Programming 101

Contents

[Introduction 3](#_Toc72351291)

[Terminology 3](#_Toc72351292)

[Basic Variable Types 3](#_Toc72351293)

[Variable Declarations 3](#_Toc72351294)

[Variable Scope 3](#_Toc72351295)

[Code Comments 3](#_Toc72351296)

[Our First Application – Hello World 4](#_Toc72351297)

[Operators 4](#_Toc72351298)

[Working with Strings 5](#_Toc72351299)

[Conditional Logic 6](#_Toc72351300)

[if/else if/else 6](#_Toc72351301)

[switch Statement 6](#_Toc72351302)

[Conditional Logic Scope 7](#_Toc72351303)

[Methods 7](#_Toc72351304)

[Static vs Instance Methods 7](#_Toc72351305)

[Properties 7](#_Toc72351306)

[Common Methods and Properties 8](#_Toc72351307)

[Random Numbers 9](#_Toc72351308)

[String Formatting 9](#_Toc72351309)

[Arrays 10](#_Toc72351310)

[Accessing Array Data 10](#_Toc72351311)

[Array Properties and Methods 10](#_Toc72351312)

[Array Performance 10](#_Toc72351313)

[Arrays and Strings 10](#_Toc72351314)

[Splitting a String 11](#_Toc72351315)

[Loops 11](#_Toc72351316)

[for Loop 11](#_Toc72351317)

[while Loop 11](#_Toc72351318)

[do while Loop 12](#_Toc72351319)

[foreach Loop 12](#_Toc72351320)

[Controlling Loops 12](#_Toc72351321)

[Infinite Loops 13](#_Toc72351322)

[Loop Scope 13](#_Toc72351323)

[Multi-Dimensional Arrays 13](#_Toc72351324)

[Traversing Multi-Dimensional Arrays 13](#_Toc72351325)

[Collections 14](#_Toc72351326)

[List<T> 14](#_Toc72351327)

[Dictionary<Tkey, Tvalue> 14](#_Toc72351328)

[HashSet<T> 14](#_Toc72351329)

[Queue<T> 15](#_Toc72351330)

[Stack<T> 15](#_Toc72351331)

[System.Linq 16](#_Toc72351332)

[Input / Output 16](#_Toc72351333)

[File System Terminology 16](#_Toc72351334)

[Working with Paths and Filenames 16](#_Toc72351335)

[Dealing with Backslash 16](#_Toc72351336)

[System.IO Library 17](#_Toc72351337)

[Accessing Directories 17](#_Toc72351338)

[Accessing Files 17](#_Toc72351339)

[File System Search Paths 18](#_Toc72351340)

[File Streams 18](#_Toc72351341)

[Object Oriented Programming 19](#_Toc72351342)

[Class 19](#_Toc72351343)

[Class Constructors 19](#_Toc72351344)

[Inheritance 19](#_Toc72351345)

[Inheritance Protection Levels 20](#_Toc72351346)

[Method 20](#_Toc72351347)

[Property 20](#_Toc72351348)

[Interface 21](#_Toc72351349)

[Child Classes and Interfaces 21](#_Toc72351350)

[Using Inheritance and Interfaces 21](#_Toc72351351)

[Enum 21](#_Toc72351352)

[Exceptions 22](#_Toc72351353)

[Persisting Class Data 22](#_Toc72351354)

[.Net and JSON 22](#_Toc72351355)

# Introduction

## Terminology

* **Variable** Stores a value

**var a** = 5;

* **Class** Contains functionality and values

public class **App** { }

* **Library** Set of Classes that can be referenced in your code

using **System**;

* **Method** Block of code that performs some action(s)

a.**ToString()**;

* **Property** Variable that can be accessed inside a Class

DateTime.**Now**;

"foo" **+** "bar";

* **Operator** An action performed on one or more values

## Basic Variable Types

* string "Hi my name is Jack"
* char 'a'
* int -132
* bool true

double for Math – greater range, less precise  
decimal for Money – smaller range, more precise

* double 765343.54
* decimal 7241.21m
* DateTime 2020-10-03 14:32

## Variable Declarations

* type variableName; Value not set & Type declared

int birthYear;

* type variableName = value; Value set & Type declared

decimal profit = 5280.32m;

* var variableName = value; Value set & Type inferred

var isSaved = true;

## Variable Scope

* Variables are only accessible inside the block of code where they are declared
* The scope of a variable is the block the variable is in
* Curly Brackets **{** **}** define blocks
* Indentions (tabs) are used emphasize code blocks
* Nothing inside brackets should line up with a bracket except its matching closing bracket

## Code Comments

/\*

Multiline Comment. This block can contain multiple lines of text and can be used to block out large parts of code

\*/

//Single line comment. Just keep

//adding more slashes to continue

//adding more comments

## Our First Application – Hello World

Tells C# what library we are using

Defines a Class

Defines the Main Method  
  
Prints "Hello World" to the screen  
Tells C# to wait for input

using System;

public class HelloWorld

{

public static void Main()

{

Console.WriteLine("Hello World");

Console.ReadLine();

}

}

# Operators

An action performed on one or more values

Operators can act on

* Methods
* Variables
* Properties
* Values

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Assignment** | Set | **=** | var a = 5; |  |  |
| **Math** | Add | **+** | var b = a + 3; | b=8 |  |
|  | Subtract | **-** | var c = b - a; | c=3 |  |
|  | Multiply | **\*** | var d = c \* 5; | d=15 |  |
|  | Divide | **/** | var e = a / c;  var f = 5.0 / c; | e=1  f=1.6 | Dividing **ints** drops decimal values |
|  | Modulus | **%** | var g = b % 2; | g=1 | Remainder |
|  | Increment | **++** | a++; | a=6 |  |
|  | Decrement | **--** | b--; | b=7 |  |
|  | Increment by x | **+=** | a+=10; | a=16 |  |
|  | Decrement by x | **-=** | b-=2; | b=5 |  |
|  |  |  |  | |  |
| **Strings** | Concatenate | **+** | var str = "foo" + "bar"; | |  |
|  | Char Access | **[ ]** | var char2 = str[2]; | char2='o' | |
| **Boolean** | And | **&&** | var h = true && false; | h=false |  |
|  | Or | **||** | var i = true || false; | i=true |  |
|  | Equals | **==** | var j = i == true; | j=true |  |
|  | Not Equals | **!=** | var k = h != false; | k=false |  |
|  | Not | **!** | var l = !k; | l=true |  |
|  | Less Than | **<** | var m = 5 < 10; | m=true |  |
|  | Greater Than | **>** | var n = 10 > 5; | n=true |  |
|  | LT or Equal | **<=** | var p = 10 >= 9; | p=true |  |
|  | GT or Equal | **>=** | var q = 1 >= 5; | q=false |  |
| **DateTime** | Difference | **-** | var ts = dt1 - dtNow; | Returns a TimeSpan | |

## Working with Strings

There is a difference between

* **"124"** The **string** containing the *characters* **1 2 4**
* **124** The **int** with a *value* of **one hundred twenty-four**

Use **ToString()** to get a string from an int (or another type)

var valInt = 232;

var valStr = valInt.ToString();

Use **int.Parse()** to get an int from a string

var valStr = "598";

var valInt = int.Parse(valStr);

# Conditional Logic

Boolean operators can be used to conditionally execute code

C# has 4 conditional logic statements which evaluate a Boolean expression and then execute the associated code

* if
* else if
* else
* switch

## if/else if/else

* else if and else are optional
* The first if or else if that matches will run and no other statement will execute
* The else will only execute if none of the if and else if statements match

var a = Console.ReadLine();

if(a == "one")

{

Console.WriteLine("User typed one!");

Console.WriteLine("1");

}

else if(a == "five")

Console.WriteLine("5");

else if(a == "six")

Console.WriteLine("6");

else

Console.WriteLine(a);

## switch Statement

* Each case must have a break at the end
* The same code can execute for multiple case statements
* default is optional and will run if no case statements match

var a = Console.ReadLine();

switch(a)

{

case "one":

Console.WriteLine("1");

break;

case "five":

Console.WriteLine("5");

break;

case "six":

case "seven":

Console.WriteLine("six or seven");

break;

default:

Console.WriteLine(a);

break;

}

## Conditional Logic Scope

var a = Console.ReadLine();

if(a == "5")

{

//Note the curly brackets

//p is only available inside

//this if statement

var p = int.Parse(a);

Console.WriteLine(p \* 2);

}

else //indent the next line

Console.WriteLine(a);

* Conditional Logic statements create new variable scope
* Any variable defined inside a logic statement will only be accessible inside that block
* A single line of code can be run inside a logic statement
  + Indent a single tab to show the code will run inside that statement
* More than one line of code requires curly brackets

# Methods

A block of code that performs an action

* All Variables have Methods
* Methods can have arguments passed in
* There may be multiple Methods with the same name and different number of arguments
* Methods have a Type they return on completion

## Static vs Instance Methods

Some Methods must be called from the Class itself. These are called **Static Methods**. Normally they create new objects or have stand-alone functionality

var i = int.Parse(Console.ReadLine());

Others Methods are called from an instance of a Class. These normally affect the instance itself and are thus called **Instance Methods**.

var str = intVal.ToString();

# Properties

Variable that can be accessed in an instance of a Class

* Most Variables have Properties
* Properties have a Type
* There are Static and Instance Properties just like Methods

# Common Methods and Properties

|  |  |  |
| --- | --- | --- |
| **String** | Length | int l = strVal.Length; |
|  | Substring(int start) | string ss = strVal.Substring(3); |
|  | Substring(int start, int length) | string ss = strVal.Substring(3,2); |
|  | Replace(string old, string new) | string r = strVal.Replace("ar", "aa"); |
|  | StartsWith(string value) | bool sw = strVal.StartsWith("fo"); |
|  | IndexOf(string value) | int i = strVal.IndexOf("bar"); |
|  | IndexOf(string val,int startIndex) | int i = strVal.IndexOf("bar"); |
| **DateTime** | Now | DateTime n = DateTime.Now; |
|  | Year | int year = DateTime.Now.Year; |
|  | AddHours(double hours) | DateTime plus =  DateTime.Now.AddHours(5); |
|  | Parse(string strDateTime) | DateTime p =  DateTime.Parse("5/6/1985"); |
| **TimeSpan** | FromMinutes(double minutes) | TimeSpan ts = TimeSpan.FromMinutes(5); |
|  | Hours | int hrs = ts.Hours; |
|  | TotalHours | double totalHrs = ts.TotalHours; |
|  | Add(Timespan ts) | TimeSpan ts2 =  ts.Add(TimeSpan.FromHours(5.0); |
| **Console** | WriteLine(object o) | Console.WriteLine("Hello World!"); |
|  | Write(object o) | Console.Write("Hello World!"); |
|  | ReadLine() | string s = Console.ReadLine(); |
|  | Clear() | Console.Clear(); |
|  | SetCursorPosition(int row,int col) | Console.SetCursorPosition(5, 20); |
|  | WindowWidth | int width = Console.WindowWidth; |
|  | WindowHeight | int height = Console.WindowHeight; |
|  | ForegroundColor | Console.ForegroundColor =  ConsoleColor.Red; |
|  | BackgroundColor | Console.BackgroundColor =  ConsoleColor.White; |
|  | ResetColor() | Console.ResetColor(); |
| **System.Math** | Math.Pow(double num, double pow) | double d = Math.Pow(5.0, 2.0); |
|  | Math.Abs(int/double num) | var a = Math.Abs(-523.32); |
|  | Math.Max(num1, num2) | var ma = Math.Max(3, 22); |
|  | Math.Min(num1, num2) | var mi = Math.Min(5, 93); |
|  | Math.Floor(double/decimal num) | var f = Math.Floor(72.82); |
|  | Math.Ceil(double/decimal num) | var c = Math.Ceil(63.12m); |
|  | Math.Round(double/decimal num) | var r = Math.Round(85.5); |

# Random Numbers

* Computers can’t generate true random numbers
* Pseudorandom generators take a value and generate a repeatable set of pseudo-random numbers using that value as a seed
* Random seed = more randomness
  + Time

var random = new Random();

int num = random.Next();

* + User input
  + Sensor values

# String Formatting

* Many classes have a ToString() method that accepts a format argument
* Number Formatting
  + **n** Number 3,341.02
  + **f** Decimal 7542.3744359
  + **c** Currency $82,002.12
  + **p** Percent 92.02%
* Date Formatting
  + **M** Month **h** Hour
  + **d** Day **m** Minute
  + **y** Year **t** AM / PM
* Some methods accept multiple arguments and then output them as a string
* Use the format **{**argNum,alignment**:**format**}** to control how the arguments get printed
  + Positive alignment = Pad on Left
  + Negative alignment = Pad on Right
  + Padding will not occur if the value is longer than the padding length

var x = 53215.2513;

x.ToString("f3"); //53214.251

DateTime.Now.ToString("MMM dd, yy h:mm tt"); //Feb 2, 21 2:02 PM

Console.WriteLine("{0,-15:c} Profit"); //$53.214.25 Profit

# Arrays

Store lists of information

* Typed

var nums = new int[5];  
var officeApps = new[] {"Powerpoint", "Excel"};

* Fixed Length
* 0 based indexing

## Accessing Array Data

* Arrays are just variables that can store multiple values
* Values are accessed using Square Brackets **[ ]**
* The Array index is the position you want to work with
* Items in an Array can be retrieved and set

// 0 1 2

var myArray = new[] {"Joe", "Sam", "Keith"};

var item = myArray[1];

myArray[2] = "Mark";

## Array Properties and Methods

|  |  |
| --- | --- |
| Length | int l = arr.Length; |
| IndexOf(Array a, object o) | int idx = arr.IndexOf("Jack"); |
| Copy(Array source, Array dest, int length) | Array.Copy(arr1, arr2, arr1.Length); |

## Array Performance

* Arrays are very fast to access and set items
* Adding items requires re-initialization or copying the Array
  + Time Consuming
  + Memory Intensive

## Arrays and Strings

strings are Arrays of Type char. Adding characters to a string is "easy" but can cause all sorts of problems. Use System.Text.StringBuilder if you are going to be doing a log of string concatenation.

using System.Text;

...

var sb = new StringBuilder();

sb.Append("a");

string s = sb.ToString();

## Splitting a String

You may sometimes have to deal with data in a file. Often data is stored as Comma Separated Values (CSV). Use string.Split() to work with this data.

var s = "Jack**,**39**,**4/3/1981**,**F1**,**254**,**McCI";

string[] splitString = s.Split(',');

var name = splitString[0]; //Jack

var room = splitString[4]; //254

# Loops

Execute the same code multiple times

## for Loop

* Iterates through the loop a fixed number of times
* Performs an action **before** the loop starts
* Checks a condition **before** each iteration
* Performs an action **after** each iteration

for(preLoopAction; condition; postIterationAction)  
{

//Code to Execute  
}

for(var i = 0; i < 5; i++)  
{

Console.WriteLine("iteration " + i);  
}

## while Loop

* Iterates **until** a condition is **met**
* Checks a condition **before** each iteration

while(condition)  
{

//Code to Execute  
}

var i = 0;

while(i < 5)  
{

Console.WriteLine("iteration " + i);

i++;  
}

## do while Loop

* Iterates **until** a condition is **met**
* Checks a condition **after** each iteration
* Loop runs through 1st time before condition is checked
* Sometimes it is cleaner using do while vs while

do   
{

//Code to Execute  
} while(condition);

var i = 0;

do   
{

Console.WriteLine("iteration " + i;

i++;  
} while(condition);

## foreach Loop

* Iterates through each item in a collection
* Item is assigned to a variable inside loop
* No condition checks
* Collection cannot have items added or removed in loop

foreach(var i in collection)  
{

//Code to Execute  
}

var names = new List<string> { "Jack", "James" };

foreach(var name in names)  
{

Console.WriteLine(name);

}

## Controlling Loops

* break Stop a loop early
* continue Skip to the next iteration

break and continue can be used with all loop types

## Infinite Loops

* If conditions are never met, a loop will run forever
* Your application will crash if not handled
* Always check conditions
* There are appropriate uses of Infinite Loops

while(true)  
{

Console.WriteLine("forever…");

}

## Loop Scope

var arr = new int[5];

for(var i = 0; i < arr.Length; i++)

{

//Note the curly brackets

//i **and** b are only available

//inside this for loop

var b = Console.ReadLine();

arr[i] = int.Parse(b);

}

int c = 0;

while(c < 5) //Indent the next line

Console.WriteLine(++c);

* Loops create new variable scope
* Any variable defined inside a loop will only be accessible inside that loop
* A single line of code can be run inside a loop
* Indent a single tab to show the code   
  will run inside that loop
* More than one line of code requires   
  curly brackets

# Multi-Dimensional Arrays

* Arrays can have multiple dimensions
* Store rows and columns or other sets of data
* Fixed Length
* Typed
* 0 based indexing

0,3

0,2

0,1

0,0

0,0

0,2

0,3

0,1

var a = new int[2,4];

a[0,2] = 5;

## Traversing Multi-Dimensional Arrays

var a = new int[2,10];

for(var x = 0; x < 2; x++)

for(var y = 0; y < 10; y++)

a[x,y] = x+y;

* Loops are just code
* They can be put inside other loops
* Useful for iterating multi-dimensional arrays

# Collections

* .NET provides other ways of storing data
* The Collections libraries provide these Classes
* Collections are defined as generic which allows you to store Types dynamically
* Very high performance
* Most of the time the Collections classes are easier to use than Arrays

## List<T>

Dynamically Expanding Array

* Normally better than Arrays
* Can expand and contract dynamically
* Items can be inserted or removed anywhere in the List

var a = new List<string>();

a.Add("foobar");

a.Add("foo");

var b = a[0];

## Dictionary<Tkey, Tvalue>

Key Value Pair

* Lookup by Tkey and retrieve Tvalue
* Incredibly useful for all sorts of tasks
* Does not allow duplicates

var a = new Dictionary<string, int>();

a["jack"] = 38;

a["joe"] = 20;

var b = a["jack"];

## HashSet<T>

Maintains single unique copy of each item

* Stores only **one** copy of each item
* Good for maintaining unique lists

var a = new HashSet<string>();

a.Add("jack");

a.Add("joe");

a.Add("jack");

var b = a.Count; //b is 2

## Queue<T>

First In, First Out

* Stores and accesses data in the order it was entered
* Can’t access items individually
* Good for items of work that must be accomplished in order

**Queues have three operations**

* **Enqueue()** Add an item
* **Dequeue()** Remove the next item
* **Peek()** Get next item without removing it

var a = new Queue<string>();

a.Enqueue("first");

a.Enqueue("second");

a.Enqueue("third");

var b = a.Dequeue(); //b = "first"

var c = a.Peek(); //c = "second"

## Stack<T>

First In, Last Out

* Stores and accesses data in the **reverse order** it was entered
* Can’t access items individually
* Useful for parsing text and places where order is important

**Stacks have three operations**

* **Push()** Add an item
* **Pop()** Remove the top item
* **Peek()** Look at the top item **without** removing it

var a = new Stack<string>();

a.Push("first");

a.Push("second");

a.Push("third");

a.Pop();

var b = a.Pop(); //b = "second"

var c = a.Peek(); //c = "first"

# System.Linq

Provides query functionality for Collections

* All Linq queries are accessed using Lambda functions
* Lambdas are special Methods that take code as their arguments
* All these methods can be found in Class Explorer under System.Linq.Queryable

# Input / Output

.NET can handle many types of I/O including Keyboard, Mouse, Files, Console, Network and Databases

## File System Terminology

* **Drive** Physical Storage Device
* **Directory** Organizational container on the Drive
* **File** Data stored on the Drive with a name
* **Extension** Part of the filename that defines the type of data
* **Path** Full route to access a File or Directory
* **Stream** .NET classes for reading and writing Files

## Working with Paths and Filenames

* Every Directory has a Path
* All Files will have a Path and a Filename
* Most Files will have an Extension

**Class Directory**

C:\Users\Character\Documents\Class Projects\Programming 101

**HelloWorld Code File**

C:\Users\Character\Documents\Class Projects\Programming 101\Lesson1\HelloWorld.cs

## Dealing with Backslash

* In C# the Backslash **\** character is special
* To use a Backslash in a string you have two options

var path = @"c:\"; @ sign tells .NET to ignore special chars

var path = "c:\\"; \\ escapes the Backslash

* Backslash is normally used to escape other characters
  + Quote **\"** var q = "And I Quote: \"He said he did it\"";
  + Tab **\t** var t = "Space this \t out";
  + Newline **\n** var n = "Line1 \n Line2";

## System.IO Library

Contains .NET Classes for working with the file system

Access and Manipulate

* Directories
* Files
* Paths
* Streams

## Accessing Directories

* NET provides two classes
  + Directory
  + DirectoryInfo
* Both provide similar functionality
* Directory returns strings
* DirectoryInfo returns System.IO classes

**EnumerateDirectories(string directory)**

IEnumerable<string> dirs = Directory.EnumerateDirectories("c:\\");

**EnumerateFiles(string directory, string searchPattern)**

IEnumerable<string> files = Directory.EnumerateFiles(@"c:\MyApp\", "\*.d\*");

**GetCurrentDirectory()**

string currDir = Directory.GetCurrentDirectory();

## Accessing Files

* NET provides two classes
  + File
  + FileInfo
* Unlike Directory and DirectoryInfo functionality is not the same
* File is normally a better fit for this course

|  |  |
| --- | --- |
| ReadAllLines(string filePath) | string[] lns = File.ReadAllLines("file.txt"); |
| ReadAllText(string filePath) | string txt = File.ReadAllText("file.txt"); |
| ReadLines(string filePath) | IEnumerable<string> iter =  File.ReadLines("file.txt"); |
| WriteAllLines(string filePath, string[] s) | File.WriteAllLines("file.txt", arrStrLines); |
| WriteAllText(string filePath, string s) | File.WriteAllText("string to write to file"); |

## File System Search Paths

**\*** Match Anything  
**?** Match a single character

* Word Files (.docx extension) \*.docx
* Anything containing "jack" \*jack\*
* 3 character filenames ???.\*
* Files starting with A A\*.\*

## File Streams

* Streams allow developers to read and write data in pieces
* More efficient than reading / writing the whole file at once
* Streams must be handled carefully or memory issues can arise
* The using keyword should normally be used to correctly handle opening and closing Streams
* using blocks create variable scope just like loops and conditional statements

using(var sw = new StreamWriter("myFile.txt"))

{

sw.WriteLine("first line");

sw.WriteLine("second line");

sw.Write("third line");

sw.WriteLine("still third line");

}

using(var sr = new StreamReader("myFile.txt"))

{

//EndOfStream is true when there

//is no more data to read

while(!sr.EndOfStream)

{

//Reads a line and advances

//to the next line

string str = sr.ReadLine();

Console.WriteLine(str);

}

}

# Object Oriented Programming

Developing in a way that models real objects

## Class

Basic Unit of OOP

* A Class is an organizational unit for
  + Methods
  + Variables
  + Properties
* Keep Classes small and uncomplicated
* Think about what you are trying to model first

### Class Constructors

* Every Class has a Constructor method that allows you to create an instance of the Class
* Classes can have multiple Constructors
* Any Property and Variable initialization should be performed in the Constructor(s)
* Constructors are always named the same as the Class with no return Type
* A Class can have multiple Constructors with different arguments
* Use the this keyword to consolidate code and pass arguments into another constructor

public class Cow

{

public Cow(int weight, decimal price)

{

...

}

public Cow(int weight) : this(weight, 500.0m)

{

...

}

}

## Inheritance

Allows one Class to expand on the functionality of another

public class Animal //Base Class

{

...

}

public class Cow : Animal //Child Class

{

...

}

* Every Class starts off inheriting from object
* .ToString(), .Equals(), .GetType() inherit from object
* Methods, Properties and internal Variables inherit from Parent classes
* Child Methods and Properties can also override their Parent’s definitions to provide new and extra functionality
* Each Class can only inherit from a single Parent
* Child class Constructors can call their Parent constructors using the base() much like calling another Constructor using this() in the same class

### Inheritance Protection Levels

* Every Class, Method and Property must be defined with a protection level
  + **public** Accessible from any other class
  + **protected** Accessible only in child (derived) classes
  + **private** Accessible only inside the defining class
* Allows the developer to control what functionality they want to expose to other Classes and developers
* These are not to prevent hacking – they just define how child classes can use these objects

## Method

Block of code that performs an action

* A Class can have as many Methods as needed
* Methods have
  + Return Type (Can be void if nothing returned)
  + Name
  + Arguments
* Name and Arguments should be easily understood
* The return keyword can be used to return a value and exit the method at any point
* Methods can have multiple definitions with different arguments called overloads
* To add additional functionality to a Parent Method or Property create a Method or Property in the child with the same definition
* Use the base keyword to call a Parent’s method from a Child

## Property

private DateTime d;

public string DateString {

get {

return d.ToShortDateString();

}

set {

d = DateTime.Parse(value);

}

}

Accessible value inside a Class

* A Class can have as many Properties

as needed

* Properties have
  + Type
  + Name
  + Getter
  + Setter (optional)
* Name should be easily understood
* The get and set can run code in addition to just returning and setting the value
* Allows the developer to add logic to these operations
* If no code is added, then the get and set just return and set the value
* These operations must be defined with a protection level

## Interface

Defines a way that a group of Classes operate

* Define specific Methods and Properties that a Class must have
* Work with a set of Classes in a predefined way
* Define a set of actions that can span different Classes
* A Class can implement multiple Interfaces
* Naming scheme should be **IInterfaceName**
* Interfaces don’t define any Method or Property bodies
* Only define the gets and sets you need

### Child Classes and Interfaces

* Child classes are also instances of their Parent/Base class
* A Variable of the Parent type can store any Child type
* Interfaces work the same way
* Interfaces can inherit from other Interfaces

var foo = new List<Human>();

foo.Add(new Boy());

foo.Add(new Girl());

### Using Inheritance and Interfaces

* Use the is keyword to check if a type is what you expect

bool isChild = foo[0] **is** Child;

* Get the Type of an instance by calling .GetType() and typeof

bool isAdult = foo[1].**GetType()** == **typeof**(Adult);

* Convert a class into a related class by casting

var kid1 = foo[0] **as** **Child**; //Safe Cast

var kid2 = **(Child)**foo[0]; //Unsafe Cast

## Enum

Set of predefined values

* Allow you to have predefined values for settings in your classes
* Easy way to limit a developer to only what you want them to set

private enum StreetType { Ave, Ct, St, Way };

## Exceptions

* Exceptions are thrown when by .NET when   
  things go wrong

try

{

throw new Exception("ouchies!");

}

catch(Exception ex)

{

Console.WriteLine(ex);

}

* Catch and handle using try/catch blocks
* The code inside the try executes first
* If an Exception occurs, the catch block   
  will execute
* Code inside the try is only accessible inside   
  the try block
* try and catch both create variable scope
* Developers can throw their own Exceptions to out of bounds situations
* Create a class that derives from Exception
* Exception classes and have additional Methods and Properties just like any other child class

## Persisting Class Data

* Class data can be persisted many ways

{

"account": 1243,

"name": "Joe",

"records": [

345,  
 643,

992

],

"address": {

"line1": "12 Main",

"city": "Rome",

"state": "GA"

}

}

* + CSV
  + Database
  + Serialization
* Serialization involves converting an object into a string
* **J**ava**S**cript **O**bject **N**otation (JSON) is a standard way of serializing objects
* JSON is the most widely used method of passing data on the Internet and other applications
* Almost any type of data can be serialized into JSON
* Easily be read and written to files, streams and Internet

### .Net and JSON

* JsonConvert is in the Newtonsoft.Json Library
* DeserializeObject is a generic method
* Pass in the Type you want to deserialize the JSON into

using Newtonsoft.Json;

...

var acct1 = new Account();

var strSerialized = JsonConvert.SerializeObject(acct1);

var acct2 = JsonConvert.DeserializeObject<Account>(strSerialized);

Console.WriteLine(acct2.AccountNumber);