128-bits data type, with more precision
    than other floating-point types,
// suited for financial and monetary calculations
decimal fooDecimal = 150.3m;

// Boolean - true & false
bool fooBoolean = true; // or false

// Char - A single 16-bit Unicode character
char fooChar = 'A';

// Strings -- unlike the previous base types which are
    all value types,
// a string is a reference type. That is, you can set it
     to null
string fooString = "\"escape\" quotes and add \n (new
    lines) and \t (tabs)";
Console.WriteLine(fooString);

// You can access each character of the string with an
    indexer:
char charFromString = fooString[1]; // => 'e'
// Strings are immutable: you can't do fooString[1] = 'X
    ';

// Compare strings with current culture, ignoring case
string.Compare(fooString, "x", StringComparison.
    CurrentCultureIgnoreCase);

// Formatting, based on sprintf
string fooFs = string.Format("Check Check, {0} {1}, {0}
    {1:0.0}", 1, 2);

// Dates & Formatting
```

```
            DateTime fooDate = DateTime.Now;
            Console.WriteLine(fooDate.ToString("hh:mm, dd MMM yyyy")
                );

            // Verbatim String
            // You can use the @ symbol before a string literal to
                escape all characters in the string
            string path = "C:\\Users\\User\\Desktop";
            string verbatimPath = @"C:\Users\User\Desktop";
            Console.WriteLine(path == verbatimPath);  // => true

            // You can split a string over two lines with the @
                symbol. To escape " use ""
            string bazString = @"Here's some stuff
on a new line! ""Wow!"", the masses cried";

            // Use const or read-only to make a variable immutable
            // const values are calculated at compile time
            const int HoursWorkPerWeek = 9001;

            /////////////////////////////////////////////////
            // Data Structures
            /////////////////////////////////////////////////

            // Arrays - zero indexed
            // The array size must be decided upon declaration
            // The format for declaring an array is follows:
            // <datatype>[] <var name> = new <datatype>[<array size
                >];
            int[] intArray = new int[10];

            // Another way to declare & initialize an array
            int[] y = { 9000, 1000, 1337 };

            // Indexing an array - Accessing an element
            Console.WriteLine("intArray @ 0: " + intArray[0]);
            // Arrays are mutable.
            intArray[1] = 1;

            // Lists
            // Lists are used more frequently than arrays as they
                are more flexible
            // The format for declaring a list is follows:
            // List<datatype> <var name> = new List<datatype>();
            List<int> intList = new List<int>();
            List<string> stringList = new List<string>();
```

```
List<int> z = new List<int> { 9000, 1000, 1337 }; //
    initialize
// The <> are for generics - Check out the cool stuff
    section

// Lists don't default to a value;
// A value must be added before accessing the index
intList.Add(1);
Console.WriteLine("intList @ 0: " + intList[0]);

// Others data structures to check out:
// Stack/Queue
// Dictionary (an implementation of a hash map)
// HashSet
// Read-only Collections
// Tuple (.Net 4+)

///////////////////////////////////////
// Operators
///////////////////////////////////////
Console.WriteLine("\n->Operators");

int i1 = 1, i2 = 2; // Shorthand for multiple
    declarations

// Arithmetic is straightforward
Console.WriteLine(i1 + i2 - i1 * 3 / 7); // => 3

// Modulo
Console.WriteLine("11%3 = " + (11 % 3)); // => 2

// Comparison operators
Console.WriteLine("3 == 2? " + (3 == 2)); // => false
Console.WriteLine("3 != 2? " + (3 != 2)); // => true
Console.WriteLine("3 > 2? " + (3 > 2)); // => true
Console.WriteLine("3 < 2? " + (3 < 2)); // => false
Console.WriteLine("2 <= 2? " + (2 <= 2)); // => true
Console.WriteLine("2 >= 2? " + (2 >= 2)); // => true

// Bitwise operators!
/*
~       Unary bitwise complement
<<      Signed left shift
>>      Signed right shift
&       Bitwise AND
^       Bitwise exclusive OR
```

```
|       Bitwise inclusive OR
*/

// Incrementations
int i = 0;
Console.WriteLine("\n->Inc/Dec-rementation");
Console.WriteLine(i++); //Prints "0", i = 1. Post-
    Incrementation
Console.WriteLine(++i); //Prints "2", i = 2. Pre-
    Incrementation
Console.WriteLine(i--); //Prints "2", i = 1. Post-
    Decrementation
Console.WriteLine(--i); //Prints "0", i = 0. Pre-
    Decrementation

////////////////////////////////////
// Control Structures
////////////////////////////////////
Console.WriteLine("\n->Control Structures");

// If statements are c-like
int j = 10;
if (j == 10)
{
    Console.WriteLine("I get printed");
}
else if (j > 10)
{
    Console.WriteLine("I don't");
}
else
{
    Console.WriteLine("I also don't");
}

// Ternary operators
// A simple if/else can be written as follows
// <condition> ? <true> : <false>
int toCompare = 17;
string isTrue = toCompare == 17 ? "True" : "False";

// While loop
int fooWhile = 0;
while (fooWhile < 100)
{
    //Iterated 100 times, fooWhile 0->99
```

```
    fooWhile++;
}

// Do While Loop
int fooDoWhile = 0;
do
{
    // Start iteration 100 times, fooDoWhile 0->99
    if (false)
        continue; // skip the current iteration

    fooDoWhile++;

    if (fooDoWhile == 50)
        break; // breaks from the loop completely

} while (fooDoWhile < 100);

//for loop structure => for(<start_statement>; <
    conditional>; <step>)
for (int fooFor = 0; fooFor < 10; fooFor++)
{
    //Iterated 10 times, fooFor 0->9
}

// For Each Loop
// foreach loop structure => foreach(<iteratorType> <
    iteratorName> in <enumerable>)
// The foreach loop loops over any object implementing
    IEnumerable or IEnumerable<T>
// All the collection types (Array, List, Dictionary...)
     in the .Net framework
// implement one or both of these interfaces.
// (The ToCharArray() could be removed, because a string
     also implements IEnumerable)
foreach (char character in "Hello World".ToCharArray())
{
    //Iterated over all the characters in the string
}

// Switch Case
// A switch works with the byte, short, char, and int
    data types.
// It also works with enumerated types (discussed in
    Enum Types),
// the String class, and a few special classes that wrap
```

```
// primitive types: Character, Byte, Short, and Integer.
int month = 3;
string monthString;
switch (month)
{
    case 1:
        monthString = "January";
        break;
    case 2:
        monthString = "February";
        break;
    case 3:
        monthString = "March";
        break;
    // You can assign more than one case to an action
    // But you can't add an action without a break
       before another case
    // (if you want to do this, you would have to
       explicitly add a goto case x
    case 6:
    case 7:
    case 8:
        monthString = "Summer time!!";
        break;
    default:
        monthString = "Some other month";
        break;
}

//////////////////////////////////////
// Converting Data Types And Typecasting
//////////////////////////////////////

// Converting data

// Convert String To Integer
// this will throw a FormatException on failure
int.Parse("123");//returns an integer version of "123"

// try parse will default to type default on failure
// in this case: 0
int tryInt;
if (int.TryParse("123", out tryInt)) // Function is
    boolean
    Console.WriteLine(tryInt);       // 123
```

```
        // Convert Integer To String
        // Convert class has a number of methods to facilitate
           conversions
        Convert.ToString(123);
        // or
        tryInt.ToString();

        // Casting
        // Cast decimal 15 to a int
        // and then implicitly cast to long
        long x = (int) 15M;
    }

    ///////////////////////////////////////
    // CLASSES - see definitions at end of file
    ///////////////////////////////////////
    public static void Classes()
    {
        // See Declaration of objects at end of file

        // Use new to instantiate a class
        Bicycle trek = new Bicycle();

        // Call object methods
        trek.SpeedUp(3); // You should always use setter and
           getter methods
        trek.Cadence = 100;

        // ToString is a convention to display the value of this
            Object.
        Console.WriteLine("trek info: " + trek.Info());

        // Instantiate a new Penny Farthing
        PennyFarthing funbike = new PennyFarthing(1, 10);
        Console.WriteLine("funbike info: " + funbike.Info());

        Console.Read();
    } // End main method

    // CONSOLE ENTRY A console application must have a main
       method as an entry point
    public static void Main(string[] args)
    {
        OtherInterestingFeatures();
    }
```

```
//
// INTERESTING FEATURES
//

// DEFAULT METHOD SIGNATURES

public // Visibility
static // Allows for direct call on class without object
int // Return Type,
MethodSignatures(
    int maxCount, // First variable, expects an int
    int count = 0, // will default the value to 0 if not
        passed in
    int another = 3,
    params string[] otherParams // captures all other
        parameters passed to method
)
{
    return -1;
}

// Methods can have the same name, as long as the signature
    is unique
// A method that differs only in return type is not unique
public static void MethodSignatures(
    ref int maxCount, // Pass by reference
    out int count)
{
    //the argument passed in as 'count' will hold the value
        of 15 outside of this function
    count = 15; // out param must be assigned before control
         leaves the method
}

// GENERICS
// The classes for TKey and TValue is specified by the user
    calling this function.
// This method emulates the SetDefault of Python
public static TValue SetDefault<TKey, TValue>(
    IDictionary<TKey, TValue> dictionary,
    TKey key,
    TValue defaultItem)
{
    TValue result;
    if (!dictionary.TryGetValue(key, out result))
        return dictionary[key] = defaultItem;
```

```
    return result;
}

// You can narrow down the objects that are passed in
public static void IterateAndPrint<T>(T toPrint) where T:
    IEnumerable<int>
{
    // We can iterate, since T is a IEnumerable
    foreach (var item in toPrint)
        // Item is an int
        Console.WriteLine(item.ToString());
}

// YIELD
// Usage of the "yield" keyword indicates that the method it
     appears in is an Iterator
// (this means you can use it in a foreach loop)
public static IEnumerable<int> YieldCounter(int limit = 10)
{
    for (var i = 0; i < limit; i++)
        yield return i;
}

// which you would call like this :
public static void PrintYieldCounterToConsole()
{
    foreach (var counter in YieldCounter())
        Console.WriteLine(counter);
}

// you can use more than one "yield return" in a method
public static IEnumerable<int> ManyYieldCounter()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
}

// you can also use "yield break" to stop the Iterator
// this method would only return half of the values from 0
    to limit.
public static IEnumerable<int> YieldCounterWithBreak(int
    limit = 10)
{
    for (var i = 0; i < limit; i++)
```

```
    {
        if (i > limit/2) yield break;
        yield return i;
    }
}

public static void OtherInterestingFeatures()
{
    // OPTIONAL PARAMETERS
    MethodSignatures(3, 1, 3, "Some", "Extra", "Strings");
    MethodSignatures(3, another: 3); // explicitly set a
        parameter, skipping optional ones

    // BY REF AND OUT PARAMETERS
    int maxCount = 0, count; // ref params must have value
    MethodSignatures(ref maxCount, out count);

    // EXTENSION METHODS
    int i = 3;
    i.Print(); // Defined below

    // NULLABLE TYPES - great for database interaction /
        return values
    // any value type (i.e. not a class) can be made
        nullable by suffixing a ?
    // <type>? <var name> = <value>
    int? nullable = null; // short hand for Nullable<int>
    Console.WriteLine("Nullable variable: " + nullable);
    bool hasValue = nullable.HasValue; // true if not null

    // ?? is syntactic sugar for specifying default value (
        coalesce)
    // in case variable is null
    int notNullable = nullable ?? 0; // 0

    // ?. is an operator for null-propagation - a shorthand
        way of checking for null
    nullable?.Print(); // Use the Print() extension method
        if nullable isn't null

    // IMPLICITLY TYPED VARIABLES - you can let the compiler
         work out what the type is:
    var magic = "magic is a string, at compile time, so you
        still get type safety";
    // magic = 9; will not work as magic is a string, not an
         int
```

```
// GENERICS
//
var phonebook = new Dictionary<string, string>() {
    {"Sarah", "212 555 5555"} // Add some entries to the
        phone book
};

// Calling SETDEFAULT defined as a generic above
Console.WriteLine(SetDefault<string,string>(phonebook, "
    Shaun", "No Phone")); // No Phone
// nb, you don't need to specify the TKey and TValue
    since they can be
// derived implicitly
Console.WriteLine(SetDefault(phonebook, "Sarah", "No
    Phone")); // 212 555 5555

// LAMBDA EXPRESSIONS - allow you to write code in line
Func<int, int> square = (x) => x * x; // Last T item is
    the return value
Console.WriteLine(square(3)); // 9

// ERROR HANDLING - coping with an uncertain world
try
{
    var funBike = PennyFarthing.CreateWithGears(6);

    // will no longer execute because CreateWithGears
        throws an exception
    string some = "";
    if (true) some = null;
    some.ToLower(); // throws a NullReferenceException
}
catch (NotSupportedException)
{
    Console.WriteLine("Not so much fun now!");
}
catch (Exception ex) // catch all other exceptions
{
    throw new ApplicationException("It hit the fan", ex)
        ;
    // throw; // A rethrow that preserves the callstack
}
// catch { } // catch-all without capturing the
    Exception
finally
```

```
{
    // executes after try or catch
}

// DISPOSABLE RESOURCES MANAGEMENT - let you handle
   unmanaged resources easily.
// Most of objects that access unmanaged resources (file
    handle, device contexts, etc.)
// implement the IDisposable interface. The using
   statement takes care of
// cleaning those IDisposable objects for you.
using (StreamWriter writer = new StreamWriter("log.txt")
   )
{
    writer.WriteLine("Nothing suspicious here");
    // At the end of scope, resources will be released.
    // Even if an exception is thrown.
}

// PARALLEL FRAMEWORK
// http://blogs.msdn.com/b/csharpfaq/archive/2010/06/01/
   parallel-programming-in-net-framework-4-getting-
   started.aspx

var words = new List<string> {"dog", "cat", "horse", "
   pony"};

Parallel.ForEach(words,
    new ParallelOptions() { MaxDegreeOfParallelism = 4
       },
    word =>
    {
        Console.WriteLine(word);
    }
);

//Running this will produce different outputs
//since each thread finishes at different times.
//Some example outputs are:
//cat dog horse pony
//dog horse pony cat

// DYNAMIC OBJECTS (great for working with other
   languages)
dynamic student = new ExpandoObject();
student.FirstName = "First Name"; // No need to define
```

```
    class first!

// You can even add methods (returns a string, and takes
     in a string)
student.Introduce = new Func<string, string>(
    (introduceTo) => string.Format("Hey {0}, this is {1}
        ", student.FirstName, introduceTo));
Console.WriteLine(student.Introduce("Beth"));

// IQUERYABLE<T> - almost all collections implement this
    , which gives you a lot of
// very useful Map / Filter / Reduce style methods
var bikes = new List<Bicycle>();
bikes.Sort(); // Sorts the array
bikes.Sort((b1, b2) => b1.Wheels.CompareTo(b2.Wheels));
    // Sorts based on wheels
var result = bikes
    .Where(b => b.Wheels > 3) // Filters - chainable (
        returns IQueryable of previous type)
    .Where(b => b.IsBroken && b.HasTassles)
    .Select(b => b.ToString()); // Map - we only this
        selects, so result is a IQueryable<string>

var sum = bikes.Sum(b => b.Wheels); // Reduce - sums all
     the wheels in the collection

// Create a list of IMPLICIT objects based on some
    parameters of the bike
var bikeSummaries = bikes.Select(b=>new { Name = b.Name,
     IsAwesome = !b.IsBroken && b.HasTassles });
// Hard to show here, but you get type ahead completion
    since the compiler can implicitly work
// out the types above!
foreach (var bikeSummary in bikeSummaries.Where(b => b.
    IsAwesome))
    Console.WriteLine(bikeSummary.Name);

// ASPARALLEL
// And this is where things get wicked - combine linq
    and parallel operations
var threeWheelers = bikes.AsParallel().Where(b => b.
    Wheels == 3).Select(b => b.Name);
// this will happen in parallel! Threads will
    automagically be spun up and the
// results divvied amongst them! Amazing for large
    datasets when you have lots of
```

```
            // cores

            // LINQ - maps a store to IQueryable<T> objects, with
                delayed execution
            // e.g. LinqToSql - maps to a database, LinqToXml maps
                to an xml document
            var db = new BikeRepository();

            // execution is delayed, which is great when querying a
                database
            var filter = db.Bikes.Where(b => b.HasTassles); // no
                query run
            if (42 > 6) // You can keep adding filters, even
                conditionally - great for "advanced search"
                functionality
                filter = filter.Where(b => b.IsBroken); // no query
                    run

            var query = filter
                .OrderBy(b => b.Wheels)
                .ThenBy(b => b.Name)
                .Select(b => b.Name); // still no query run

            // Now the query runs, but opens a reader, so only
                populates as you iterate through
            foreach (string bike in query)
                Console.WriteLine(result);



    }

} // End LearnCSharp class

// You can include other classes in a .cs file

public static class Extensions
{
    // EXTENSION METHODS
    public static void Print(this object obj)
    {
        Console.WriteLine(obj.ToString());
    }
}
```

```
// DELEGATES AND EVENTS
public class DelegateTest
{
    public static int count = 0;
    public static int Increment()
    {
        // increment count then return it
        return ++count;
    }

    // A delegate is a reference to a method
    // To reference the Increment method,
    // first declare a delegate with the same signature
    // ie. takes no arguments and returns an int
    public delegate int IncrementDelegate();

    // An event can also be used to trigger delegates
    // Create an event with the delegate type
    public static event IncrementDelegate MyEvent;

    static void Main(string[] args)
    {
        // Refer to the Increment method by instantiating the
            delegate
        // and passing the method itself in as an argument
        IncrementDelegate inc = new IncrementDelegate(Increment)
            ;
        Console.WriteLine(inc());  // => 1

        // Delegates can be composed with the + operator
        IncrementDelegate composedInc = inc;
        composedInc += inc;
        composedInc += inc;

        // composedInc will run Increment 3 times
        Console.WriteLine(composedInc());  // => 4


        // Subscribe to the event with the delegate
        MyEvent += new IncrementDelegate(Increment);
        MyEvent += new IncrementDelegate(Increment);

        // Trigger the event
        // ie. run all delegates subscribed to this event
        Console.WriteLine(MyEvent());  // => 6
    }
```

```
}


// Class Declaration Syntax:
// <public/private/protected/internal> class <class name>{
//    //data fields, constructors, functions all inside.
//    //functions are called as methods in Java.
// }

public class Bicycle
{
    // Bicycle's Fields/Variables
    public int Cadence // Public: Can be accessed from anywhere
    {
        get // get - define a method to retrieve the property
        {
            return _cadence;
        }
        set // set - define a method to set a property
        {
            _cadence = value; // Value is the value passed in to
                the setter
        }
    }
    private int _cadence;

    protected virtual int Gear // Protected: Accessible from the
        class and subclasses
    {
        get; // creates an auto property so you don't need a
            member field
        set;
    }

    internal int Wheels // Internal: Accessible from within the
        assembly
    {
        get;
        private set; // You can set modifiers on the get/set
            methods
    }

    int _speed; // Everything is private by default: Only
        accessible from within this class.
                // can also use keyword private
    public string Name { get; set; }
```

```
// Properties also have a special syntax for when you want a
    readonly property
// that simply returns the result of an expression
public string LongName => Name + " " + _speed + " speed";

// Enum is a value type that consists of a set of named
   constants
// It is really just mapping a name to a value (an int,
   unless specified otherwise).
// The approved types for an enum are byte, sbyte, short,
   ushort, int, uint, long, or ulong.
// An enum can't contain the same value twice.
public enum BikeBrand
{
    AIST,
    BMC,
    Electra = 42, //you can explicitly set a value to a name
    Gitane // 43
}
// We defined this type inside a Bicycle class, so it is a
   nested type
// Code outside of this class should reference this type as
   Bicycle.Brand

public BikeBrand Brand; // After declaring an enum type, we
   can declare the field of this type

// Decorate an enum with the FlagsAttribute to indicate that
    multiple values can be switched on
// Any class derived from Attribute can be used to decorate
   types, methods, parameters etc
// Bitwise operators & and | can be used to perform and/or
   operations

[Flags]
public enum BikeAccessories
{
    None = 0,
    Bell = 1,
    MudGuards = 2, // need to set the values manually!
    Racks = 4,
    Lights = 8,
    FullPackage = Bell | MudGuards | Racks | Lights
}
```

```
// Usage: aBike.Accessories.HasFlag(Bicycle.BikeAccessories.
    Bell)
// Before .NET 4: (aBike.Accessories & Bicycle.
    BikeAccessories.Bell) == Bicycle.BikeAccessories.Bell
public BikeAccessories Accessories { get; set; }

// Static members belong to the type itself rather than
    specific object.
// You can access them without a reference to any object:
// Console.WriteLine("Bicycles created: " + Bicycle.
    bicyclesCreated);
public static int BicyclesCreated { get; set; }

// readonly values are set at run time
// they can only be assigned upon declaration or in a
    constructor
readonly bool _hasCardsInSpokes = false; // read-only
    private

// Constructors are a way of creating classes
// This is a default constructor
public Bicycle()
{
    this.Gear = 1; // you can access members of the object
        with the keyword this
    Cadence = 50;  // but you don't always need it
    _speed = 5;
    Name = "Bontrager";
    Brand = BikeBrand.AIST;
    BicyclesCreated++;
}

// This is a specified constructor (it contains arguments)
public Bicycle(int startCadence, int startSpeed, int
    startGear,
                string name, bool hasCardsInSpokes, BikeBrand
                    brand)
    : base() // calls base first
{
    Gear = startGear;
    Cadence = startCadence;
    _speed = startSpeed;
    Name = name;
    _hasCardsInSpokes = hasCardsInSpokes;
    Brand = brand;
}
```

```
// Constructors can be chained
public Bicycle(int startCadence, int startSpeed, BikeBrand
    brand) :
     this(startCadence, startSpeed, 0, "big wheels", true,
        brand)
{
}

// Function Syntax:
// <public/private/protected> <return type> <function name
    >(<args>)

// classes can implement getters and setters for their
    fields
// or they can implement properties (this is the preferred
    way in C#)

// Method parameters can have default values.
// In this case, methods can be called with these parameters
     omitted
public void SpeedUp(int increment = 1)
{
     _speed += increment;
}

public void SlowDown(int decrement = 1)
{
     _speed -= decrement;
}

// properties get/set values
// when only data needs to be accessed, consider using
    properties.
// properties may have either get or set, or both
private bool _hasTassles; // private variable
public bool HasTassles // public accessor
{
     get { return _hasTassles; }
     set { _hasTassles = value; }
}

// You can also define an automatic property in one line
// this syntax will create a backing field automatically.
// You can set an access modifier on either the getter or
    the setter (or both)
```

```
// to restrict its access:
public bool IsBroken { get; private set; }

// Properties can be auto-implemented
public int FrameSize
{
    get;
    // you are able to specify access modifiers for either
       get or set
    // this means only Bicycle class can call set on
       Framesize
    private set;
}

// It's also possible to define custom Indexers on objects.
// All though this is not entirely useful in this example,
   you
// could do bicycle[0] which returns "chris" to get the
   first passenger or
// bicycle[1] = "lisa" to set the passenger. (of this
   apparent quattrocycle)
private string[] passengers = { "chris", "phil", "darren", "
   regina" };

public string this[int i]
{
    get {
        return passengers[i];
    }

    set {
        passengers[i] = value;
    }
}

//Method to display the attribute values of this Object.
public virtual string Info()
{
    return "Gear: " + Gear +
            " Cadence: " + Cadence +
            " Speed: " + _speed +
            " Name: " + Name +
            " Cards in Spokes: " + (_hasCardsInSpokes ? "yes
               " : "no") +
            "\n------------------------------\n"
            ;
```

```
    }

    // Methods can also be static. It can be useful for helper
       methods
    public static bool DidWeCreateEnoughBicycles()
    {
        // Within a static method, we only can reference static
           class members
        return BicyclesCreated > 9000;
    } // If your class only needs static members, consider
       marking the class itself as static.


} // end class Bicycle

// PennyFarthing is a subclass of Bicycle
class PennyFarthing : Bicycle
{
    // (Penny Farthings are those bicycles with the big front
       wheel.
    // They have no gears.)

    // calling parent constructor
    public PennyFarthing(int startCadence, int startSpeed) :
        base(startCadence, startSpeed, 0, "PennyFarthing", true,
            BikeBrand.Electra)
    {
    }

    protected override int Gear
    {
        get
        {
            return 0;
        }
        set
        {
            throw new InvalidOperationException("You can't
               change gears on a PennyFarthing");
        }
    }

    public static PennyFarthing CreateWithGears(int gears)
    {
        var penny = new PennyFarthing(1, 1);
        penny.Gear = gears; // Oops, can't do this!
```

```
        return penny;
    }

    public override string Info()
    {
        string result = "PennyFarthing bicycle ";
        result += base.ToString(); // Calling the base version
            of the method
        return result;
    }
}

// Interfaces only contain signatures of the members, without
    the implementation.
interface IJumpable
{
    void Jump(int meters); // all interface members are
        implicitly public
}

interface IBreakable
{
    bool Broken { get; } // interfaces can contain properties as
        well as methods & events
}

// Classes can inherit only one other class, but can implement
    any amount of interfaces,
// however the base class name must be the first in the list and
     all interfaces follow
class MountainBike : Bicycle, IJumpable, IBreakable
{
    int damage = 0;

    public void Jump(int meters)
    {
        damage += meters;
    }

    public bool Broken
    {
        get
        {
            return damage > 100;
        }
    }
```

```
}

/// <summary>
/// Used to connect to DB for LinqToSql example.
/// EntityFramework Code First is awesome (similar to Ruby's
    ActiveRecord, but bidirectional)
/// http://msdn.microsoft.com/en-us/data/jj193542.aspx
/// </summary>
public class BikeRepository : DbContext
{
    public BikeRepository()
        : base()
    {
    }

    public DbSet<Bicycle> Bikes { get; set; }
}

// Classes can be split across multiple .cs files
// A1.cs
public partial class A
{
    public static void A1()
    {
        Console.WriteLine("Method A1 in class A");
    }
}

// A2.cs
public partial class A
{
    public static void A2()
    {
        Console.WriteLine("Method A2 in class A");
    }
}

// Program using the partial class "A"
public class Program
{
    static void Main()
    {
        A.A1();
        A.A2();
    }
}
```

```
// String interpolation by prefixing the string with $
// and wrapping the expression you want to interpolate with {
    braces }
// You can also combine both interpolated and verbatim strings
    with $@
public class Rectangle
{
    public int Length { get; set; }
    public int Width { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Rectangle rect = new Rectangle { Length = 5, Width = 3
            };
        Console.WriteLine($"The length is {rect.Length} and the
            width is {rect.Width}");

        string username = "User";
        Console.WriteLine($@"C:\Users\{username}\Desktop");
    }
}

// New C# 6 features
class GlassBall : IJumpable, IBreakable
{
    // Autoproperty initializers
    public int Damage { get; private set; } = 0;

    // Autoproperty initializers on getter-only properties
    public string Name { get; } = "Glass ball";

    // Getter-only autoproperty that is initialized in
        constructor
    public string GenieName { get; }

    public GlassBall(string genieName = null)
    {
        GenieName = genieName;
    }

    public void Jump(int meters)
    {
```

```
        if (meters < 0)
            // New nameof() expression; compiler will check that
                the identifier exists
            // nameof(x) == "x"
            // Prevents e.g. parameter names changing but not
                updated in error messages
            throw new ArgumentException("Cannot jump negative
                amount!", nameof(meters));

        Damage += meters;
    }

    // Expression-bodied properties ...
    public bool Broken
        => Damage > 100;

    // ... and methods
    public override string ToString()
        // Interpolated string
        => $"{Name}. Damage taken: {Damage}";

    public string SummonGenie()
        // Null-conditional operators
        // x?.y will return null immediately if x is null; y is
            not evaluated
        => GenieName?.ToUpper();
}

static class MagicService
{
    private static bool LogException(Exception ex)
    {
        /* log exception somewhere */
        return false;
    }

    public static bool CastSpell(string spell)
    {
        try
        {
            // Pretend we call API here
            throw new MagicServiceException("Spell failed", 42);

            // Spell succeeded
            return true;
        }
```

```
        // Only catch if Code is 42 i.e. spell failed
        catch(MagicServiceException ex) when (ex.Code == 42)
        {
            // Spell failed
            return false;
        }
        // Other exceptions, or MagicServiceException where Code
            is not 42
        catch(Exception ex) when (LogException(ex))
        {
            // Execution never reaches this block
            // The stack is not unwound
        }
        return false;
        // Note that catching a MagicServiceException and
            rethrowing if Code
        // is not 42 or 117 is different, as then the final
            catch-all block
        // will not catch the rethrown exception
    }
}

public class MagicServiceException : Exception
{
    public int Code { get; }

    public MagicServiceException(string message, int code) :
        base(message)
    {
        Code = code;
    }
}

public static class PragmaWarning {
    // Obsolete attribute
    [Obsolete("Use NewMethod instead", false)]
    public static void ObsoleteMethod()
    {
        /* obsolete code */
    }

    public static void NewMethod()
    {
        /* new code */
    }
```

```
        public static void Main()
        {
            ObsoleteMethod(); // CS0618: 'ObsoleteMethod is obsolete
                : Use NewMethod instead'
#pragma warning disable CS0618
            ObsoleteMethod(); // no warning
#pragma warning restore CS0618
            ObsoleteMethod(); // CS0618: 'ObsoleteMethod is obsolete
                : Use NewMethod instead'
        }
    }
} // End Namespace

using System;
// C# 6, static using
using static System.Math;

namespace Learning.More.CSharp
{
    class StaticUsing
    {
        static void Main()
        {
            // Without a static using statement..
            Console.WriteLine("The square root of 4 is {}.", Math.
                Sqrt(4));
            // With one
            Console.WriteLine("The square root of 4 is {}.", Sqrt(4)
                );
        }
    }
}

//New C# 7 Feature
//Install Microsoft.Net.Compilers Latest from Nuget
//Install System.ValueTuple Latest from Nuget
using System;
namespace Csharp7
{
    // TUPLES, DECONSTRUCTION AND DISCARDS
    class TuplesTest
    {
        public (string, string) GetName()
        {
            // Fields in tuples are by default named Item1, Item2...
            var names1 = ("Peter", "Parker");
```

```
    Console.WriteLine(names1.Item2);  // => Parker

    // Fields can instead be explicitly named
    // Type 1 Declaration
    (string FirstName, string LastName) names2 = ("Peter", "
       Parker");

    // Type 2 Declaration
    var names3 = (First:"Peter", Last:"Parker");

    Console.WriteLine(names2.FirstName);  // => Peter
    Console.WriteLine(names3.Last);  // => Parker

    return names3;
}

public string GetLastName() {
    var fullName = GetName();

    // Tuples can be deconstructed
    (string firstName, string lastName) = fullName;

    // Fields in a deconstructed tuple can be discarded by
       using _
    var (_, last) = fullName;
    return last;
}

// Any type can be deconstructed in the same way by
// specifying a Deconstruct method
public int randomNumber = 4;
public int anotherRandomNumber = 10;

public void Deconstruct(out int randomNumber, out int
   anotherRandomNumber)
{
    randomNumber = this.randomNumber;
    anotherRandomNumber = this.anotherRandomNumber;
}

static void Main(string[] args)
{
    var tt = new TuplesTest();
    (int num1, int num2) = tt;
    Console.WriteLine($"num1: {num1}, num2: {num2}");  // =>
        num1: 4, num2: 10
```

```
        Console.WriteLine(tt.GetLastName());
    }
}

// PATTERN MATCHING
class PatternMatchingTest
{
    public static (string, int)? CreateLogMessage(object data)
    {
        switch(data)
        {
            // Additional filtering using when
            case System.Net.Http.HttpRequestException h when h.
               Message.Contains("404"):
                return (h.Message, 404);
            case System.Net.Http.HttpRequestException h when h.
               Message.Contains("400"):
                return (h.Message, 400);
            case Exception e:
                return (e.Message, 500);
            case string s:
                return (s, s.Contains("Error") ? 500 : 200);
            case null:
                return null;
            default:
                return (data.ToString(), 500);
        }
    }
}

// REFERENCE LOCALS
// Allow you to return a reference to an object instead of just
   its value
class RefLocalsTest
{
    // note ref in return
    public static ref string FindItem(string[] arr, string el)
    {
        for(int i=0; i<arr.Length; i++)
        {
            if(arr[i] == el) {
                // return the reference
                return ref arr[i];
            }
        }
```

```
            throw new Exception("Item not found");
        }

        public static void SomeMethod()
        {
            string[] arr = {"this", "is", "an", "array"};

            // note refs everywhere
            ref string item = ref FindItem(arr, "array");
            item = "apple";
            Console.WriteLine(arr[3]);  // => apple
        }
    }

    // LOCAL FUNCTIONS
    class LocalFunctionTest
    {
        private static int _id = 0;
        public int id;
        public LocalFunctionTest()
        {
            id = generateId();

            // This local function can only be accessed in this
               scope
            int generateId()
            {
                return _id++;
            }
        }

        public static void AnotherMethod()
        {
            var lf1 = new LocalFunctionTest();
            var lf2 = new LocalFunctionTest();
            Console.WriteLine($"{lf1.id}, {lf2.id}");  // => 0, 1

            int id = generateId();
            // error CS0103: The name 'generateId' does not exist in
                the current context
        }
    }
}
```