C# Fundamentals Cheat Sheet

Data Types (CS-ASP_007)

==========

Comments: // Single line ... /* Multiline */

string => alphanumeric (defaults to null)

int => 2 billion to -2 billion (defaults to 0)

double => fractional value (defaults to 0.0)

bool => true or false (defaults to false)

More: http://is.gd/data_types

Data Type Conversion (CS-ASP_008)

==========

Implicit conversions - smaller type to larger type

without data loss, "upcasting"

Explicit conversions - require developer intervention,

possibility of data loss, "downcasting", either in

the form of cast or using a helper method.

Casting numbers:

```
int myInteger = (int)myDouble;
```

Numbers to strings:

```
string myString = myInteger.ToString();
```

String to Numbers:

```
int myInteger = int.Parse(myString);
```

More: http://is.gd/datatype_conversion

Arithmetic Operators (CS-ASP_009)

==========

= NOT equality, it's assignment

Math Operators: + - * /

Addition Assignment

```
total = total + 5;
total += 5;
```

Increment Operator: i++;

Decrement Operator: i--;

Beware of order of precedence (use parenthesis)

Beware of down casting (you'll lose precision)

Beware of overflow (use bigger types)

To make overflow throw an exception:

```
checked
{
// some arithmetic operation
```

// that could potentially overflow

}

C# Syntax (CS-ASP_010)

==========

Operands - variable names, object / server control

names, literals - "Nouns" (you name these)

Operators - "Verbs ... they act on the operands.

http://is.gd/operators

Expressions - One or more operands and zero or more

operators that evaluate to a single value

http://is.gd/expressions

Statements - A complete instruction - assignment of an

expression to a variable, an increment/ decrement, etc.

http://is.gd/statement

Statements must end in a semi-colon ;

Whitespace is ignored (use for humans)

Conditional if ... else if ... else Statement (CS-ASP_011)

==========

= Assignment

== Equality

```
if (a == b)
{
// execute when the expression is true
}
else
{
// executes when the expression is false
}
```

... or ... evaluate other mutually exclusively options:

```
if (a == b) { // some code }
else if (a == c) { // some code }
else if (a == d) { // some code }
else { // catch all }
```

CheckBox Server Control = Checked prop is bool

RadioButton Server Control = GroupName prop groups them together, check prop is bool

Conditional Ternary Operator (CS-ASP_012)

==========

Shortcut for evaluating an expression and returning a result.

```
result = (a == b) ? "Equal" : "Not Equal";
```

Comparison and Logical Operator (CS-ASP_013)

==========

Comparison Operators

used for conditional statements

==

!=

< >

<= >=

!someBooleanValue - means NOT is true

Logical Operators

used to combine multiple expressions / evaluation

&& - AND

|| - OR

Combine with parenthesis ( ) for order of precendence

Working with Dates and Times (CS-ASP_014)

==========

Creating new DateTime objects

DateTime myDateTime = DateTime.Now;

DateTime myDateTime = DateTime.Parse("12/7/1969");

Formatting ... many options:

myDateTime.To_____()

Retrieving Parts:

myDateTime.Year // int

myDateTime.Hour // int

myDateTime.DayOfWeek // "Monday"

myDateTime.DayOfYear // int 175

DateTime Math:

myDateTime.AddHours(3)

myDateTime.AddMinutes(-5)

"Chaining" = using multiple helper methods

together with the dot . operator

myDateTime.AddHours(3).AddMinutes(-5).ToString()

Working with Spans of Time (CS-ASP_015)

==========

Create and initialize new TimeSpans

// Days.Hours:Minutes:Seconds.Milliseconds

TimeSpan myTimeSpan = TimeSpan.Parse("1.2:3:30.5");

DateTime myBirthday = DateTime.Parse("12/7/1969");

TimeSpan myAge = DateTime.Now.Subtract(myBirthday);

More info at http://is.gd/timespan

Get individual parts

myAge.Hours

myAge.Seconds

... or get TOTAL elapsed time

as a double representing both

the number of days / hours / etc.

AND fractional values representing

"left overs".

myTimeSpan.TotalDays // double

myTimeSpan.TotalHours // double

Formatting Strings (CS-ASP_019)

==========

Concenate Strings

+ +=

Format Strings

String.Format("Hello {0}. You are from {1}", "Bob", "Chicago")

Format Numbers

String.Format("Reference Code: {0:000_000-0}", 1234567)

123_456-7

Formatting Dates

http://is.gd/formattingdates

String.Format("REference Date: {0:ddd - d, MM, yyyy}", someDate)

Tue - 5, 07, 2014

Formatting Currency

http://is.gd/formattingcurrency

String.Format("Total: {0:C}", totalAmount);

$50,000.00

Single Dimensional Arrays (CS-ASP_021)

==========

Indexes vs. Elements

Accessor vs. Stored Values

Indexes are zero based

Declaring Arrays

string[] myArray = new string[3];

Declaring and Initializing Arrays

string[] myArray = new string[3] { "Moe", "Larry", "Curly" };

Setting / Getting Values

string myString = myArray[1]; // Retrieve the second element

myArray[0] = myString; // Sets first element

Multi-Dimensional Arrays (CS-ASP_022)

==========

Same as single dimensional ... just requires

more indexes (in dimensions) to get to the element

double[,] myArray = new double[2,3]; // contains 6 elements

```
int[,,] rubicsCube = new int[3,3,3] // contains 27 elements

rubicsCube[0,1,2] = 42;
myInteger = rubicsCube[0,1,2];
```

Changing the Length of an Array (CS-ASP_023)

==========

Arrays are immutable = cannot be changed in memory

HOWEVER .NET Framework providers helper methods to

resize an array ... creates a new array and copies

the old values into it.

```
Array.Resize(ref myArray, myArray.Length + 1);

// Get the highest index:
int highestIndex = myArray.GetUpperBound(0);
// 0 = dimension we want to retrieve the
// upper boundary for

// Arrays have other helper methods
myArray.Sum()
myArray.Min()
myArray.Max()
myArray.Average()

Array.Sort(myArray)
Array.Reverse(myArray)
```

Looping with the for Iteration Statement (CS-ASP_026)

==========

Snippet: for [tab] [tab]

Then you can tab through the replaceable bits, hit enter to

start writing code in the code block body.

```
for (int i = 0; i < 10; i++)
{
// Your code here
}
```

i - Any variable name

1st part - counter declaration, can be initialized to any number

2nd part - condition, can be any expression that equates to a bool

3rd part - increment i++ / decrement i--, can step more than 1 using +=,

```
string[] names = new string[] { "Wolverine", "Cyclops", "Professor X",

"Phoenix" };

for (int i = 0; i < names.Length; i++)
{
// Cna search for a specific value
if (names[i] == "Professor X")
{
// Do something here
```

```
// Can break out of additional iterations if you need to
break;
}
}
```

Looping with the while and do ... while Iteration Statements (CS-ASP_027)

==========

```
Random randon = new Random();
// random.Next(lowerBounds, upperBounds)
random.Next(1, 100); // returns a value between 1 and 100

// If someExpression is already false, this will never execute
while(someExpression) {
// Code that would affect whether
// someExpression is true or false
}

// If someExpression is already false, this will run AT LEAST ONCE
do {
// Code that would affect whether
// someExpression is true or false
} while (someExpression)
```

Creating Overloaded MEthods (CS-ASP_031)

==========

Different METHOD SIGNATURE, but same basic function.

METHOD SIGNATURE ... the number and type of parameters

Can have different return types.

Creating Optional Parameters (CS-ASP_032)

==========

Optional parameters provide default values when you define

the method. So, if you don't supply a value, the default

will be used.

```
private void myMethod(string myRequiredParam,
int myOptionalParam = 1,
int myOtherOptionalParam = 5);

// Can be called ...

myMethod("Hello Required Param!");

// or ...

myMethod("Hello Required Param!", 100);

// or ...

myMethod("Hello Required Param!", 100, 500);

// You cannot skip an optional parameter:

myMethod("Hello Required Param!", , 500); // ERROR
```

Passing Named Arguments Into Input Parameters (CS-ASP_033)

==========

Allow us to send in parameter arugments OUT OF ORDER!

We just previx the input parameter argument with the name

of the parameter we're passing in, a colon, then the value:

```
myMethod(myOtherOptionalParam: 500,
myRequiredParam: "Hello Required PAram",
myOptionalParam: 100);

// You still have to pass in REQUIRED parameters.
```

Creating Methods with Output Parameters (CS-ASP_034)

==========

Allows you to return a value the normal way AND

return a value via a method parameter:

```
private bool myMethod(string myRequiredParam,
out int myOptionalParam) { }


int myValue = 0;
if (myMethod("Some required text", out myValue)) return "Hello World!";
```

Manipulating Strings (CS-ASP_035)

```
================================

// Escape double quotes
string myString = "This is a double quote: \".";

// Accessing a specific char:
myString[2]

// StartsWith(), EndsWith(), Contains()
// Check to see if a given string has a set of
// characters beginning, end or somewhere inside.
// Return true / false

// IndexOf()
// Find the index for one string inside of
// another string.
int myIndex = myString.IndexOf("howdy");

// Insert(), Remove()
// Insert adds characters starting at a given index
// Remove removes characters starting at a given
// index, and all the way through the length you
// input.

// Substring()
// Retrieve characters beginning at a given index
// all thr way through the length you input.

// Trim(), TrimStart(), TrimEnd()
```

```
// Remove space characters both, or just the start
// or the end of the string.

// PadLeft(), PadRight()
// Allow you to specify a length for a string
// and a character to pad the string with if its
// length is less than the specified length.
myString = someValue.PadLeft(10, '#');
// Notice that we're inputting a char, not a string
// therefore we have to use a single quote ' not
// a double quote.

// ToUpper(), ToLower()
// Important! Compare two strings regardless of
// the case, beacuse in C#, two strings with
// different cases are NOT equal.


// Replace()
// Replace every occurance of one string with
// some other string.
myString.Replace("$$$", myValue);

// Split()
// Take a string and split it into many strings
// and store them in a string array.
string[] names = myString.Split(';');

// Concatenating strings, immutability
```

// StringBuilder - memory efficient way of concatenating strings.

Introduction to Classes and Objects (CS-ASP_036)

==========

Class is a code block that defines a data type.

An Object is an instance of a Class.

Metaphors:

Blueprint vs. Houses

Recipe vs. Cupcakes

Pattern vs. Bluejeans

Cookie cutter vs. Cookes

Classes have members, like Properties and Methods.

Properties define the attributes that are set on
an instance of the class / represent the "state"
of the object. You can set (assign) and get
(retrieve) properties values on an object.

Methods define actions an instance of a class
can perform, usually on the object instance itself.

You can create an instance of a class using the
new keyword. Think: "factory".

You can access the members of an object by using
the member access operator, the dot (.)

Conceptually, classes are delegated a responsibility
in the system or represent some domain concept
in the system.

Classes are ultimately custom data types, more complex
than the simple data types we've worked with.
Therefore you can use them anywhere you use other
data types (like as input parameters or return
values from a method.)

```
class Car {

}
```

Auto Implemented Properties - simple properties
prop [tab] [tab] [enter] [enter]

this keyword - Access a member ofthe current instance
of the class.

```
public void MyMethod()
{
this.Year = 1976;
}
```

Creating Class Files, Creating Cohesive Classes and Code Navigation (CS-

ASP_037)

==========

====

Prefer more classes w/ narrowly defined responsibilities

Prefer to put each class in its own file

Prefer high cohesion - similarity / signleness of purpose of the class

members

To achieve high cohesion, a rule of thumb: try to make your classes fit on

one "screen" of your IDE (no scrolling required)

Understanding Object References and Object Lifetime (CS-ASP_038)

==========

An object reference variable holds a reference to an instantied object

in the computer's memory.

MyClass myObject;

The new keyword creates an instanceof the class and returns the addres of

object in memory to the reference variable.

myObject = new MyClass();

More than one object reference variable can hold an address to the object in

memory.

MyClass myOtherObjectReference = myObject;

Each time a new reference is added, the reference count increases by one.

Each time

an object reference variable goes out of scope or is set to null, the

reference

count decreases by one.

If the reference count is zero, the .NET Framework Runtime's Garbage

Collector

removes the objet from memory at an indeterminate point in time in the

future. You

can take control of the finalization process and even handle events just

before the

object is removed. See: "deterministic finalization".

Understanding the .NET Framework and Compilation (CS-ASP_039)

==========

The .NET Framework consists of:

- Runtime (Common Language Runtime, CLR) "protective bubble", manages

memory,

protects the user's machine, etc.

- .NET Framework Class Library (FCL, Base Class Library, BCL) - thousands

of classes built by Microsoft for every imaginable purpose.

- Compilers (C# compiler, VB compiler) - turns your human readable source

code into

Microsoft Intermediate Language (MSIL, IL) and packaged into a .NET assembly

(.exe - executable, or .dll - class library)

- Many other tools and features

Initial compilation to Intermediate Language, then a second compilation

JIT - Just In Time compilation - an optimized version of the assembly for

the

specific hardware and software. Happens at first request on that computer.

Understanding Namespaces and the using Directive (CS-ASP_040)

==========

Namespaces disambiguate class names inside of class libraries or

applications.

You must reference class names by their full name:

System.Text.Stringbuilder sb = new System.Text.StringBuilder();

... or, you can employ a using directive at the top of the code file to instruct the compiler to look in those namespaces to find the class that is referenced.

using System.Text;
. . .
StringBuilder sb = new StringBuilder();

You must always do this if the code you're writing is outside of the namespace of the class you want to use, even if it's in the same project.

Default namespace defined in Project Properties (right-clicking on Poject name in Solution Explorer, select Properties ...)

Creating Class Libraries and Adding Reference to Assemblies (CS-ASP_041)
==========
Class Library project - creates a .dll that can be referenced in other

projects.

Add a Reference - the FCL is split into tiny pieces, and you must reference the assemblies that contain the parts of the library you want to use.

Right-click project's References node in Solution Explorer, select Add

Reference

Accessibility Modifiers, Fields and Properties (CS-ASP_042)

==========

http://v.gd/access

Public - Class or member can be accessed by any code

Private - Class or member can only be accessed by parent class

Protected - Class or member can only be accessed by parent class or derived

class

Internal - Class or member can only be accessed by code inside the same

assembly

Classes are internal by default

Methods and properties are private by default

Encapsulation - hiding implementation behind npublic interfaces, reduces

coupling

increases plug-ability / resuability, maintainability, etc.

private fields have two purposes:

(1) reference to object or variable that used for internal implementation of

class

(2) hold the state of an object, backing field for public property

propfull [tab] [tab]

```
private int myField;
public int MyProperty
{
get { return myField; }
set {
if (value > 100)
myField = value;
else
// tell the caller that they can't do this
}
}
```

Full property definition and private fields to control

access to private fields / state of the object.

propg [tab] [tab]

```
public int MyProperty { get; private set; }
```

Restricts setting of property to just the class' internal implementation

Creating Constructor Methods (CS-ASP_043)

==========

Constructors are called at the moment of instantiation.

Used to put the new instance of the class into a valid state.

```
public class Foo
{
public Foo()
{
...
}
}
```

Whether you define it or not, there's a default constructor.

You can override the default (no input parameters) or

overload the constructor to allow the user to set the new

instance to a valid state.

Naming Conventions for Identifiers (CS-ASP_044)

==========

PascalCase - public

camelCase - private, protected

Public classes, methods and properties - PascalCase

Private helper methods, input parameters - camelCase

Locally scoped variables - camelCase

Private field - camelCase prefixed w/ underscore: _firstName

Choose long, memorable, understandable names

that convey meaning / intent.

Static versus Instance Members (CS-ASP_045)

==========

Static members - no instance of the class required to call method

Instance member - must create an instance w/ new keyword to call

methods and properties

Can mix both in the same class, but can't reference instance

members from inside of static members.

Classes can be decaoted w/ static keyword ... all members

must be static, can't create a new instance of that class,

System.Math

http://v.gd/static

Working with the List Collection (CS-ASP_046)

==========

Use Generic Collections to work with items in a strongly typed fashion.

Better than arrays:

Know the type of the item for a certainty, no casting / converting

Better performance inserting / removing / updating

Collections provide more flexible otions to access items in the collection.

Allows for LINQ extension methods

Many different collections - specialties

"Generic Collections"

List<T>

Dictionary<TKey, TValue>

T => data type you need

"You make a generic specific by providing a data type."

List<string> - only store strings (strong typed)

List<Car> - only store Cars in that collection

```
// Assume I have three objects: car1, car2, car3
List<Car> cars = new List<Car>();
cars.Add(car1);
cars.Add(car2);
cars.Add(car3);

int numberOfCars = cars.Count;
Car myCar = cars.ElementAt(1); // Return 2nd car in the collection

// Terminology: You access a MEMBER of a collection

// LINQ queries
```

Object Initializers (CS-ASP_047)

==========

Concise way to initialize a new object (or collection) with values.

```
// Didn't talk about this form:
```

```
Car car1 = new Car() { Make = "BMW", Model = "528i", Year = 2010, Color =

"Black" };

// No local variable name for the new Car instance needed!
cars.Add(new Car() { Make = "BMW", Model = "528i", Year = 2010, Color =

"Black" });
```

Collection Initializers (CS-ASP_048)

==========

Shortcut to create new instance of a generic collection AND initialize it by IMMEDIATELY adding new instances of a given type.

```
List<Car> cars = new List<Car>() {
new Car{Make="BMW", Model="528i", Color="Black", Year=2010},
new Car{Make="BMW", Model="745i", Color="Black", Year=2005},
new Car{Make="Ford", Model="Escape", Color="White", Year=2010},
};
```

Working with the Dictionary<Tkey, TValue> Collection (CS-ASP_049)

==========

Dictionary allows you to use a key to access a member of the collection.

Think: Webster's dictionary ... the word (key), then the definition

(instance of a given type)

Key is anything meaningful in YOUR system.

Key must be unique.

TKey => type of the key

TValue => type of the value

```
Dictionary<string, Car> cars = new Dictionary<string, Car>();
cars.Add("V123", new Car{Make="BMW", Model="528i", Color="Black",

Year=2010});
cars.Add("V234", new Car{Make="BMW", Model="745li", Color="Black",

Year=2010});
cars.Add("V345", new Car{Make="Ford", Model="Escape", Color="White",

Year=2010});

cars.ElementAt(1).Key // Return "V234"
cars.ElementAt(1).Value // Return the Car object in the 2nd position

// Beter way to access Dictionary ...
Car v2;
if (cars.TryGetValue("V234", out v2))
{
result += v2.Year;
}

// Remove
if (cars.Remove("V345")) {
result += "Successfully removed car.";
```

}

Looping with the foreach Iteration Statement (CS-ASP_050)

==========

More elegant way of iterating through collections.

Code snippet: foreach [tab] [tab]

```
foreach (Car car in cars) {
result += car.Make;
}
```

Implicitly Typed Local Variables with the var Keyword (CS-ASP_051)

==========

(Applies to locally scoped variable declaractions)

Compiler is smart enough to figure out the data type

when you initialize the variable.

Becomes increasingly important because sometimes it's

difficult to know what the data type is supposed to be. (LINQ)

```
int hitPoints = 0;
```

... is the equivalent of ...

```
var hitPoints = 0;
```

```
string heroName = "Pentagorn";
```

... is the equivalent of ...

```
var heroName = "Pentagorn"


var cars = new List<Car>() {
...
}
```

a

Rules:

1. Must initialize the variable.

2. Variable is permanently set to the implicit data type.

3. Can't be used for a PUBLIC property / variable

Creating GUIDs (CS-ASP_052)

==========

Globally Unique Identifier

```
System.Guid newGuid = System.Guid.NewGuid();
```

Working with Enumerations (CS-ASP_053)

==========

A data type accepting only enumerated values that you define.

Strongly typed, ridding you app of "magic strings".

```
public enum Occupation {
Doctor,
Lawyer,
IndianChief
}
```

```
Occupation whatIDo = Occupation.IndianChief;

Occupation occupation;
if (Enum.TryParse("Doctor", out occupation)) {
...
}
```

Creating Constants with the const Keyword (CS-ASP_053b)

==========

Remove magic values (strings, integers, etc.) using
permanent, immutable identifiers.
Only use for things that NEVER change - not for product prices, etc.
Define const at local or field.

```
const double valueOfPi = 3.14;
```

Understanding the switch Statement (CS-ASP_054)

==========

Think: train switch ... logic based on evaluation
of a variable or property value.

```
swtich (whatIDo) {
case Occupation.Doctor
...
break;
case Occupation.IndianChief
case Occupation.Lawyer
...
goto case Occupation.Doctor;
```

default:

...

break;

}

First Pass at the Separation of Concerns Principle (CS-ASP_055)

==========

Separate concerns to mitigate the impact of change on a software system.

Common "concerns":

- Presentation logic

- Business / Domain logic

- Persistence logic

Recommendation: Separate concerns into projects within a given solution.

Naming Convention:

MyApplication (Solution name)

-- MyApplication.Presentation (.Web, etc. presentation project)

-- MyApplication.Domain (domain / business rules project)

-- MyApplication.Persistence (.Data, .DB persistence project)

Understanding Exception Handling (CS-ASP_056)

==========

Wrap try ... catch around code:

- That you are calling into, that you didn't write

- Code that accesses external resources

- Code that accepts input from outher sources

```
try {

}
catch (SpecificException ex) {
// catch more specific exceptions first
// more general last
}
catch (Exception ex) {
// log it
// swallow it?
// re-throw it
}
finally {
// optional ... clean up
}
```

Understanding Global Exception Handling (CS-ASP_057)

==========

Unhandled exception bubble up until they are exposed to the

end user (aka "yellow screen of death")

Best place to handle exceptions is the nearest locale

to the exception itself. However, you CAN handling globally.

In global.asax

Add:

```
void Application_Error(object sender, EventArgs e)
{
// What just happened?
Exception ex = Server.GetLastError();

// ex will always be of type HttpUnhandledException.
// To get to the exception that CAUSED that to happen
// you'll need to look at the ex.InnerException.
var innerException = ex.InnerException;

// Handle a specific type of error differently.
if (innerException.GetType()
== typeof(ArgumentOutOfRangeException))
{
Server.Transfer("Error.aspx");
}

// You must do this if you want to hide the
// yellow page of death ... any existing exceptions
// after this point will send the end user the
// exception page.
Server.ClearError();
}
```

Understanding Custom Exceptions (CS-ASP_058)

==========

Inherit from Exception like so:

public class MyCustomException : Exception

{ ... }

Creating a Database in Visual Studio 2013 (CS-ASP_059)

==========

LocalDb - Local dev-only version of SQL Server

Project > Add New Item ... > Installed > Data > SQL Server Database

Creates an .mdf file

SQL Server Data Tools (SSDT) - Tools to create and manage SQL Server

databases from Visual Studio.

Creating an Entity Model in Visual Studio 2013 (CS-ASP_060)

==========

Entity Data Model - Object-Relational Mapper to treat database tables w/

columns as classes w/ properties

Project > Add New Item ... > Installed > Data > ADO.NET Entity Data Model >

Entity Data Model Wizard > EF Designer from Database > Connection > Database

Objects

DbContext == Handle to the entity model > database

DbSet == Collection of all entities in the DbContext

```
ACMEEntities db = new ACMEEntities();
var dbCustomers = db.Customers;
```

Displaying the DbSet Result in an ASP.NET GridView (CS-ASP_061)

==========

GridView Server Control - Databinds to enumerable collections of objects and

renders in a tabular format

Must call ToList() on a DbSet to bind to a databound control.

```
gridControl.DataSource = dbCustomers.ToList();
gridControl.DataBind();
```

Implementing a Button Command in a GridView (CS-ASP_062)

==========

Click Chevron => GridView Tasks > Edit Columns...

BoundField - Databind to a object property

ButtonField - Hyperlink button

Handle button click in the GridView_RowCommand event handler.

```
protected void GridView1_RowCommand(object sender, GridViewCommandEventArgs

e)
{
// Retrieve the ROW CLICKED in the grid
int index = Convert.ToInt32(e.CommandArgument);
GridViewRow row = GridView1.Rows[index];

// Accessing cells is risky because the order
// of the columns may change over time
// (and you might forget that this code
// depends on it!)
// Also ... 0 based!
var someValue= row.Cells[1].Text;

}
```

Using a Tools-Centric Approach to Building a Database Application (CS-ASP_063)

==========

Tools-Centric approach / workflow = Use Visual Sudio's designers, tools,

etc. to build applications w/ minimal code.

Great for small, departmental apps with very little business logic, change

is not anticipated and there's a tight timeframe.

Using a Maintenance-Centric Approach to Building a Database Application (CS-ASP_064)

==========

Maintenance-Centric approach / workflow = Anticipate change, mitigate it's

negative effects on software by separating concerns, applying unit testing,

etc.

Great for larger, enterprise scale apps with many business rules, where

change is anticipated because it is crucial to the operation of the business

and there's a longer development timeframe.

DTO - Data Transfer Object -- model used for transfering
from one layer to another to avoid a leaky abstraction.
Ex., I don't want Entity Framework leaking out of
persistence because other layers would be dependent on
it!

Creating a New Instance of an Entity and Persisting to the Database (CS-

ASP_065)

==========

var customer = new Customer();

```
// Populate properties of customer
dbCustomers.Add(customer);
db.SaveChanges();
```

Filtering DbSets using LINQ method syntax:

```
ACMEEntities db = new ACMEEntities();
var dbCustomers = db.Customers.OrderBy(p => p.Name).ToList();

.Where(p => p.Name == "Bob").ToList();
```

Lambda Expression - "mini methods"

Package Management with NuGet (CS-ASP_066)

==========

Package Manager - installs files, folders, adds references to third party

packages supplying common functionality.

Adds dependencies.

Updates packages and dependencies.

Tools > NuGet Package Manager > Manage NuGet Packages for Solution ... >

Manage NuGet Packages Dialog

Tools > NuGet Package Manager > Package Manager Console

Install-Package elmah

Update-Package elmah

packages.config file keeps track of packages, dependencies and versions.

NuGet No-Commit Workflow (CS-ASP_067)

==========

To restore deleted packages at comppile:

Solution Explorer > Right-click Solution > Enable NuGet Package Restore

Adds a .nuget folder in the project.

Or, go to the Manage NuGet Packages dialog and click the Restore button at

the top (yellow area).

Introduction to the Twitter Bootstrap CSS Framework (CS-ASP_068)

==========

Add:

<link href="Content/bootstrap.min.css" rel="stylesheet" />

<script src="Scripts/jquery-2.1.3.min.js"></script>

<script src="Scripts/bootstrap.min.js"></script>

Complete Reference:

http://getbootstrap.com/css/

<label>TextBox: </label>

<asp:TextBox ID="testTextBox" runat="server" CssClass="form-control">

<asp:Button ID="testButton" runat="server" Text="Test" CssClass="btn btn-lg

btn-primary" />

Mapping Enum Types to Entity Properties in the Entity Framework Designer (CS-ASP_069)

==========

Enums have implicit numeric indices

public enum BallType {

Baseball = 0,

Basketball = 1,

Football = 2

}

You can change the "seed":

// Implies 1, 2, 3

public enum BallType {

Baseball = 1,

Basketball,

Football

}

The Entity Framework Designer maps enums to int data types.

In Entity Data Model diagram (edmx)

- Right-click Column

- Select Convert to Enum

- In the Add Enum Type dialog, either:

-- Select Reference external type, enter namespace + type, or...

-- Enter Enum Type Name, enter Members and optional Values

Deploying the App to Microsoft Azure App Services Web Apps (CS-ASP_070)

==========

Solution Explorer > Right-click Project Name > Publish ...

MAKE SURE YOU DELETE THE PUBLISHSETTINGS BEFORE DISTRIBUTING YOUR SOURCE CODE TO OTHERS!!!