Basics

Just some notes on basic C#, as I have recently been learning .NET

Microsoft - Tour of C#

preproccessor directives

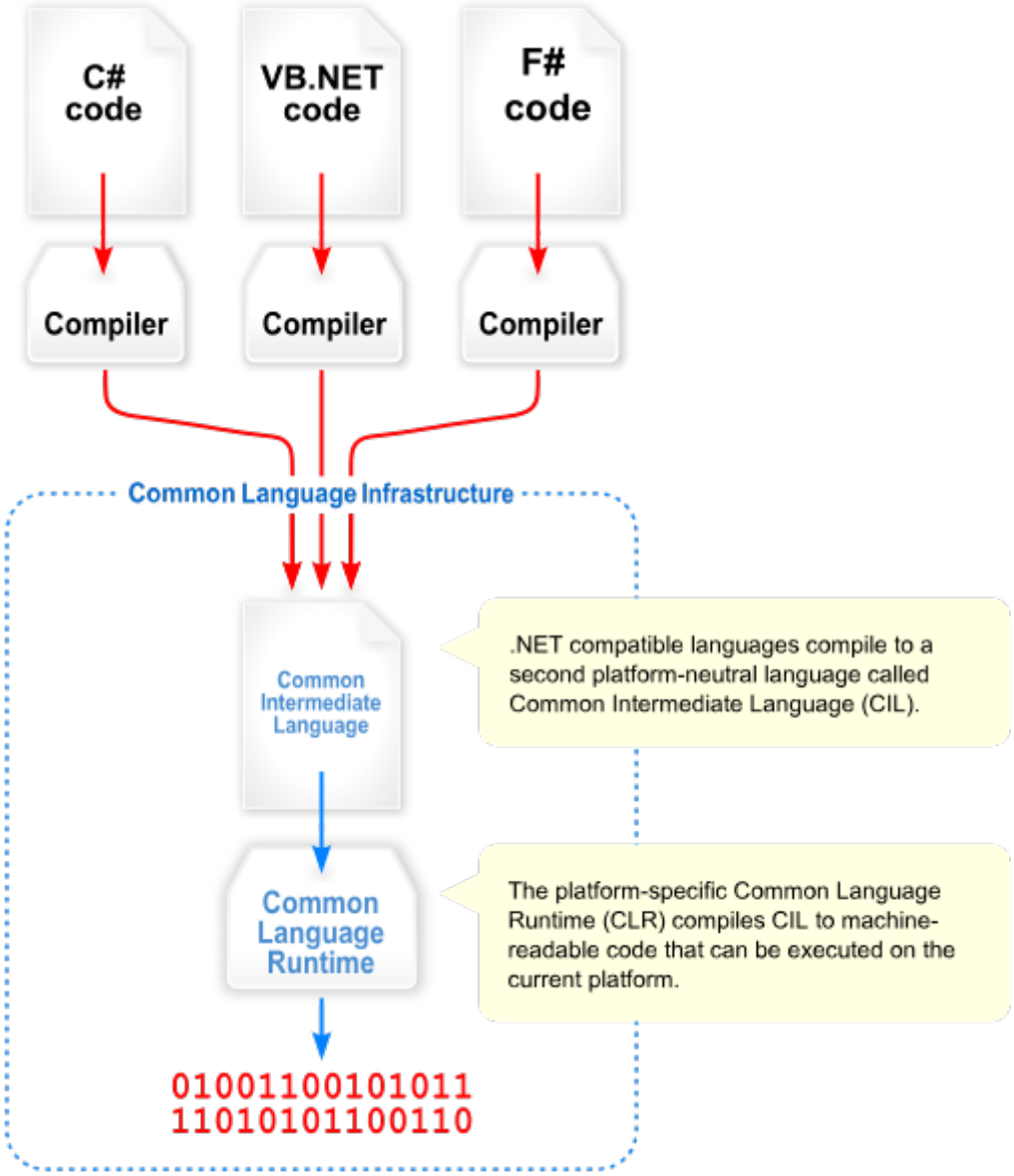
Architecture

You can download .NET as an SDK or a runtime. The SDK is for developing applications, while the runtime is for executing applications that have been developed with the .NET SDK. The SDK includes runtimes required to run .NET applications. The .NET Framework consists of the Common Language Runtime (CLR) and the .NET class library that provides basic classes and types for use with Common Intermediate Language (CIL) languages.

More information on contents of the SDK and .NET runtime, and Runtime Libraries

Compiling to Microsoft Intermediate Language (MSIL). MSIL is also known as Common Intermediate Language (CIL), or Intermediate Language (IL). These terms are all referring to the same platform independent instructions produced by .NET languages after compilation, which exist under the Common Language Infrastructure (CLI) specification. This IL comes in the form of `.exe` or `.dll` files.

These files that adhere to the CLI are then used by the JIT compiler to produce native code for the platform that runs the CLR, and the native code can then be executed on this platform. The following diagram helps to understand this process visually.



Credit to the Author on Wikipedia

Common Language Runtime (CLR) is a virtual machine that compiles IL into native instructions using a Just In Time (JIT) compiler and executes applications natively. The JIT compiler translates Intermediate Language (IL) from compiled C# into native machine code that the processor understands. C# is just used as an example here, and is not the only .NET language that compiles to IL. The JIT compiler has a feature called Tiered Compilation which enables the recompilation of individual methods at run time, which in turn supports quick compilation of large applications.

Mono provides a cross-platform implementation of the CLR, and also provides us with a C# compiler to produce IL. More information on C# compilers, and the benefits of using Mono

The CLR is responsible for Automatic Memory Management through the process of Garbage Collection

It is possible to use unmanaged resources within a .NET application. For example, a FileHandle attached to a FileStream must be explicitly released by the caller. The FileStream itself is however a managed object. Unamanged objects implement the IDisposable interface. IDisposable objects call the `Dispose()` method which releases any contained unmanaged resources.

More information on cleaning up unmanaged resources

More .NET Terminology

.NET uses NuGet for package management, which can be installed with `sudo apt install nuget` on Ubuntu 20.04. Using NuGet is easy to figure out via the `nuget` CLI command and it's help menus, but one should also read up on Managing Dependencies and Package Restoration

Garbage Collection

The GC manages memory by allocating a contiguous section of memory for a new process. Using a pointer to the base adress of this section in memory, the GC can allocate new blocks of data for objects within managed memory. As each new object is created and memory is allocated to it, the pointer moves from the base of the managed memory block to the end of the last object allocated. Because this pointer is managed in this way, allocating new objects on the managed heap is faster than allocating objects in unmanaged memory. Because the block of managed memory is contiguous and we know all objects within it are before our pointer, accessing these objects is also fast and efficient.

The GC determines when it should clean up unused objects, and automatically kicks off this process. To determine which objects can be freed from managed memory, the GC requests an application's roots, which includes all variables and fields in an application, and builds a graph of all reachable objects within the application. The GC then compares this graph to the objects in managed memory, and frees objects that are not reachable from any point in the application. To free memory of an unused object, the GC uses a memory-copying function to compact the reachable objects in memory over the unused objects. This process both frees the memory taken up by the objects and ensures the objects that are still in use remain at the top of the managed memory block. The GC then corrects pointers to the objects, updating the graph locations to the new locations in memory, and places the managed heap's pointer at the last memory address used by existing objects. This process of memory compaction is only triggered when the GC discovers a signifigant amount of unreachable objects - if all objects remain within the application there is no need to compact memory.

The GC allocates large objects in a seperate heap, and automatically releases these objects as needed. To avoid copying large objects in memory, there is no compacting applied to this block of memory. By default the large object heap stores objects greater than 85,000 bytes, but this threshold can be configured if required.

GC Algorithm Generations explains the 0, 1, and 2 generations applied to objects within the managed heap. Generation 0 contains the newest objects, 1 is short-lived objects, and 2 is long-lived objects. By separating objects into these categories, the GC can avoid compacting the entire heap each time it frees up unused objects. For example, a generation 0 collection only requires the compacting of the generation 0 block of memory, and the blocks for 1 and 2 remain the same, so the total work required is reduced. When an object survives a generation 0 collection it is promoted to generation 1. When an object survives a generation 1 collection it is promoted to generation 2. Objects that survive generation 2 collections remain in generation 2.

Concurrent garbage collection is applied to workstation .NET 3.5 and earlier, as well as .NET server 4.0 and earlier.

After .NET 4.0, concurrent garbage collection was replaced with Background Garbage Collection. Both concurrent and background GC applies only to generation 2 collections.

Read more on what happens during garbage collection

We can use Induced Garbage Collection at points in our code where we have recently stopped using a large number of objects. This is useful in scenarios where the programmer may know that a certain path of the program results in several objects no longer being needed. Instead of depending on the GC to figure this out on its own, we can just call `GC.Collect()` to trigger collection at this time.

The GC class contains documentation on `GC.Collect()` and all other methods of the `GC` class.

MSBuild

MSBuild for .NET 6.0 and later uses Implicit Using Directives for different project types. You can optionally disable implicit using directives, but I will likely not disable these as I feel I should probably get used to default .NET settings for now.

To specify the version of C#, use the LangVersion Property within your `.csproj` file. You can check your current version by writing `#error version` in your program and running it to check the output of the produced error.

Type Categories

Microsoft - Reference Types

Microsoft - Value Types

Microsoft - Pointer Types

Collections

I'm coming to .NET from C++, so it will help me to go through the System.Collections.Generic documentation and find the containers that closely match those which I use in C++.

The obvious -

LinkedList<T> is equivalent to `std::list` as a doubly-linked list in C++

Queue<T> is equivalent to `std::queue` in C++

Stack<T> is equivalent to `std::stack` in C++

The not-so obvious -

List<T> is equivalent to `std::vector` in C++

SortedSet<T>is equivalent to `std::set` in C++

HashSet<T> is equivalent to `std::unordered_set` in C++

SortedDictionary<TKey, TValue> is equivalent to `std::map`, as it is sorted by keys with O(log N) insertion time and retrieval.

Dictionary<TKey, TValue> is equivalent to `std::unordered_map` and provides O(1) retrieval as it is implemented using a hash table.

Collections with no C++ equivalent, or not similar enough to be compared to C++ containers -

SortedList<TKey, TValue> is similar to `SortedDictionary<TKey, TValue>`, but uses less memory and has O(n) insertion time with O(log n) retrieval. If a `SortedList` is constructed from pre-sorted data, it is faster than `SortedDictionary`.

For collections that use `<TKey, TValue>`, we can anticipate the enumerator to provide each element as a KeyValuePair<TKey, TValue>. For example, we can iterate over each element in a `Dictionary` using the following `foreach` loop

```
foreach( KeyValuePair<string, string> kvp in myDictionary )
{
    Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
}
```

PriorityQueue<TElement, TPriority>

Array is the type applied to arrays created using `[]`. For example, the following variables `a`, `b`, and `c` are all of the same `Array` type, but each were created using a different approach

```
int[] a = { 1, 2, 3};
Array b = new int[3];
Array c = Array.CreateInstance(typeof(int), 3);
Console.Write("\na.GetType: {0}", a.GetType());
Console.Write("\nb.GetType: {0}", b.GetType());
Console.Write("\nc.GetType: {0}", c.GetType());
```

The output of this code is

```
a.GetType: System.Int32[]
b.GetType: System.Int32[]
c.GetType: System.Int32[]
```

These are all arrays of the `Int32` type, but you could create arrays of custom class objects, or other builtin types.

For help on selecting the correct collection, see Collections and Data Structures, where collections and their operation complexity are compared.

Concurrency

ConcurrentQueue<T>

ConcurrentStack<T>

ConcurrentBag<T>

ConcurrentDictionary<TKey, TValue>

Input / Output

```
void TestInput()
{
  string formattingString = "Captured {0} input: {1}\n";

  Console.Write("\nInput a character, then press enter: ");
```

```
  int ascii = Console.Read();
  char ch = Convert.ToChar(ascii);
  Console.Write(formattingString, "character", ch);
  Console.ReadLine(); // Discard any left over input

  Console.Write("\nPress a key: ");
  ConsoleKeyInfo key = Console.ReadKey();
  Console.Write("\n" + formattingString, "key", key.KeyChar);

  Console.Write("\nEnter a line: ");
  string? line = Console.ReadLine();
  Console.Write(formattingString, "line", line);
}
```

TODO: Read from file

String Composite Formatting

String Composite Formatting takes a list of objects that follow the initial formatting string. We use `{0}` to select the first object, `{1}` to select the second, and so on. We can reuse `{0}` or any other index as many times as we like within the formatting string. We can also use Format String Components such as `{0:F6}`. This example outputs a float to the 6th decimal place.

```
string fmt = "This is pi: {0}\nThis is the date: {1}\nThis is also pi: {0:F6}";
Console.WriteLine(fmt, Math.PI, DateTime.Now);
```

The output of this code is

```
This is pi: 3.141592653589793
This is the date: 5/1/2022 6:04:20 PM
This is also pi: 3.141593
```

String Interpolation

String interpolation is similar to f-strings in Python. By leading a string with the `$` character, we define an interpolated string in C#. If we want to include `{` or `}` in our output, we need to escape them by doubling the brackets with `{{` and `}}` respectively.

```
string a = "This is my string!";
// Right-align using `, 30` or any positive integer to represent; Negative integers are for left-align
Console.WriteLine($"This is my rifle; {a, 30}");
```

```
Console.WriteLine($"This is {{my}} rifle; {a}");
var b = $"This {{is}} my rifle; {a}";
Console.WriteLine(b);
```

The output of this code is

```
This is my rifle;           This is my string!
This is {my} rifle; This is my string!
This {is} my rifle; This is my string!
```

There is also conditional formatting and the Format String Component

```
var b = $"This {{is}} my rifle; {a}";
// Conditional formatting must be wrapped in ( and )
Console.WriteLine($"Conditional formatting result: {(b.Length == 0 ? "Empty" : "Not empty")}");
var pi = Math.PI;
// Formatting string components
Console.WriteLine($"{pi:F3}, {pi:F10}, {DateTime.Now:d}, {DateTime.Now:f},
{DateTime.Now.ToLocalTime():h:mm:ss tt zz}");
```

The output of this code is

```
Conditional formatting result: Not empty
3.142, 3.1415926536, 5/1/2022, Sunday, May 1, 2022 5:55 PM, 5:55:56 PM -04
```

It's worth mentioning you can do some neat stuff in C# 11 or later, using .NET 7.0; Unfortunately I don't have access to these things or rather I don't want to set them up at the moment - I'm just learning. See the bottom-half of this section for more info

String Verbatim

C# - Verbatim (@)

Not to be confused with a string literal. A string literal is simply a string that is defined literally within the code of an application. The following is an example of a string literal

```
string lit = "This is a literal\nWe are now on a new line.";
Console.WriteLine(lit);
```

The output of this code is

```
This is a literal
We are now on a new line.
```

Verbatim string literals are useful when formatting long strings or strings with several `\` characters within them. To declare a verbatim string literal, prepend a `@` character to the opening double-quotes of your string literal

```
var lit = @"
hi
    how
""are"" you? \this\is\a\literal
";
Console.WriteLine(lit);
```

The output of this code is exactly as we defined the `lit` string above, aside from the appearances of `""` being replaced with a single double-quote `"`

```
hi
    how
"are" you? \this\is\a\literal
```

The verbatim `@` symbol can also be used to allow us to define functions or variables with otherwise reserved names in C#.

```
// Without @ we wouldn't be able to declare a variable named `foreach`
string[] @foreach = {@"\this\is\new\a\test\n", "Not verbatim\nBut still literal"};
foreach (string s in @foreach)
{
  Console.WriteLine(@s);
}
```

The output of this code is

```
\this\is\new\a\test\n
Not verbatim
But still literal
```

Lamdas

Lamdas in C# were a bit odd to read at first, but once I understood the types behind them and what these types meant, things started to make more sense.

```
// Both of these lambdas are of the same type; Func<string, int> where int is the value returned
var getLen = (string s) => s.Length;
Func<string, int> funcLen = (string s) => s.Length;
Console.WriteLine("Length: {0}", getLen("Hello").ToString());
Console.WriteLine("Length: {0}", funcLen("Hello").ToString());

var isEqual = (string a, string b) => a == b;
Console.WriteLine(isEqual("Test", "Test"));
Func<string, string, bool> funcIsEqual = (string a, string b) => a == b;
Console.WriteLine(funcIsEqual("Test", "Test"));

// These two lamdas are both of type Action<string>, as they do not return a result
var statement = (string s) =>
{
  var arr = s.ToCharArray();
  Array.Reverse(arr);
  Console.WriteLine($"\"{s}\" reversed: {new string(arr)}");
};
Action<string> actionReverse = (string s) =>
{
  var arr = s.ToCharArray();
  Array.Reverse(arr);
  Console.WriteLine($"\"{s}\" reversed: {new string(arr)}");
};

// This lamda is a Func<string, string> as it take a string parameter and returns a string as a result
Func<string, string> revString = (string s) =>
{
  var revArr = s.ToCharArray();
  Array.Reverse(revArr);
  return new string(revArr);
};
string testS = "Racecar";
Console.WriteLine($"{testS} reversed: {revString(testS)}");
```

The output of this code is

```
Length: 5
Length: 5
True
```

```
True
Racecar reversed: racecaR
"Test" reversed: tseT
```

Class

C# Polymorphism has several examples of using `override`, `virtual`, `sealed`, and Inheritance outlines some basic rules and limitations. See C# Properties for more information on properties, and their getters / setters.

See C# Specification - Classes for a detailed outline of different usecases for classes with examples.

C# - Class (reference types)

Unless otherwise specified, the Access Modifier for classes is `internal`, and all class members are `private`. This means a class can by default only be accessed within the code assembly where it was declared (AKA the same compilation), and the class members can by default only be accessed from within the class itself.

A class is a Reference Type, which means they are allocated on the heap and assignment actually creates a reference to this data on the heap, rather than a copy. For example

```
var shape = new Square(); // Heap allocation
shape.Print();
var shapeRef = shape; // Reference to heap data
shapeRef.Height = 20;
shape.Print();
```

This code produces the following output, showing that the `shapeRef` variable is actually modifying the same data within `shape` itself.

```
Printing Square info...
Square WxH is 10x10 at position (0,0)
Printing Square info...
Square WxH is 20x20 at position (0,0)
```

Classes only support single inheritance, which means a class may inherit from a single base class and extend or define functionality. Classes may inherit from multiple *interfaces*, but may only inherit from a single base class. These are not mutually exclusive, so the following class is valid. Here, we declare an `abstract` base class `Animal` and inherit from it to create the `Dog` and `Human` classes. More information on the abstract modifier

```
// By declaring Animal as abstract, we cannot construct an Animal; We must inherit from it and define Speak()
public abstract class Animal
{
  public Animal(string n, string p)
  {
    this.Name = n;
    this.Phrase = p;
  }

  // All abstract methods or members will need to be defined to inherit from an abstract base class
  public abstract void Speak();

  private string name;
  public string Name { get; set; }
  private string phrase;
  public string Phrase { get; set; }
}

// Inherit from the Animal abstract base class
public class Human : Animal
{
  public Human(string n, string p) : base(n, p) { }

  public override void Speak()
  {
    Console.WriteLine("{0} (Human): {1}", Name, Phrase);
  }
}

// Inherit from Human, and N Interfaces
public class Teacher : Human, IComparable, ICloneable
{
  public Teacher(string n, string p) : base(n, p) { }

  public int CompareTo(object? obj)
  {
    throw new NotImplementedException();
  }

  public object Clone()
```

```
  {
    throw new NotImplementedException();
  }
}
```

We can use another example of a `Square` class to show more features of C# classes and polymorphism. Here, we define a few `virtual` members that we will later use to show some interesting uses of class polymorphism. More information on virtual keyword

```
class Shape
{
  public Shape(int y, int x)
  {
    Y = y;
    X = x;
    Name = this.ToString();
  }

  // Default ctor will be used for inheriting classes
  public Shape()
  {
    X = 0;
    Y = 0;
    Name = this.ToString();
  }

  // Shape position; Private setter, public getter
  public int X { get; private set; }
  public int Y { get; private set; }
  public string Name { get; private set; }

  // Auto-implemented properties may use a default value initializer
  public virtual int Width { get; set; } = 1;
  public virtual int Height { get; set; } = 2;

  public virtual void Print()
  {
    Console.WriteLine($"{Name} WxH is {Width}x{Height} at position ({X},{Y})");
  }
```

```
}
```

What if we want to make a `Square` shape class? We can inherit from `Shape` and override the `Width` and `Height` properties to enforce the requirements of a square; Width must always be equal to height. To do this, we create two new encapsulated values and `override` the `Height` and `Width` properties. More information on override modifier

```
class Square : Shape
{
  // Set default value on encapsulated value for non auto-implemented properties
  private int width = 10;
  private int height = 10;

  // We can override properties just as we can functions
  // + Height and Width properties can no longer set default values
  public override int Height
  {
    // Can use expressions for getters / setters
    get => height;
    set => width = height = value;
  }

  public override int Width
  {
    get => width;
    set
    {
      // Same setter as Height, just within a block of statements
      width = value;
      height = value;
    }
  }

  public override void Print()
  {
    Console.WriteLine("Printing Square info...");
    base.Print(); // Will now use Square's getter / setter to print private int width, height
  }
}
```

Now what if we want to create a `Cube` class? We will need to handle `Print()` differently, and the call to `base.Print()` will no longer be useful to us because we don't need that output. Instead, we want to print `WxHxD` where we have added the `Depth` to represent the third-dimension of our cube. Notice the `new` keyword specified for `Print()`, which indicates it is an entirely new implementation and not an override. More information on the new modifier

```
class Cube : Square
{
  // Add new properties or encapsulated values as needed
  private int depth = 10;
  public int Depth
  {
    get => depth;
    set => depth = value;
  }
  // Implement a `new` Print() function which acts as a new stand-alone implementaton
  public new void Print()
  {
    Console.WriteLine("Printing Cube info...");
    Console.WriteLine($"{Name} WxHxD is {Width}x{Height}x{Depth} at position ({X},{Y})");
  }
}
```

As a final example, we look at the `sealed` keyword, which marks itself as the final implementation for a previously `virtual` member. This means future classes that inherit will no longer be able to override this member, which could be useful in some cases. More information on the sealed modifier

```
class Rectangle : Shape
{
  // Classes that inherit from Rectangle can not override Print
  public sealed override void Print()
  {
    Console.WriteLine("Printing sealed Rectangle info...");
    base.Print();
  }
}

class Rect : Rectangle
```

```
{
  // Rect can't override Print(), since its base class declared it as `sealed`
}
```

With all that said, we can use our above shape classes with the following code

```
// Test our custom ctor for Shape
var shape = new Shape(5, 5);
shape.Print();

// Test default ctor for Shape
shape = new Shape();
shape.Print();

// Test square, which no longer has access to a custom ctor but can still be default constructed
shape = new Square();
shape.Print();
shape.Height = 8; // Modify the height of the square, which also sets width to 8
shape.Print();

// Create a cube to test our new Print implementation
// + We can't reuse shape here; We need to declare a new object of type Cube
Cube cube = new Cube();
cube.Print();

// Create a rectangle to show sealed overrides
shape = new Rectangle();
shape.Print();

// Create a rect to show the use of our inherited sealed memebr
shape = new Rect();
shape.Print();
```

Which gives us this output

```
Shape WxH is 1x2 at position (5,5)
Shape WxH is 1x2 at position (0,0)
Printing Square info...
Square WxH is 10x10 at position (0,0)
Printing Square info...
```

```
Square WxH is 8x8 at position (0,0)
Printing Cube info...
Cube WxHxD is 10x10x10 at position (0,0)
Printing sealed Rectangle info...
Rectangle WxH is 1x2 at position (0,0)
Printing sealed Rectangle info...
Rect WxH is 1x2 at position (0,0)
```

Note that because of polymorphism, we can produce the same output as above, using the following code

```
var box = new List<Shape>();
box.Add(new Shape(5, 5));
box.Add(new Square());
box.Last().Width = 8; // Access the last element we added to the List, set its width to 8
box.Add(new Cube());
box.Add(new Rectangle());
box.Add(new Rect());
// Use a lambda to find a Cube, get a reference to it; If we found a Cube, set its depth to 5
if (box.Find((Shape s) => s.GetType() == typeof(Cube)) is Cube cubeRef) cubeRef.Depth = 5;
foreach (var s in box) s.Print(); // Print all the Shapes
```

The above example shows use of the keyword `is`, which tests whether casting is successful, and then allows us to create the `cubeRef` local variable for use as a `Cube`.

We could also use `as` to perform a similar test, where instead of throwing an exception the `as` expression would return `null`. An example of this is below.

```
var cub = new Cube();
var sqr = cub as Square;
if (sqr != null) sqr.Print();
```

Which has the following output - note that the reason `Cube` appears in the last line is because the cube constructor initializes this member to be the name of the initial type constructed. Since we declared `cub` as a `Cube`, `cub.Name` is Cube! Upcasting to a `Square` does not change this value. `Square` appears on the first line of the output because this line is a literal string within the `Square` class, and does not reference the `Name` property.

```
Printing Square info...
Cube WxH is 10x10 at position (0,0)
```

Inherited classes are initialized in a bottom-up fashion, where the initialization of class fields occurs first, then the constructor is called, and the process repeats for the classes we have inheritied from. To test this, I wrote the following example code

```
class A
{
  public A()
  {
    Console.WriteLine("A default constructor was called");
    Val = 0;
  }
  public A(int v)
  {
    Console.WriteLine("A parameterized constructor was called");
    Val = v;
  }
  private int val;
  public int Val
  {
    // No setter; We can only initialize on construction
    init
    {
      Console.WriteLine($"A.val was initialized: {val}");
      val = value;
    }
  }
}

class B : A
{
  public B()
  {
    Console.WriteLine("B default constructor was called");
    BVal = 0;
  }
  public B(int bv)
  {
    Console.WriteLine("B parameterized constructor was called");
    BVal = bv;
  }
```

```
  private int bVal;

  public int BVal
  {
    init
    {
      bVal = value;
      Console.WriteLine($"B.bVal was initialized: {bVal}");
    }
  }
}

class C : B
{
  public C()
  {
    Console.WriteLine("C default constructor was called");
    CVal = 0;
  }
  public C(int cv)
  {
    Console.WriteLine("C parameterized constructor was called");
    CVal = cv;
  }

  private int cVal;

  public int CVal
  {
    init
    {
      cVal = value;
      Console.WriteLine($"C.cVal was initialized: {cVal}");
    }
  }
}
```

And then constructed a `C` object

```
Console.WriteLine("\nTesting construction order...");
// What will be the value of cVal? Our constructor parameter, or the initialization value, CVal = 10?
var aClass = new C(5) {CVal = 10};
```

Which produced the following output, which was *not* what I had expected before writing this example code.

```
Testing construction order...
A default constructor was called
A.val was initialized: 0
B default constructor was called
B.bVal was initialized: 0
C parameterized constructor was called
C.cVal was initialized: 5
C.cVal was initialized: 10
```

Operator Overloading

Object

The object base class can be used for boxing and unboxing of any type, which makes the following code completely valid. An `object` is not to be confused with the concept of Objects, which are intantiations of classes.

```
var container = new List<object>();
container.Add(new object());
container.Add(new Shape());
container.Add(new int());
container.Add(new C());
container.Add(new string(""));
```

What's really happening here is each time we call `Add` to our `container`, we are implicitly boxing the values with the `object` class. So we place a `Shape` within the container of objects, but before the shape is added to the container a new `object` is constructed, which holds our `Shape`. This is expensive, but it can be applied to all types in C#. To program for different types more efficiently, we should take advantage of covariance in polymorphism, or we could use Generics. With generics, we must provide type parameters just as we do within `List<T>`, but once we provide this type we must only use this type or its subclasses within the container.

We can box and unbox objects using the following approach. All of these examples are valid.

```
int x = 9;
object obj = x; // Box the int
int y = (int)obj; // Unbox the int
object obj = 9;
long x = (int) obj;
object obj = 3.5; // 3.5 inferred to be type double
int x = (int) (double) obj; // x is now 3
```

Casting

The following example shows the use of `as` for testing if a cast succeeds. The `as` operator is comparable to `dynamic_cast` in C++, and should be checked for `null` if there is any chance of the cast failing. If the cast will never fail, consider using the `is` operator to cast one variable to a new variable within an `if` statement, which limits the scope to that block of code.

To use the `as` or `is` keywords, we must be referring to a base type of an inherited class. For example, we can check `asset as House`, but we cannot check `stock as House`.

```
public class Asset
{
  protected int _value = 10;
  public int Value { get => _value; }
}

public class Stock : Asset
{
  protected int _count = 5;
  public int Count { get => _count; }
  public void PrintStock()
  {
    Console.WriteLine($"We own {Count} stocks valued at {Value};" +
                      $" Total value: {Count*Value}");
  }
}

public class House : Asset
{
  private int _sqFt = 100;

  public int SquareFt
  {
```

```
    get => _sqFt;
  }

  public void PrintHouse()
  {
    Console.WriteLine($"We own a house valued at {Value * SquareFt}");
  }
}

var stock = new Stock();
// Upcast Stock to an Asset
var asset = stock as Asset;
// asset variable does not have access to PrintStock; We can only see Value
Console.WriteLine($"Asset value is {asset.Value}");

// Downcast an Asset to a Stock; If we succeed, call PrintStock()
// + assetStock is declared as the downcasted variable result
if (asset is Stock assetStock)
{
  assetStock.PrintStock();
}

// We use `as` here, and check for null
var notHouse = asset as House;
if (notHouse != null)
{
  Console.WriteLine("Asset is a house! This case will not happen.");
  notHouse.PrintHouse();
}
else
{
  // notHouse is not available within this scope
  Console.WriteLine($"Variable `asset` is not an asset; Type: {asset.GetType().Name}");
}
```

The output of this code is seen blow. Note that the reason `asset.GetType().Name` returns `Stock` as our type is because we instantiated the `asset` variable using an upcast from a `Stock` to an `Asset`

```
Asset value is 10
We own 5 stocks valued at 10; Total value: 50
```

```
Variable `asset` is not an asset; Type: Stock
```

We can also perform explicit casts with the following format

```
var stock = new Stock();
var asset = (Asset) stock; // Upcast
var downCast = (Stock)asset; // Downcast
downCast.PrintStock();
```

Interfaces

Microsoft - Interface

Some useful interfaces to consider implementing -

- IDisposable
- IEnumerable
- IEnumerable<T>
- IEnumerator
- IEquatable<T>
- ICloneable
- IComparable<T>
- ICollection
- ICollection<T>
- IList
- IList<T>
- IDictionary
- IDictionary<T>
- IFormatProvider
- IFormatter
- ISerializable
- IQueryable
- IQueryProvider
- Collection<T>

```
void PrintEnum(IEnumerable<int> obj)
{
```

```
  Console.WriteLine();
  foreach (var i in obj)
  {
    Console.Write("{0}, ", i);
  }
}
```

Structs

Supports multiple inheritance, where N interfaces can inherit from each other to create a single interface. Structs do not support inheriting from other structs or classes. Structs can not be declared as `abstract` or `sealed`, because stuct types are never abstract and implicitly sealed. Member variables (fields) of a struct may not include initializers unless declared `static`. Fields of a struct that are of reference type are automatically initialized to `null` when constructed. Value Semantics is important to understand and is a key difference between a `class` and a `struct`. A struct can not have a parameterless constructor, field initializers, a finalizer, or vir- tual / protected members.

Since a struct is a value type, we can save ourselves many heap allocations by using a struct in place of a class when we want to instantiate a large number of them. For example, a `List<T>` where T is a struct only requires the single heap allocation for the `List<struct>` itself - Each item in our list will be a value type that is placed on the stack, and thus we save ourselves from allocating each list item on the heap. Usually a struct makes more sense when we are using integer types or other values where assignment should produce a copy, instead of a reference.

C# Specification - Structs

C# Specification - Differences Between Class and Struct

C# - Structs (value types)

Below we create a class `Bag` that holds `Item` structs

```
// Inherit from IEquatable interface so we can check our Bag for a certain Item :D
public struct Item : IEquatable<Item>
{
  public Item(double v, int q, string name)
  {
    Value = v;
    Qty = q;
    Name = name;
  }
```

```
  public Item(double v, string name)
  {
    Value = v;
    Qty = 1;
    Name = name;
  }

  public string Name { get; set; }
  public double Value { get; set; }
  public int Qty { get; set; }

  public static bool operator ==(Item a, Item b)
  {
    if (((object)a) == null || ((object)b) == null) return Object.Equals(a, b);
    return a.Equals(b);
  }

  public static bool operator !=(Item a, Item b)
  {
    return !(a == b);
  }

  public bool Equals(Item other)
  {
    return Name == other.Name && Value == other.Value;
  }

  public override bool Equals(object? obj)
  {
    return obj is Item other && Equals(other);
  }

  public override int GetHashCode()
  {
    return HashCode.Combine(Name, Value);
  }
}

class Bag
```

```
{
  public Bag()
  {
    contents = new List<Item>();
    maxCarry = 10;
  }
  private List<Item> contents;
  private int maxCarry;

  public bool AddItem(Item i)
  {
    if (contents.Count >= maxCarry) return false;
    contents.Add(i);
    return true;
  }

  public Item? TakeItem(Item i)
  {
    var found = contents.Find((Item inBag) => inBag == i);
    if (found == default(Item)) return null;
    contents.Remove(found);
    return found;
  }
}
```

And we can use our `Bag` class and `Item` struct, where a bag is a reference type and an item is a value type.

```
var bag = new Bag();
Item wrench = new Item(1.5, 1, "wrench"); // Create a wrench using Item ctor
var spanner = wrench; // Copy wrench to a new item
spanner.Name = "spanner";
spanner.Value = 5.0;
spanner.Qty = 2;
Item socket = new Item(2.5, 5, "socket"); // Create a new item using ctor
var bagRef = bag; // Reference to the same Bag
bag.AddItem(wrench);
bag.AddItem(spanner);
bag.AddItem(socket);
Item? bagSpanner = bagRef.TakeItem(spanner); // Take an item from bag / bagRef
```

```
Console.WriteLine(Object.ReferenceEquals(bagSpanner, spanner)); // Proof that spanner and bagSpanner are each their own instance of item
Item? noSpanner = bag.TakeItem(spanner); // Take the same item from bag / bagRef; Returns null because we no longer have a spanner :(
Console.WriteLine(noSpanner == null);
```

The output of this code is

```
False
True
```

Records

C# - Records (reference types)

Generics

Generics are supported by `class`, `struct`, `interface`, and `delegate` types. For basic examples see C# Type System - Generics. Generics are used to implement `System.Collections.Generic` much like templates are used to implement the Standard Template Library in C++.

This does not imply that Generics and Templates are the same, as there are a few key differences between the two.

TODO: Differences from C++ templates

Generics can be applied to a class, or a single method of a non-generic class. The appearance of type parameters (`<T>`) indicates the method or class is generic

```
public class Generic<T>
{
    public T Field;
}

// Non-generic class A with generic method G<T>
class A
{
    T G<T>(T arg)
    {
        T temp = arg;
        //...
        return temp;
```

```
    }
}
```

Covariance, Contravariance, and Invariance

Unmanaged Memory

Cleaning up unmanaged resources

IDisposable

Marshall - For allocating unmanaged memory

The following code shows a quick example of how destructors are used in C#. Destructors are also referred to as Finalizers

```
// Test class to show destructor use by the garbage collector
public class Garbage
{
  private static int count = 1;
  ~Garbage()
  {
    Console.WriteLine($"Dtor called: {count++}");
  }
}

// Produce 9 pieces of garbage
foreach (int i in Enumerable.Range(1, 10))
{
  var trash = new Garbage();
}
GC.Collect();
```

The output of this program is seen below. Note that if we had not called `GC.Collect()`, we would have left it up to the garbage collector to perform collection on the `trash` we created in the loop. We explicitly call this to

```
Dtor called: 1
Dtor called: 2
Dtor called: 3
Dtor called: 4
```

```
Dtor called: 5
Dtor called: 6
Dtor called: 7
Dtor called: 8
Dtor called: 9
```

To take this example further, we create a new `Reader` class that has some managed and unamanged resources. The destructor (finalizer) is used to ensure unmanaged resources are never left undisposed, while the `Dispose` methods are used to dispose of managed resources.

```
public Reader()
{
 _client = new HttpClient();
 _stream = File.OpenRead("/home/kapper/test.txt");
 _unmanagedString = (IntPtr) Marshal.StringToHGlobalAnsi("This is our unmanaged string");
}
public void PrintString()
{
 Console.WriteLine(Marshal.PtrToStringAnsi(_unmanagedString));
}
public void Dispose()
{
 Dispose(true);
 GC.SuppressFinalize(this);
}
public void Dispose(bool disposing)
{
 // Prevent from disposing multiple times
 if (!_isDisposed)
 {
  if (disposing)
  {
   // Cleaning up managed resources
   Console.WriteLine($"Disposing: {_count++}");
   _client.Dispose();
   _stream.Dispose();
  }
  // Clean up unmanaged resources here
  Marshal.FreeHGlobal(_unmanagedString);
  _isDisposed = true;
```

```
    }
  }

  ~Reader()
  {
    Console.WriteLine($"Calling destructor {_count}");
    // Ensure we clean up unmanaged resources; Do not dispose managed resources
    // + This is for the case where we forget to call dispose
    Dispose(false);
  }
}



foreach (int i in Enumerable.Range(1, 5))
{
  // Create `trash` with using; Automatically calls Dispose when we leave scope
  using var trash = new Reader();
  trash.PrintString();
}
// Nothing to collect; We disposed of our `trash` when we left the scope of using
// GC.Collect();
```

The output of this code is seen below -

```
This is our unmanaged string
Disposing: 0
This is our unmanaged string
Disposing: 1
This is our unmanaged string
Disposing: 2
This is our unmanaged string
Disposing: 3
This is our unmanaged string
Disposing: 4
```

More on finalization / disposal

Nullable

.NET Nullable<T> supports nullable types for languages within .NET, but the use of `Nullable<T>` isn't needed for C# and Visual Basic as these languages both have syntax for nullable types built-in.

```
dynamic? a = null;
var b = a ?? "A is null";
a ??= "A is null; ??= will apply this assignment";
b ??= "This value will not be changed; B is not null";
Console.WriteLine(a);
Console.WriteLine(b);

char? c = null;
// Null conditional operator returns null if lhs of ?. is null
string d = c?.ToString();
// ? operator can be applied to [] operators to prevent out of bounds exceptions
Console.WriteLine(d?[2] == null ? "Is null" : "Not null");
```

The output of this code is

```
A is null; ??= will apply this assignment
A is null
Is null
```

C# Nullable provides more examples of nullable types in C#.

async / await

LINQ

Language-Integrated Query (LINQ) Overview

C# LINQ API Documentation provides information on various LINQ classes and methods, but a notable mention is Enumerable, which represents any collection of objects that can be enumerated on.

For example, we can use Where and Aggregate to conditionally select values using `Where` within a collection, and then perform some custom function to transform the selected elements via `Aggregate`.

The following function selects all values within the `nums` array and multiplies them together - notice the use of `i` in the for loop to prevent us from including the current index in the final

product. The `_` symbol is used to discard the `TSource` (`int`, in this case, since our array type source is Int32) value of the predicate `Func<TSource,Int32,Boolean>`. In this example, we pass a lambda to `Where` that is defined as `(_, index) => index != i` - this expression simply returns an enumerable from our `nums` array, where we skip the value at index `i`.

```
public int[] ProductExceptSelf(int[] nums) {
  int[] result = new int[nums.Length];
  for (int i = 0; i < nums.Length; i++) {
    result[i] = nums.Where((_, index) => index != i).Aggregate((a, b) => a * b);
  }
  return result;
}
```

We can also format LINQ expressions as shown below, where we extract all characters from a string and convert them to lowercase. Notice that we only select characters that are within the alphabet.

```
public class Solution {
// Input: s = "A man, a plan, a canal: Panama"
  public bool IsPalindrome(string s) {
    // Select each char from s that is a letter or digit; Convert to lower case
    IEnumerable<char> query =
      from c in s
      where Char.IsLetterOrDigit(c)
      select Char.ToLower(c);
    // arr: "amanaplanacanalpanama"
    var arr = query.ToArray();
    Array.Reverse(arr); // Copy query array; Reverse it
    return new string(query.ToArray()) == new string(arr); // Use string comparison to test palindrome
  }
}
```

Which is the same as

```
public class Solution {
  public bool IsPalindrome(string s) {
    // Select each character that is a letter or digit; Convert to lower case
    var query = s.Where(c => Char.IsLetterOrDigit(c)).Select(c => Char.ToLower(c));
    var arr = query.ToArray(); // Copy array returned from query
    Array.Reverse(arr); // Reverse arr to test palindrome
// Using string compare against query result and reversed result
```

```
    return new string(query.ToArray()) == new string(arr);
  }
}
```

Both of these functions produce the same results. This is just one example of using LINQ expressions vs instance method LINQ syntax.

RPC / WCF

Microsoft - RPC Types

Cool Code

nameof can be used to obtain the name of a variable, type, or method.

Revision #14
Created 30 April 2022 14:24:19 by Shaun Reed
Updated 10 May 2022 03:44:48 by Shaun Reed