

# M.T White

## C# and Programming

The stuff you either forgot,  
didn't understand or are  
currently forgetting!



# CONTENTS

[Chapter 1: Classes, Objects, Pillars, and Relationships](#)

[Chapter 2: Methods revised: The hard stuff](#)

[Chapter 3: Classes part 2 and other weird stuff](#)

[Chapter 4: Typing](#)

[Chapter 5: Arrays](#)

[Chapter 6: Collections](#)

[Chapter 7: Generics the really cool stuff](#)

## About this book.

So, you're fresh out of college, about to graduate college, learning to program, or looking to make a career change. Maybe you've been on a few interviews, maybe you have a job offer in hand. No matter what, if you're reading this book it is a clear indication that you want to either brush up on the stuff you learned in college or you want to enhance your skills as C# programmers. Either way you've come to the right place.

Over the years I have been on many interviews and in the preparation for the interview and on the interview itself I have learned a lot. I love interviewing because from a knowledge perspective it lets me know where I'm weak. Not only that, I've found that most of the places that I was weak other developers are weak as well. So, one day I decided to sit down and compile a list of common interview questions, from the better organizations I've interviewed with, and study them along with the underlining material I needed to understand the concepts. After spending a few months doing this I discovered two things. The first, the farther I delved and the more I learned the better the job offers I got. The second was I was becoming a much better developer. I also found that many programmers, especially you newbies and fresh grads are obsessed with syntax and algorithms and that many concepts that would otherwise make you look like a rock star to a potential employer are lost in the mix.

This where this book comes in. My goal with this book is to shed light on topics and hopefully fill in some gaps that are keeping you from being a decent programmer. I'm hoping to take the tricks I've learned and pass them on to you. Not only that I'm going to try to give context to these concepts and cover topics that are often brought up in interviews. To use this book properly don't worry about memorizing syntax or patterns, try to understand the how and why behind what's being covered. This book should be used more than a textbook to memorize potential interview questions. This book should be used as a means of understanding the little tid bits of information that you either don't know, are learning, or just fell through the cracks over the years.

This book, unlike most, will try to take a lighthearted approach to learn programming. There are way too many books, and programmers, that take themselves way too seriously. If you read an average programming

book, chances are if you have any sanity it's going to be a complete yawn fest. So, being the rebel programmer, I am I decided to try to sprinkle in some humor into this to make it a little more interesting. Hopefully, as you read through this book, you'll have fun and learn a thing or two.

## **What You Need to Get Started**

To use this book all you need is a computer that is capable of running Visual Studios. As I'm sure you've figured with the last statement, you'll need to download Visual Studios as well. Visual Studios Community is free to download and can be downloaded from the Visual Studio website. Visual Studios is a large program and it may take some time to get it installed and moving.

## **Who is this book for?**

The material covered in this book may seem rudimentary at first, and to some extent it is. The topics covered are mostly from interview questions that I was asked when I was starting out or weak spots that I've noticed in new programmers. The book is best used along with a programming 1 or 2 course, by a fresh grad that needs help filling in the gaps for concepts that weren't fully grasped in college, or a person that needs a review before an interview. In short, this book is a general-purpose book that is designed as a guide for the more advanced concepts in C# and programming in general.

## **Code examples**

For this book, I'm going to try to keep code examples as consistent as possible. I'm going to do my best to add and not remove any code in the examples to demonstrate how the new code and old code do not clash. The examples will not necessarily build on top of each other to demonstrate the concepts portrayed. I'm doing this so you, the reader, can experiment, observe and explore. However, there are times when code will be changed and removed to keep things easy to understand. The Main method will be altered drastically from example to example to demonstrate the concept that we are going over. I'm hoping this will make the book easier to understand.

## **Feedback**

This is my first book. So, if you have any constructive criticism please leave a comment. Did I miss a topic? Did I do a poor job at explaining something? Please let me know. I also recommend checking for updates every so often. I'm always polishing my work. I'm poor and can't afford a fancy proofreader yet.

# CHAPTER 1: CLASSES, OBJECTS, PILLARS, AND RELATIONSHIPS

To start off I want to start with a little story. This little tale revolves around the time I interviewed with a NASA contractor at the Johnson Space Center. Long story short, I was interviewing with several engineers that were responsible for making a very important component of the Orion spacecraft. The interview was your standard B.S when one of the engineers said something that raised a very red flag. We were talking about OOP when he said, “to us, OOP is just programming with classes.” At that point my mind almost caved in, I saw fairies and leprechauns dancing on the tables, and out of nowhere I saw the universe begin to break apart. Wow, what a statement from the develops helping man reach the stars. This is a true story; I kid you not. I actually met these people that are responsible for writing spaceflight software. At this point you might be thinking to yourself, well OOP is programming with class, and yes, the engineer was right but is OOP just programming with classes? The answer is no. For most fresh out of college graduates, OOP is programming with classes, and that is a very simplistic and sloppy way of thinking about OOP.

So, if OOP isn’t just coding with classes what is it? Well, OOP is a way of organizing and reusing code. What in the world does that mean? Well, let’s start by looking at code organization and how that relates to classes and objects.

## **Methods: The easy stuff**

When exploring the magical world of OOP we first need to look at methods. There’s a lot to methods so for now we’re going to keep it simple and circle back around to the harder stuff in a later chapter. So, to understand methods you must first drill in this saying, “a programmer does not repeat his

or herself”. As such, if you’re reading this, I’m assuming you’re a programmer of some kind, so when you write code that does whatever it does you should never write it again. If you end up writing a block of code that does the same thing more than once quit your job immediately and go become a software tester! With that being said, how does one avoid the horrid life of a tester and write blocks of code that don’t have to be repeated? The answer to that is to write methods.

What is a method? A method is a named, callable block of code that can accept arguments and return a value. If you’re an old school programmer a method may sound a lot like a function, and indeed it should because in C# methods are functions. Technically speaking there is a slight difference between a method and a function. Technically, a function is a method when it is declared in a class or struct. In other words, if it’s a function and it is in a class it’s a method, if a function is not in a class it’s not a method. C# does not allow functions to be declared outside of a class or struct, and as such C# does not technically support functions it only supports methods. Now, with all that being said functions versus methods in everyday language is just semantics and the terms are oftentimes used interchangeably. This splitting of hairs is just so you know the difference when you’re arguing with pompous programmers on interviews.

With that being said, to have a method you need a class or struct, and every program must have one method named Main just like the following,

```
using System;

namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The Main method is a required method as it is the entry point for the program. In other words, when a C# program starts up the Main method is

automatically called and the methods that make your application awesome are called from there. So, in short, no Main method no program.

For any program that does more than says, “Hello World” you’re going to need more than one method. So, how do you declare custom methods? The pattern for declaring a method is very simple, and the following is the basic template,

```
<Access specifier> <return type> <name> (args,...)
{
}
```

We’re going to get into access specifiers later when we cover abstraction and encapsulation. For now, just assume that our access specifier is going to be the keyword public. So, let’s create a method called Test that takes in two integers a and b as arguments. If we followed our template correctly we should have something that looks like the following,

```
public double Test(int a, int b)
{
    return a + b;
}
```

Pretty cool, now let’s see the method in actions. Let’s modify our program file to match the following code snippet.

```
using System;

namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            double sum = Test(2, 1);
            Console.WriteLine("Sum: {0}", sum);
        }

        public static int Test(int a, int b)
        {
            return a + b;
        }
    }
}
```



The output from the program is.

Sum: 3

Okay, notice the

```
double sum = Test(2, 1);
```

What this line of code is doing is calling the method. Essentially, the conversation with the C# compiler will go something like this, Hey C#, call the Test method and tell it that a is 2 and b is 1 and take the return value and assign it the variable sum. Long story short, to call a method you just use its name and pass in the values that it is looking for.

Notice that we used the keyword static in our method declaration. That is because we are calling those methods from the Main method which is a static method. Not all methods have to be static as we will see in our next few examples. For now, think of static as a necessary evil that was needed for the method to be called by the main method. For the next few examples, we are not going to use the static keyword, so if you don't see static don't freak out is it not supposed to be there. We're going to learn what static means in the next chapter.

## **Classes and objects: What they are and what they're not**

Classes and I don't mean those things you fall or fell asleep in for the past several years. If you want to be a cool programmer, and by that, I mean an employed programmer, the one thing you must understand is classes and objects. Classes and objects are the backbones of modern programming, so with all that in mind what are they?

Everything needs a blueprint, the car you drive, your house, even your cat has some type of basic blueprint that when put on paper describes what it is and how it works. In programming, these blueprints are called classes. For example, consider a motor vehicle. For a vehicle to be a vehicle it needs some basic parts such as an engine, a transmission, and breaks. Each of these components does certain things and if we need to (and we will) create a vehicle class we will need the ability to call these components multiple times throughout the program's life cycle. As we've so far established programmers don't repeat themselves so we're going to turn those components into methods. So, our blueprint for a vehicle will look like this,

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApp2
{
    class Vehicle
    {
        public void engine()
        {
            Console.WriteLine("vrooom");
        }

        public void breaks()
        {
            Console.WriteLine("stop");
        }

        public void transmission(int rpms)
        {
            Console.WriteLine("RPMs: " + rpms);
        }
    }
}

```

Great, we now have a blueprint for a vehicle but that it's. For this blueprint to do anything we need to build it into something. This is where objects come into play. What is an object in real-life? An object is something that is built using some blueprints. In terms of object-oriented programming, an object is something you build using a class. Consider the following example,

```

using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Vehicle car = new Vehicle();
            Vehicle truck = new Vehicle();

            car.engine();
            car.breaks();
            car.transmission(1000);
        }
    }
}

```

```
truck.engine();
truck.breaks();
truck.transmission(1500);

    }
}
```

When we hit the green play button, we get the following output.



```
vroom
stop
RPMs: 1000
vroom
stop
RPMs: 1500
```

It looks like we just built a vehicle. Let's break this down, mainly the following two lines of code,

```
Vehicle car = new Vehicle();
Vehicle truck = new Vehicle();
```

These two lines of code create objects. We're telling C#, "hey we got two things that are a car and a truck. Construct them using the blueprints of a vehicle." In more technical programming lingo, we're creating two variables, car and truck, that are of type Vehicle. Since car and truck are two different objects of type Vehicle, we use the following syntax to access attributes that are in the class they reference,

```
<variable_name>.<attribute_name>
```

Blah, blah, blah. However, if a class is a blueprint of a thing how do we define what a class is and what it is not. Firstly, a class is not just a file that you dump attributes in. Believe it or not, this is quite common in the field. Many people do not fully consider their object-oriented design and simply use classes as files to dump attributes. This is one of the worst things you can do. Just dumping in random attributes will make code unscalable and unmaintainable, in short, your codebase will not age well. So, with this in mind how do we define what goes in a class? Well, the answer to that is simple, a class is a noun. If a class cannot be described as a noun you need to rethink think what that class does. Thinking about our example, our class,

vehicle, is a noun while our objects, car, and truck, are things. A good rule of thumb to make sure you define your classes and objects correctly is to ask yourself if your class is a noun and if your objects are things.

## **The four pillars of OOP**

Object-oriented programming is defined by four principles that are as follows:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

When used correctly these principles can mean the difference between code that will age well and code that will need to be re-written in a few months. So, let's break all these principles down.

### **Encapsulation**

Encapsulation is hiding attributes of a class. This may seem a little counter-intuitive at first because if you're taking the time to program in an attribute why hide it? Think about driving a car. When your speeding down the highway do you need to know or manually operate the inner working of the engine. Do you care that sparkplugs are firing and igniting the gas that drives down the pistons? Chances are that's going to be a major no. For the normal driver, the only thing they need to worry about is pressing their foot against the accelerator. In actuality, manually operating (if possible) and in some cases even knowing how the inner workings of the engine operates can be detrimental which is why working on a modern engine is like breaking into Fort Knox. Object-oriented programming should be treated the same way. If an object doesn't need to know about an attribute, then it needs to be as hidden as possible. In other words, you want your attributes to be as encapsulated as possible. Generally, you want the fewest parts of the program to be aware of an attribute as possible.

Attributes are hidden using access specifiers. Access specifiers tell the C# or any other language you're using how deeply to encapsulate an attribute. Depending on the language, the type and level of encapsulation an access specifier provides may vary; however, the three most common access

specifiers you'll encounter in the real-world are as follows:

Private = the attribute cannot be used outside the class

Public = any file or class can access the attribute

Protected = the attribute can only be accessed by an inherited class (We'll get to that later)

Another way to think of these is as numerical values.

Private = level 3

Protected = level 2

Public = level 1

The higher the level the harder it is to access the attributes from outside the class where level 3 is impossible to access from outside the class.

Access specifiers are keywords in pretty much every object-oriented language. As I stated before other languages have different levels of encapsulation but private, public, protected are the most common and you should know these by heart. So let's look at some code to see how these work

```
private int var1; //can only be access by call file  
public int var2; //can be accessed anywhere  
protected int var3; //can only be accessed by inherited classes
```

The variable declarations are just examples. In the real-world variables should almost always be set to private and manipulated through properties.

## Abstraction

Abstraction walks hand and hand with encapsulation. However, where encapsulation is hiding attributes, abstraction is showing the attributes. Again, this goes back to the public and protected attributes. Think about our car example. To drive a car a driver may not need to know the inner workings of an engine, but they do need to know how to press the gas pedal. Abstraction is the same thing. When developing a class, you need to consider what the "gas pedal" attributes are.

So, how do you know what level to declare your attribute at? Well, that depends on what you're trying to do. Generally, I like to set my access



specifiers to private unless I know it is going to be used by an outside class in which case, I'll set it to either protected or public. If I find that the attribute is only being used by inherited classes, I'll go ahead and lower the access level to protected. On the other hand, if I see that I'm using the attribute across multiple, non-inherited classes I'll set it to public. Theoretically, you should figure out the access level in the design phase; however, it is not uncommon to change access specifiers several times during development.

## Inheritance

The next pillar is inheritance. Inheritance in programming is similar to inheritance in the real-world. You're taking stuff from the parent class and using it in the child class. In the case of programming, the stuff you're taking from the parent class are public and protected attributes.

Inheritance is a very powerful tool to use in object-oriented programming. In short, it allows for code reusability. To see this let's look at the following code example,

```
//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Truck chevy = new Truck();

            chevy.engine(); //Vehicle
            chevy.breaks(); //Vehicle
            chevy.transmission(2000); //Vehicle
            chevy.TruckBed(10);
        }
    }
}

//Vehicle
using System;

namespace ConsoleApp2
{
    class Vehicle
    {
```

```

    public void engine()
    {
        Console.WriteLine("vroom");
    }

    public void breaks()
    {
        Console.WriteLine("stop");
    }

    public void transmission(int rpms)
    {
        Console.WriteLine("RPMs: " + rpms);
    }
}
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public void TruckBed(int length)
        {
            Console.WriteLine("Truck Length: " + length);
        }
    }
}

```

So, here we have two classes and a main class. In this case, the Vehicle class lays down the base for the Truck class which is why a class that is inherited from is often called a base class or parent class. Notice in the main program the object chevy is a reference to the Truck class but can still use all the attributes, in this case, methods, that are in the Vehicle class. The best way to think about inheritance is that you're taking all the attributes in one class and combining the public and protected attributes in another. At its heart, you are reusing existing code. C# has a very simple syntax for inheritance. To inherit another class, you use the following syntax in the class declaration

```
class childClass : baseClass
```

Inheritance is great; however, C# does have limits on inheritance. In short, C# only supports single inheritance. This means that at most you

can inherit from exactly one class and one class alone. However, just because you can inherit from one class does not mean you can't inherit from a class that inherits from another class.

Inheriting a class that inherits from another class is called an inheritance chain and it is a common technique in object-oriented programming. Consider our car example, we have a Truck class that inherits from a Vehicle class. Suppose our boss says our software needs to support a super duty truck. A super duty is a truck that among other things has a heavier chassis. So here we have a dilemma. We can add a method that handles heavy chassis to our Truck class which doesn't make a lot of sense since just because it is a truck doesn't make it a super duty. We could also make a super duty class and inherit from the Truck class. The lateral is a much better option because it separates the unneeded heavy chassis method from the truck class, and as such only trucks that are super duties can use it. At the same time, we can inherit from the truck class and use all of its methods and since the Truck class inherits from the Vehicle class all of those public and protected methods are a part of the Truck class as well. This means that by extension everything in the Vehicle class will be available in the newly minted super duty class. Let's demonstrate this with an example,

```
//Vehicle
using System;

namespace ConsoleApp2
{
    class Vehicle
    {
        public void engine()
        {
            Console.WriteLine("vroom");
        }

        public void breaks()
        {
            Console.WriteLine("stop");
        }

        public void transmission(int rpms)
        {
            Console.WriteLine("RPMs: " + rpms);
        }
    }
}
```

```
}

//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {

        public int getNumOfWheels(int numberOfWheels = 4)
        {
            return numberOfWheels;
        }

        public void TruckBed(int length)
        {
            Console.WriteLine("Bed Length: " + length);
        }

    }
}
```

```
//super duty
using System;

namespace ConsoleApp2
{
    class SuperDuty : Truck
    {
        public void HeavyChassis()
        {
            Console.WriteLine("Heavy Chassis!");
        }
    }
}
```

```
//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            SuperDuty f350 = new SuperDuty();
        }
    }
}
```

```

        Console.WriteLine("Super Duty");
        f350.breaks();
        f350.engine();
        f350.getNumOfWheels(6);
        f350.HeavyChassis();
        f350.TruckBed(6);
    }
}

```

As can be seen in the Main method, the f350 object which is of type SuperDuty can access all the attributes in the Tuck class and the Vehicle class.

I've seen many developers try to cheat and use the inheritance chain as a means of multiple inheritances. This is one of the worst things you can do because you end up lumping together functionality that shouldn't be lumped together. In turn, the inheritance chain can be abused and as a result, you're going to have software this hard to maintain. Use the inheritance chain wisely or don't use it at all!

## Polymorphism

Polymorphism, a big word with a simple meaning. Ignoring the long overly nerdy explanation, polymorphism, in the real-world means you are changing the definition or implementation of an inherited method in the child class. There are two types of polymorphism, dynamic and static. Dynamic polymorphism occurs at run-time and is associated with method overriding as we will see in the example below, method overriding is when you redefine a method in a child class that was inherited from a parent class.

Consider our last example We have a truck class that is derived from a vehicle class, like so,

```

//Vehicle
using System;

namespace ConsoleApp2
{

```



```

class Vehicle
{
    public void engine()
    {
        Console.WriteLine("vroom");
    }

    public void breaks()
    {
        Console.WriteLine("stop");
    }

    public void transmission(int rpms)
    {
        Console.WriteLine("RPMs: " + rpms);
    }
}
//Truck
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public void TruckBed(int length)
        {
            Console.WriteLine("Truck Length: " + length);
        }
    }
}

```

In this case, the truck method inherits and can use the Engine method that goes vroom, but like a real truck, it doesn't make a vroom sound it makes a rumbling sound. So, to solve this problem we can override the engine method in the Truck class. To do this we need to use the override and the virtual keywords.

So, to start let's explore the virtual keyword. When you use the virtual keyword, you are creating what is called a virtual method. A virtual method is a method that can be overridden in a derived class. So, we're going to take our vehicle class and turn the Engine method into a virtual function that we're going to override.

```
//Vehicle
using System;

namespace ConsoleApp2
{
    class Vehicle
    {
        public virtual void engine()
        {
            Console.WriteLine("vrooom");
        }

        public void breaks()
        {
            Console.WriteLine("stop");
        }

        public void transmission(int rpms)
        {
            Console.WriteLine("RPMs: " + rpms);
        }
    }
}
```

This code creates a virtual method called Engine. Great, now how do we override it? Well, as I'm guessing you've already figured out, we use the override keyword in the derived class, or for this case the Truck class.

The simplest way to think of the override keyword is as a way of telling C# that this is the new implementation of the method. Let's take a look at the following,

```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public void TruckBed(int length)
        {
            Console.WriteLine("Truck Length: " + length);
        }

        public override void engine()
        {
            Console.WriteLine("rumble");
        }
    }
}
```

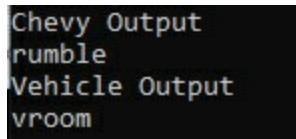
```

    }
}
}
//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Truck chevy = new Truck();
            Vehicle vehicle = new Vehicle();
            Console.WriteLine("Chevy Output");
            chevy.engine();
            Console.WriteLine("Vehicle Output");
            vehicle.engine();

        }
    }
}

```



```

Chevy Output
rumble
Vehicle Output
vroom

```

As can be seen, we changed the Truck's definition to say rumble instead of vroom. However, we can still call the original definition of the engine which still says vroom. We can still use the original engine method from the base class either by calling it a base class object or an object that does not override the method.

Consider this, suppose Chevy has a race truck with a high-performance engine that goes vroom. In this case, our overridden method that goes rumble won't work, we either need to reimplement the original method or figure out a way to reuse the old engine method in the vehicle class. Since, the whole hoopla about object-oriented languages is code reusability the right thing to do is use the old method, and this is where the base method comes in.

The base keyword allows us to access the members in the base class.

So, if we wanted to use the base class's version of the engine method, we would use something like the following,

```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public void TruckBed(int length)
        {
            Console.WriteLine("Truck Length: " + length);
        }

        public override void engine()
        {
            Console.WriteLine("rumble");
        }

        public void truckType(int choice)
        {
            if (choice == 1)
                engine();
            else
                base.engine();
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Truck chevy = new Truck();
            Console.WriteLine("Regular Chevy");
            chevy.truckType(1); //regular truck
            Console.WriteLine("Race Truck");
            chevy.truckType(2); //race truck
        }
    }
}
```

```
Regular Chevy  
rumble  
Race Truck  
vroom
```

We added a method to the Truck class called truckType. The truckType method takes in an integer, if the passed-in integer is 1 it calls the overridden version the engine method which says rumble. If the passed-in integer is 2 it uses the base keyword to call the base class's version the engine method which says vroom. Overall, the moral of this story is that just because a method has been overridden by a class that class can still use the base class's implementation of the method.

Now, that we have a decent understanding of dynamic polymorphism we need to examine static polymorphism. Static polymorphism occurs at compile time and is associated with method overloading and operator overloading. In my experience, the most common form of static polymorphism is method overloading. Method overloading occurs when two methods have the same name but one or both of the following occurs,

- 1) The number of parameters differs between the two methods
- 2) The datatypes of the parameters differ between the two methods

What's cool about method overloading is as soon as one or more of the above conditions are met you are essentially creating a new method which means the logic and return type may differ between the methods.

Going back to our truck example, a truck can be either a long bed or short bed truck. For our example, we'll assume that a short bed truck's length will only have lengths in integers and a long bed truck will only come in double lengths. So, we'll need to add another TruckBed method that accepts double values in its parameter list like the following,

```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public void TruckBed(int length)
        {
```



```

        Console.WriteLine("Short Bed Length: " + length);
    }

    public void TruckBed(double length)
    {
        Console.WriteLine("Long Bed Length: " + length);
    }

    public override void engine()
    {
        Console.WriteLine("rumble");
    }

    public void truckType(int choice)
    {
        if (choice == 1)
            engine();
        else
            base.engine();
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Truck chevy = new Truck();
            chevy.TruckBed(2);//short bed
            chevy.TruckBed(3.5);//long bed
        }
    }
}

```

```

Short Bed Length: 2
Long Bed Length: 3.5

```

As can be seen, by simply passing in arguments of different datatypes we were able to, essentially call two different methods with the same name.

Now let's look at changing the return type. Currently both TruckBed methods have a void return type. Let's see what happens when we modify one of the methods to have a return type of double.

```

//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public void TruckBed(int length)
        {
            Console.WriteLine("Short Bed Length: " + length);
        }

        public double TruckBed(double length)
        {
            return 2 * length;
        }

        public override void engine()
        {
            Console.WriteLine("rumble");
        }

        public void truckType(int choice)
        {
            if (choice == 1)
                engine();
            else
                base.engine();
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Truck chevy = new Truck();
            chevy.TruckBed(2); //short bed
            double truckLenth = chevy.TruckBed(3.5); //long bed
            Console.WriteLine("Long Bed Length : " + truckLenth);
        }
    }
}

```

```
Short Bed Length: 2
Long bed length : 7
```

As can be seen, we were also able to modify the return type by changing the arguments. This form of polymorphism is very handy but can be very dangerous because someone can call the right method but pass in different arguments and get the wrong results. So, when you're designing a program pay attention to how you design and document overloaded methods.

The final form of static polymorphism is operator overloading. Operator overloading essentially assigns a special operation to an operator like the + or - sign. In terms of C# you're altering the behavior of one of the operators in terms of an associated class's objects. Let's assume we're modifying our car class to accommodate a new car that has the combined length of a Mustang and a Focus. Consider the following code example,

```
namespace ConsoleApp2
{
    class Car : Vehicle
    {
        private double length;
        public double CarLength
        {
            get { return length; }
            set { length = value; }
        }

        public static Car operator+ (Car Mustang, Car Focus)
        {
            Car hybrid = new Car();
            hybrid.length = Mustang.length + Focus.length;
            return hybrid;
        }

        public double Hybrid()
        {
            return length;
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
```

```

{
    static void Main(string[] args)
    {
        Car Mustang = new Car();
        Car Focus = new Car();
        Car hybrid = new Car();

        Mustang.CarLength = 12.2;
        Focus.CarLength = 2.3;
        hybrid = Mustang + Focus;

        Console.WriteLine("Hybrid length: " + hybrid.Hybrid());
    }
}

```

Hybrid lenght: 14.5

Let's take a look at what's going on here, mainly let's analyze the method with the operator keyword in it. This is the method responsible for overloading the + symbol. Essentially, what we're doing is overriding the logic associated with the plus sign. As can be seen in that method we are passing in two-car objects and using the method to create a new car object, assign the sum of the two-car lengths, and return the new car object. Now, consider the Mustang + Focus line in the main program. In this case, the plus sign acts as the method with Mustang and Focus as the arguments that are being passed in.

At first glance operator overloading may seem a bit confusing and a bit useless. This form of polymorphism is best understood by playing with it. I would suggest using the above example as a basis and experimenting with it. Try experimenting with the logic, try overloading different operators, try just having fun with it.

## Relationships: Inheritance vs Composition

Relationships, and no I don't mean the ones where you get to kiss and hold the ones you love ever so tightly. I'm talking about class relationships, not as fun, but when you do it right, they keep the checks rolling in, so you get to keep your relationship with the one you love. Relationships in programming are very important but are often misunderstood. Relationships tie into a program's architecture and mean the

difference between a loosely coupled architecture that is easy to maintain and a program that is unnecessarily tightly coupled and is hard to maintain.

Classes in a program can have many different relationships; however, the two most common relationships you will see are *is-a* and *has-a*. The *is-a* and *has-a* relationships correspond to two different class consumption techniques. When there's an *is-a* relationship you use inheritance as we did before. When there is a *has-a* relationship you use a technique called composition. Chances are you've used composition before but just never realized it.

Inheritance has its place in object-oriented programming, when used properly it is a very powerful tool. However, inheritance prompts a tightly coupled architecture, and in many cases that limits the flexibility of a program. This means that if something goes wrong in a base class you just killed the child class as well. All too many programmers, especially newbies, love to use inheritance without considering the underline relationships.

Let's consider our current vehicle program, so far, we have a car and a truck class that inherits from the vehicle class. This is fine, we have a car which is a vehicle and a truck which is also a vehicle. Now, suppose our boss asks us to modify the Vehicle class to accommodate a method that keeps track of the number of doors a vehicle has. So, we go ahead and do that, and the new Vehicle class looks like this,

```
//Vehicle
using System;

namespace ConsoleApp2
{
    class Vehicle
    {
        public void engine()
        {
            Console.WriteLine("vrooom");
        }

        public void breaks()
        {
            Console.WriteLine("stop");
        }

        public void transmission(int rpms)
```



```

    {
        Console.WriteLine("RPMs: " + rpms);
    }

    public void DoorCount(int doors)
    {
        Console.WriteLine("Number of doors: {0}", doors);
    }
}
}

```

Now, a few days go by and our manager tells us that our software is now going to support motorcycles. Since a motorcycle is a vehicle, we follow the same pattern that we've been using, and we create a class named motorcycle that inherits from Vehicle and we create a motorcycle object. The motorcycle class should look like the following,

```

using System;

namespace ConsoleApp2
{
    class motorcycle : Vehicle
    {
        public void handlebars()
        {
            Console.WriteLine("Turn");
        }
    }
}

```

and we'll modify the Main method to create an object for all three vehicles.

```

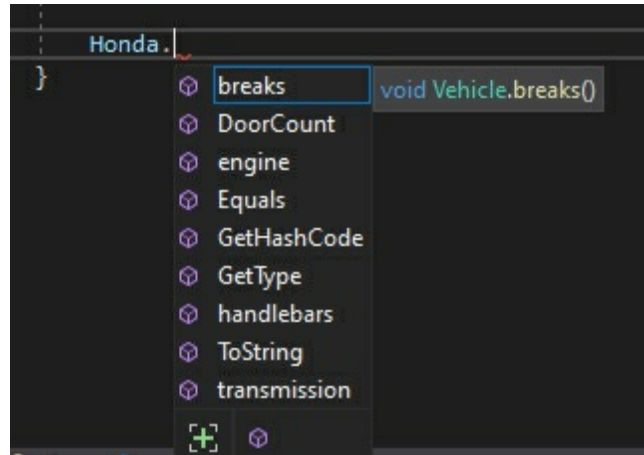
//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car Mustang = new Car();
            Truck Chevy = new Truck();
            motorcycle Honda = new motorcycle();
        }
    }
}

```

```
}  
}
```

Now, take a look at the following screenshot.



As you can see, we inherited a door count for a vehicle that doesn't have doors! Is this the end of the world, well no. We can just ignore the DoorCount method, but what if our boss wants to add seatbelts later? Do we ignore that, what if at another point in time our boss wants to add a new vehicle called boat to the program? We'll eventually end up with objects that don't fit into the inheritance hierarchy well and they'll have so much useless functionality that mistakes are bound to happen. So, what can we do to fix this problem? We can use composition.

To use composition for our program we need to redesign the program so that the class has a has-a relationships. So, we can break the vehicle class out and instead have Engine, Transmission, Breaks, and DoorCount classes.

```
//Engine  
using System;  
  
namespace ConsoleApp2  
{  
    class Engine  
    {  
        public void engine()  
        {  
            Console.WriteLine("vrooom");  
        }  
    }  
}
```

```

}

//Transmission
using System;

namespace ConsoleApp2
{
    class Transmission
    {
        public void transmission(int rpms)
        {
            Console.WriteLine("RPMs: " + rpms);
        }
    }
}

//Breaks
using System;

namespace ConsoleApp2
{
    class Breaks
    {
        public void breaks()
        {
            Console.WriteLine("stop");
        }
    }
}

//DoorCount
using System;

namespace ConsoleApp2
{
    class DoorCount
    {
        public void doorCount(int doors)
        {
            Console.WriteLine("Number of doors: {0}", doors);
        }
    }
}

```

Okay, now we broke everything out into individual classes. Now, we can construct our car, truck, and motorcycle using composition.

Let's think about what components a car and truck have,

- An engine
- A transmission
- Breaks
- Multiple doors

Now let's do the same for a motorcycle,

- An engine
- A transmission
- Breaks

Now we can construct the classes,

```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck
    {
        public Engine engine = new Engine();
        public Breaks breaks = new Breaks();
        public Transmission transmission = new Transmission();
        public DoorCount doors = new DoorCount();
    }
}

//Car
using System;

namespace ConsoleApp2
{
    class Car : Vehicle
    {
        public Engine engine = new Engine();
        public Breaks breaks = new Breaks();
        public Transmission transmission = new Transmission();
        public DoorCount doors = new DoorCount();
    }
}

//motorcycle
using System;

namespace ConsoleApp2
{
    class motorcycle : Vehicle
    {

```

```

    public Engine engine = new Engine();
    public Breaks breaks = new Breaks();
    public Transmission transmission = new Transmission();
}
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car Mustang = new Car();
            Truck Chevy = new Truck();
            motorcycle Honda = new motorcycle();

            Console.WriteLine("Mustang");
            Mustang.breaks.breaks();
            Mustang.doors.doorCount(2);
            Mustang.engine.engine();
            Mustang.transmission.transmission(2500);
            Console.WriteLine("-----");

            Console.WriteLine("Chevy");
            Chevy.breaks.breaks();
            Chevy.doors.doorCount(4);
            Chevy.engine.engine();
            Chevy.transmission.transmission(1500);
            Console.WriteLine("-----");

            Console.WriteLine("Motorcycle");
            Honda.breaks.breaks();
            Honda.engine.engine();
            Honda.transmission(2000);

        }
    }
}

```

```
Mustang
stop
Number of doors: 2
vroom
RPMs: 2500
-----
Chevy
stop
Number of doors: 4
vroom
RPMs: 1500
-----
Motorcycle
stop
vroom
RPMs: 2000
```

Now, as can be seen when we use composition all we need to do was create an object to the class for the component in the class of the vehicle. The composition relationship also offers much more flexibility, since the components are broken out into their classes, we can add additional functionality to them without the said functionality being out of place. For example, if we wanted to, we could add cylinder count to the Engine class where if we added cylinder count to the Vehicle class, we would start to lose clarity to the underline blueprint of the vehicle.

Some developers say composition should always be used over inheritance and to that I say they need a head examination. Both inheritance and compositions are tools. Saying that one should always be used over the other is like saying you should always use a hammer over a screwdriver. I, like many others, favor composition when developing the architecture of a program. Composition will force you to use more classes but allows you to scale the functionality of the classes and makes the code more modular. I can't tell you if you should use composition over inheritance, all I can say is that it depends on what you're trying to accomplish.

# CHAPTER 2: METHODS

## REVISED: THE HARD STUFF

We all know what methods are, right? We just used them in the last chapter. They're named, callable blocks of code that can return a value and can take arguments. However, what is that darned static keyword actually doing in the method declaration? What does pass-by-value versus pass-by-reference mean? What is recursion? Why am I asking so many questions? To answer the last one, I'm building suspense. The answers to all the others and more will be found below.

### Static, that darned keyword explained

Now's a good time to look at what that pesky static keyword means. If you take a look at the Add method in the below code snippet you'll see the keyword static and if you look at our methods in the vehicle class, you'll see that the static keyword is nowhere to be found. That's because when an attribute, such as a method, is declared as static you're telling C# that the method belongs to the class and not to an object. So, what does that mean? Let's look at some static variables to demonstrate this.

```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public static int wheels;

        public int getNumOfWheels()
        {
            return wheels;
        }
    }
}
```



```

    }

    public void TruckBed(int length)
    {
        Console.WriteLine("Short Bed Length: " + length);
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Truck chevy = new Truck();
            Truck ford = new Truck();

            Truck.wheels = 4;

            Console.WriteLine("Ford's Wheel Count: {0}", ford.getNumOfWheels());
            Console.WriteLine("Chevy's Wheel Count: {0}", chevy.getNumOfWheels());

        }
        public static int Add(int a, int b)
        {
            return a + b;
        }
    }
}

```

```

Ford's Wheel Count: 4
Chevy's Wheel Count: 4

```

First, in the truck class notice, we declared that the wheels variable as static. This means that we are telling the C# that the wheel variable belongs to the class and not to an object. Now, move your attention to the

```
Truck.wheels = 4
```

Notice, that we don't use an object to manipulate this variable. Again, this

falls back to the variable belonging to the class and not the object. Once you manipulate a static attribute that change will be seen across all the objects, so be careful.

Now there are a few rules when dealing with static members. A non-static member can call or manipulate a static member. For example, consider the following code,

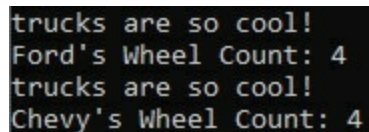
```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public static int wheels;

        public int getNumOfWheels()
        {
            message();
            return wheels;
        }

        public static void message()
        {
            Console.WriteLine("trucks are so cool!");
        }

        public void TruckBed(int length)
        {
            Console.WriteLine("Short Bed Length: " + length);
        }
    }
}
```



```
trucks are so cool!
Ford's Wheel Count: 4
trucks are so cool!
Chevy's Wheel Count: 4
```

In this case, the `getNumOfWheels` method, which is a non-static method can call `message` which is a static method. However, trying to call `getNumOfWheels` method from the `message` methods will result in the

following:

```
public static void message()
{
    getNumOfWheels();
    Console.Wri
    int Truck.getNumOfWheels()
}
An object reference is required for the non-static field, method, or property 'Truck.getNumOfWheels()'
```

In other words, a static member cannot manipulate or call a non-static member. This means that all methods that are called by the Main method have to be static. It should also be noted, that since static attributes belong to the class and not an object, all static members whether they be public or not will be invisible and un-callable to the object variable.

Static members are very handy but can be very dangerous. As we saw with the wheel example if you accidentally change a static member you just changed that member for all the objects. However, for the same reason sometimes you may want the changes that are caused by calling on a static member to rickshaw through all the objects. In fact, some developers choose a static class, which is a class that only contains static members, over the Singleton pattern; however, that is religious dogma and I'm not getting into that here.

## Pass-by-reference vs pass-by-value

Pass-by-reference versus pass-by-value is one of the most commonly asked interview questions I have ever heard for entry-level positions, it is on almost every programming test you're ever going to take, it'll be written on your tombstone, and it is one of the most overlooked concepts in programming. When used properly though it can really speed up your software. So, with all that being said, what is the difference between pass-by-reference and pass-by-value?

Pass-by-reference versus pass-by-value is actually a pretty easy concept, however, it is one of those concepts that for the average programmer writing average software is for the most part irrelevant. Pass-by-value means that when arguments are passed to a method the value of the variable being passed is actually being copied to the method's argument variable. This means that any operation done to the variable is not reflected across the

program because the values are in two different memory blocks. By default, C# and most other modern OOP languages are pass-by-value by default, as such all of the past examples where values are passed to arguments are examples of pass-by-value.

Pass-by-reference means that when an argument is passed the address of the memory block is passed to the argument. This means that when an operation is performed on that argument it is being done in the memory block itself which in turn means that change will reflect across the whole program. In C# you have to specify that you want to pass a variable by reference with the `ref` keyword.

```
using System;

namespace ConsoleApp2
{
    class Car : Vehicle
    {
        public void SetNumberOfPassengers(ref int numOfPassengers)
        {
            numOfPassengers = 4;
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car mustang = new Car();

            int passengers = 2;
            mustang.SetNumberOfPassengers(ref passengers);
            Console.WriteLine("Number of passengers: {0}", passengers);
        }
    }
}
```

Number of passengers: 4

Now, the result from the code looks confusing at first glance. In the Car class, we created a method called `setNumberOfPassengers` and we used the `ref` keyword in the arguments. In the method, we set the number of passengers to 4. Now, moving to the main class we created a variable called `passengers` and set it to two. We then passed in the variable using the `ref` keyword to the `setNumberOfPassengers` method which as we established set the passed in value to four. Lastly, we displayed the `passengers` variable again and it changed to four. Normally, this would not have happened. Normally, what happens in the method stays in the method, but this time we passed by reference so, what we actually did was copy the memory address of the passenger variable to the `numOfPassengers` argument variable. As such, when we performed an operation on the `numOfPassengers` we altered the contents of that memory address which is the address of the passenger variable which means that the passenger variable will now reflect the change.

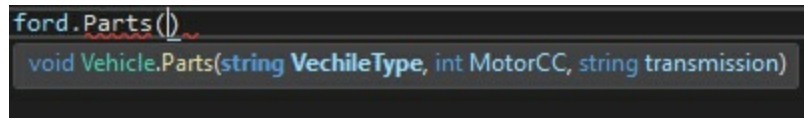
Pass-by-reference for primitive types is usually considered bad because it defeats the purpose of encapsulation. Passing by reference can make it difficult to trace bugs in programs due to potentially any number of methods causing the issue. However, pass-by-reference does have some handy uses. Generally, it is faster to pass objects by reference than it is to pass them by value. When you pass an object by value the computer has to make a copy of all the data associated with that object, but when passing by reference you only pass in the memory address of that object. So, if you need to speed up your program and can afford a global change in the object pass-by-reference may be the way to go.

## **Named parameters**

Okay, it's your first day at a new job, it's your dream job. You're tasked with creating a new method to order parts for a car. Since this is your dream job you want to impress your bosses and you start to get really nervous. You get so nervous, that you break out in hives and accidentally have a brainstem stroke. Worst of all you forget the order of the arguments for your new method. At this point, your life is looking bleak. You can either get drunk on the job, hopefully go see a doctor about your new brain condition, or you can just re-write the method using named parameters. Hopefully, you'll see a brain doctor first before you re-write the method, but

hey it's your life.

C# 4.0 introduces a neat little concept known as name parameters. On its own the newer versions of Visual Studios makes the named parameters concept kind of pointless because the moment we type in the method name the order the arguments are declared in is shown,



```
ford.Parts()  
void Vehicle.Parts(string VechileType, int MotorCC, string transmission)
```

However, when mixed with optional parameters that we're going to review here in a bit it is a very powerful tool.

Anyways, named parameters allow you to pass in an argument in any order you choose. Consider the following example,

```
//Vehicle  
using System;  
  
namespace ConsoleApp2  
{  
    class Vehicle  
    {  
        public void breaks()  
        {  
            Console.WriteLine("stop");  
        }  
  
        public void transmission(int rpms)  
        {  
            Console.WriteLine("RPMs: " + rpms);  
        }  
  
        public void Parts(string VechileType, int MotorCC, string transmission)  
        {  
            Console.WriteLine("ordering car type {0}", VechileType);  
            Console.WriteLine("ordering for motor cc {0}", MotorCC);  
            Console.WriteLine("ordering for transmission {0}", transmission);  
        }  
    }  
}  
  
//main  
using System;
```

```

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Vehicle vehicle1 = new Vehicle();
            vehicle1.Parts(MotorCC: 1000, transmission: "Ford", VechileType: "Truck");
        }
    }
}

```

Notice in the

```

vehicle1.Parts(MotorCC: 1000, transmission: "Ford", VechileType: "Truck");

```

that the arguments are pass in no particular order. All we did was specify the variable's name followed by a colon and the value. In a nutshell, this is the principle behind named parameters.

## Optional parameters

Consider this crazy notion, suppose we have a method and at least one of the arguments only changes sometimes. For the 9/10 times, we call that method the argument is going to be the same. Do we really want to pass in that value all the time? What if you're typing and you accidentally press the wrong key and the mistake gets by QA? The next thing you know the ground is opening up, people are shouting in the streets, the sky turns red, and it starts raining corn, I don't know why it is raining corn but I didn't press the wrong key. So, how can we avoid this mayhem? We could use method overriding and create a separate method with that value set as a default value in that version of the method. Well, that may work for one maybe two arguments, but what if there are 500 arguments and each argument may or may not have a default value? Also, if you have a method with 500 arguments you may want to re-think your design, and probably your life choices as well. For this situation, our best bet would be to use what is known as optional parameters.

As the name suggests optional parameters are arguments that you do not have to pass into the method. The reason we don't have to pass in a value for the argument(s) is because when we declare the method, we give

those arguments a preset value. Consider the example below.

```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {

        public int getNumOfWheels(int numberOfWheels = 4)
        {
            return numberOfWheels;
        }

        public override void engine()
        {
            Console.WriteLine("rumble");
        }

        public void TruckBed(int length)
        {
            Console.WriteLine("Short Bed Length: " + length);
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car mustang = new Car();
            Truck chevy = new Truck();
            Truck ford = new Truck();
            Truck mack = new Truck();

            Console.WriteLine("Number of wheels for the Chevy: {0}", chevy.getNumOfWheels());
            Console.WriteLine("Number of wheels for the Ford: {0}", ford.getNumOfWheels());
            Console.WriteLine("Number of wheels for the Mack: {0}", mack.getNumOfWheels(18));

        }
    }
}
```



```
}
```

```
Number of wheels for the Chevy: 4  
Number of wheels for the Ford: 4  
Number of wheels for the Mack: 18
```

Here we have three Truck objects, a Chevy, a Ford, and a Mack. A standard Ford and Chevy truck are normally going to have four wheels, but a Mack truck is an 18-wheeler and as such has 18 wheels. To compensate for this not so common truck notice what did to this method,

```
public int getNumOfWheels(int numberOfWheels = 4)  
{  
    return numberOfWheels;  
}
```

In the argument declaration, we set the numberOfWheels variable to four. This declaration tells C# that if we don't pass in a value for numberOfWheels that's cool just assume it's four.

Now, what if we have more than one argument? What if we have three arguments? How do we manage optional parameters then? This is where name parameters shine. Suppose we need a new method that sets the characteristics of a car. This method needs to take the number of doors, the transmission type, and the make. For this method the make is always going to change; the number of doors is usually going to be four, but it can be a coupe with two doors, and the transmission is usually going to be automatic, but it can be standard. So, let's play with this example and see what happens,

```
using System;  
  
namespace ConsoleApp2  
{  
    class Car : Vehicle  
    {  
        public void SetNumberOfPassengers(ref int numOfPassengers)  
        {  
            numOfPassengers = 4;  
        }  
  
        public void carChar(string make, int numberOfDoors = 4, string transmission = "Automatic")  
        {  
            Console.WriteLine("Number of doors: {0}", numberOfDoors);  
            Console.WriteLine("Transmission type: {0}", transmission);  
        }  
    }  
}
```

```

        Console.WriteLine("Car make: {0}", make);
    }

}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car altima = new Car();

            altima.carChar("Nissan");
        }
    }
}

```

```

Number of doors: 4
Transmission type: Automatic
Car make: Nissan

```

Notice in the main program that we created a car object that is an Altima. By default, a Nissan Altima is a family car and as such, it is a sedan which means it has four doors and is usually automatic. As such we can use the default values and we only have to pass in the car's make which is Nissan.

Now, what if we have a Mustang Sports car? A Mustang has two doors, but they can come in either manual or automatic transmissions. So, let's see how we can use named parameters to help us here, let's look at the following code,

```

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car Mustang = new Car();

            Mustang.carChar("Ford", numberOfDoors: 2); //automatic mustang
        }
    }
}

```

```
    Mustang.carChar("Ford", numberOfDoors: 2, transmission: "standard");//standard mustang
  }
}
```

```
Number of doors: 2
Transmission type: Automatic
Car make: Ford
-----
Number of doors: 2
Transmission type: standard
Car make: Ford
-----
```

First off, when you run the Mustang example above and don't get the line don't freak out, you're fine. I added these lines in my code to make it easier for you to read. Now, turn your attention to the automatic Mustang line in the example. As you can see, we passed in the car's make which is Ford. Then we specified the number of doors with a named parameter, but we left the transmission type alone and we got the first block of text which is expected. On the next line we specified everything we did in the first block, but this time used named parameters to change the transmission to standard and we got the second block of text which is also as expected.

There is a gotcha when using optional parameters. All mandatory parameters, that is, parameters that do not have a default value must be declared before optional parameters are declared. This is an annoying little gotcha because it is something that is easily looked over so if you get a squiggle check to make sure your mandatory parameters are declared before your optional ones are.

So, moving on the moral of the story here is that if you use optional parameters chances are, you're going to need to use named parameters as well. Many developers, myself included, usually see named and optional parameters as two halves of the same whole. So, if you're going to use optional parameters be prepared to use name parameters as well.

## **Recursion: What it is and why you should never use it**

If you interview with a Boomer be prepared to get quizzed on Recursion. Why old school programmers feel the need to bother with Recursion is beyond me. Recursion is slow, clunky, and generally outdated in OOP programming, but those grey-haired grandmas and grandpas love to

quiz us youngsters on it anyways. The only time you need to worry about recursion is when you're either on an interview or using a functional programming language like F#. However, for the sake of learning, we're going to cover it here.

Recursion is a simple enough concept to understand but if you're using it heavily in a modern piece of software you need to stop and learn how to effectively use loops. Recursion is simply a method that calls itself. It's a way of looping. Personally, I have used recursion once in my entire career and I could have used a loop, but I didn't have the time to really mess with it, so I took the I'm in a hurry route and used recursion. Modern OOP programmers generally frown on recursion as it is slower and more resource heavy. To make matters worse for all the recursion-loving Boomers many compilers optimize recursive code by converting recursive methods to loops.

Recursion is also dangerous as you can easily create an endless recursive cycle. Essentially, your method will be calling itself until the end of time if you don't put the proper exit statements in. Now, most compilers will warn you of the dreaded endless recursive loop and won't start the program until it finds an exit path. However, no matter how you try to sugar coat it, recursion is still bad, so let's look at an example.

```
using System;
namespace ConsoleApp2
{
    class Car : Vehicle
    {
        public void SetNumberOfPassengers(ref int numOfPassengers)
        {
            numOfPassengers = 4;
        }

        public void carChar(string make, int numberOfDoors = 4, string transmission = "Automatic")
        {
            Console.WriteLine("Number of doors: {0}", numberOfDoors);
            Console.WriteLine("Transmission type: {0}", transmission);
            Console.WriteLine("Car make: {0}", make);
            Console.WriteLine("-----");
        }

        public void rev(int i = 0)
        {
```

```

        if (i < 3)
        {
            Console.WriteLine("rev");
            rev(++i);
        }

    }
}
//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car Mustang = new Car();

            Mustang.rev();
        }
    }
}

```

Let's look at our car class first. We modified the class and added a method called rev. The rev method has a default parameter of 0. Notice the code in the if statement,

```

if (i < 3)
{
    Console.WriteLine("rev");
    rev(++i);
}

```

The second line in the if statement calls the rev method and passes in whatever the variable i is plus one. It'll keep calling that method as long as the if condition is true. Congratulations you know understand recursion in a nutshell.

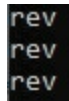
Now, what would happen if we took the if statement out? First off, the method will be called, and we'll eventually get this

Stack overflow.

In other words, we broke the program. This is why recursion is dangerous. If there is a weird logic condition that isn't accounted for, or something goes wrong, well, you just killed the program.

Looking at the rev method again, we wanted to rev the engine three times. A much safer, faster, and effective way of writing that method would be with a for loop.

```
public void rev()  
{  
    for(int i = 0; i < 3; ++i)  
    {  
        Console.WriteLine("rev");  
    }  
}
```



As can be seen in the output, we get the same result.

In all honesty, I don't think recursion has a place in modern object-oriented programming. There is simply too much that can go wrong. I did a little research on recursion and why it is such a popular interview topic and I was not satisfied with any of the answers. I read answers ranging from they ensure the candidate can grasp elegance to the candidate understanding algorithmic structures. Sure, some algorithms are built around recursion, but chances are you're never going to use those. However, as we saw with the last example recursion bloated and drastically over-complicated what should have been a very simple method. Generally, the responses that I've heard thus far and the people that defend recursion are usually old-timers that are holding on to old ways or people that read too many *This is how you interview programmers'* tutorials. Do I think you should have a basic understanding of recursion? Well, yes. Should you ever use recursion in the real-world? Well, no! Honestly, I believe the only people that have ever asked me anything about recursion are very old-timers, arrogant punk managers, or hiring managers that didn't know the first thing about software. If you ever have to defend yourself for why you never use recursion you may want to seriously reconsider the people you're working with.

## Constructors

A constructor is a special method in a class. A constructor can accept arguments just like any other method, but it does not have a return type and the constructor has the same name as the class. A constructor is automatically called when a class is initiated. As such, constructors are good for doing the needed tasks to prep the object. Usually, constructors are used to establish database connections, connections to hardware, prep GUIs, and set variables. The basic syntax for creating a constructor is the following,

```
public <class_name>(args,..)
{
}
```

Suppose our boss says that our program needs to fuel the vehicle before they are can run. Common sense, right? A car or truck needs gas to run. Gas is a vital resource for a vehicle. Without gas we're going nowhere. Gas should be added automatically when the object is created, so to make sure the vehicle has gas we can use a constructor to fill the tank. Consider the following,

```
//Vehicle
using System;

namespace ConsoleApp2
{
    class Vehicle
    {
        //constructor
        public Vehicle()
        {
            Console.WriteLine("filled the tank");
        }

        public void engine()
        {
            Console.WriteLine("vroom");
        }

        public void breaks()
        {
            Console.WriteLine("stop");
        }

        public void transmission(int rpms)
        {

```

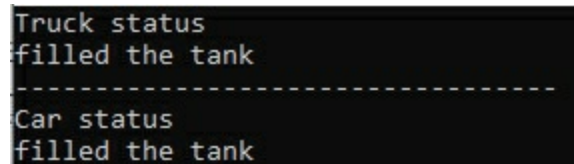
```

        Console.WriteLine("RPMs: " + rpms);
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Truck status");
            Truck Chevy = new Truck();
            Console.WriteLine("-----");
            Console.WriteLine("Car status");
            Car Mustang = new Car();
        }
    }
}

```



```

Truck status
filled the tank
-----
Car status
filled the tank

```

Notice in the Main method all we did was create an object and we still fired the constructor method. This is because the constructor is called as soon as an object is created. Also, notice that our objects are of type Car and Truck and the constructor that was called was in the vehicle class. Therefore, the constructor at the end of the day is just another method and, as such, it gets inherited just like other methods. Now, let's add a constructor method to the Car and Truck class.

```

//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        //constructor
    }
}

```



```

public Truck()
{
    Console.WriteLine("The truck is filled!");
}

public int getNumOfWheels(int numberOfWheels = 4)
{
    return numberOfWheels;
}

    public void TruckBed(int length)
    {
        Console.WriteLine("Bed Length: " + length);
    }
}

//Car
using System;

namespace ConsoleApp2
{
    class Car : Vehicle
    {
        //constructor
        public Car()
        {
            Console.WriteLine("The car is filled!");
        }

        public void SetNumberOfPassengers(ref int numOfPassengers)
        {
            numOfPassengers = 4;
        }

        public void carChar(string make, int numberOfDoors = 4, string transmission = "Automatic")
        {
            Console.WriteLine("Number of doors: {0}", numberOfDoors);
            Console.WriteLine("Transmission type: {0}", transmission);
            Console.WriteLine("Car make: {0}", make);
            Console.WriteLine("-----");
        }

        public void rev()
        {

```

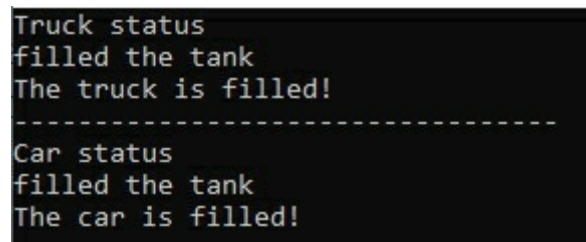
```

        for(int i = 0; i < 3; ++i)
        {
            Console.WriteLine("rev");
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Truck status");
            Truck Chevy = new Truck();
            Console.WriteLine("-----");
            Console.WriteLine("Car status");
            Car Mustang = new Car();
        }
    }
}

```



```

Truck status
filled the tank
The truck is filled!
-----
Car status
filled the tank
The car is filled!

```

As can be seen, the constructors for the child classes are also called. Also, notice that the constructor for the base class was called before the constructor for the child class. When working with constructors and inheritance is involved remember you're working from the base class on up to the child class.

Great, so now we can fill up our car and truck but there's a little problem our brilliant, drug-addicted boss forgot about, cars and trucks have different size gas tanks. So, we need to modify our program to accommodate the different size tanks. To accommodate for different size tanks, we can pass the size of the tank to the constructor.

```

//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {

        public Truck(int tankSize)
        {
            Console.WriteLine("{0} gallons pumped in the truck", tankSize);
        }

        public int getNumOfWheels(int numberOfWheels = 4)
        {
            return numberOfWheels;
        }

        public void TruckBed(int length)
        {
            Console.WriteLine("Bed Length: " + length);
        }

    }
}

using System;

namespace ConsoleApp2
{
    class Car : Vehicle
    {
        public Car(int tankSize)
        {
            Console.WriteLine("{0} gallons pumped in the car", tankSize);
        }

        public void SetNumberOfPassengers(ref int numOfPassengers)
        {
            numOfPassengers = 4;
        }

        public void carChar(string make, int numberOfDoors = 4, string transmission = "Automatic")
        {
            Console.WriteLine("Number of doors: {0}", numberOfDoors);
            Console.WriteLine("Transmission type: {0}", transmission);
        }
    }
}

```

```

        Console.WriteLine("Car make: {0}", make);
        Console.WriteLine("-----");
    }

    public void rev()
    {

        for(int i = 0; i < 3; ++i)
        {
            Console.WriteLine("rev");
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Truck status");
            Truck Chevy = new Truck(25);
            Console.WriteLine("-----");
            Console.WriteLine("Car status");
            Car Mustang = new Car(16);
        }
    }
}

```

```

Truck status
filled the tank
25 gallons pumped in the truck
-----
Car status
filled the tank
16 gallons pumped in the car

```

As can be seen in the main program when we pass values into the constructor you do it when you create the object. Essentially, the code that comes after the new keyword is what calls the constructor. Any arguments that you need to pass go into those parentheses.

## Properties: The getter and setters rolled into one

Variables should never be directly manipulated from outside the class they are declared in. In fact, it is usually a good rule of thumb to declare all variables private. If you're a Java dork you're hopefully familiar with getter and setter methods that handle variable manipulation. C# provides us with a shorthand way of creating a getter and a setter method with properties. A property in itself is technically not a method. A property is better thought of as getter and setter methods along with variables wrapped into one.

The method like nature of properties gives us the ability to set logic that can filter out unwanted changes to variables and allows us to read private variables. Though properties are the correct way to access variables in another class you still have to exercise caution because even though there is a level of protection when setting values, properties can still be accessed from anywhere in the program and as such can be accidentally altered.

To demonstrate properties let's modify our vehicle program to read the miles per gallon of our vehicles and display the gas mileage. To do this we're going to modify the Car and Vehicle classes with a get property that manipulates the gas mileage that we passed in when we create the objects. We're also going to modify the Main method to access the properties like the following,

```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        private int TruckTank;

        public int truckTank //Property
        {
            get { return TruckTank; }
            set { TruckTank = value; }
        }

        public Truck(int tankSize)
        {
            TruckTank = tankSize;
        }
    }
}
```

```

    }

    public int getNumOfWheels(int numberOfWheels = 4)
    {
        return numberOfWheels;
    }

    public void TruckBed(int length)
    {
        Console.WriteLine("Bed Length: " + length);
    }

}

//car
using System;

namespace ConsoleApp2
{
    class Car : Vehicle
    {
        private int CarTank;

        public int carTank //Property
        {
            get { return CarTank; }
            set { CarTank = value; }
        }
        public Car(int tankSize)
        {
            CarTank = tankSize;
        }

        public void SetNumberOfPassengers(ref int numOfPassengers)
        {
            numOfPassengers = 4;
        }

        public void carChar(string make, int numberOfDoors = 4, string transmission = "Automatic")
        {
            Console.WriteLine("Number of doors: {0}", numberOfDoors);
            Console.WriteLine("Transmission type: {0}", transmission);
            Console.WriteLine("Car make: {0}", make);
            Console.WriteLine("-----");
        }

        public void rev()
    }
}

```

```

    {
        for(int i = 0; i < 3; ++i)
        {
            Console.WriteLine("rev");
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car Mustang = new Car(16);
            Truck Chevy = new Truck(25);
            Console.WriteLine("Mustang tank size: {0} gallons", Mustang.carTank);
            Console.WriteLine("Chevy tank size: {0} gallons", Chevy.truckTank);
        }
    }
}

```

```

filled the tank
filled the tank
Mustang tank size: 16 gallons
Chevy tank size: 25 gallons

```

Properties are traditionally declared with the pattern in the code sections marked property in the Car and Vehicle classes. Traditionally, you create a private variable and use the property patterns in the class to read the variable. However, C# offers a shortcut syntax with less code. In C# all you have to do is declare the property like the following,

```

public int carTank { get; set; }
public int truckTank { get; set; }

```

To demonstrate the shorthand syntax, comment out the properties and the variables that we included in the previous example. Then add the above two lines and after you run the program you should get the following output.

```
filled the tank  
filled the tank  
Mustang tank size: 16 gallons  
Chevy tank size: 25 gallons
```

Time for a little bit of opinion. When you declare a property with just the get and set, you're making an accessible variable. Even though many will consider this a better alternative to a public variable, in practice you have the same thing. So, be very careful, in practice a property without any logic is just a variable.

As I'm sure you've figured out now that the get and set keywords represent the actual getter and setter methods. So, let's try adding some getting and setting logic. To start with let's assume that our gas tanks vary, and for a car, anything over 20 gallons is too big and anything under 10 gallons is too small. If any of the tank sizes are under or over display a message in the console and assign a default value of 0 for the tank size. To do this we could modify the properties in the Car and Truck class like the following:

```
//Truck  
using System;  
  
namespace ConsoleApp2  
{  
    class Truck : Vehicle  
    {  
  
        private int TruckTank;  
        public int truckTank  
        {  
            set  
            {  
                TruckTank = value;  
            }  
  
            get  
            {  
                if (TruckTank < 20 || TruckTank > 40)
```



```

        {
            Console.WriteLine("Truck tank out of range!");
            return 0;
        }
        else
        {
            return TruckTank;
        }
    }
}

    public Truck(int tankSize)
    {
        truckTank = tankSize;
    }

    public int getNumOfWheels(int numberOfWheels = 4)
    {
        return numberOfWheels;
    }

    public void TruckBed(int length)
    {
        Console.WriteLine("Bed Length: " + length);
    }

}

}

//car
using System;

namespace ConsoleApp2
{
    class Car : Vehicle
    {
        private int CarTank;

        public int carTank
        {
            set
            {
                CarTank = value;
            }
            get
            {

```

```

        if (CarTank < 10 || CarTank > 20)
        {
            Console.WriteLine("Car gas tank out of range!");
            return 0;
        }
        else
        {
            return CarTank;
        }
    }
}

public Car(int tankSize)
{
    carTank = tankSize;
}

public void SetNumberOfPassengers(ref int numOfPassengers)
{
    numOfPassengers = 4;
}

public void carChar(string make, int numberOfDoors = 4, string transmission = "Automatic")
{
    Console.WriteLine("Number of doors: {0}", numberOfDoors);
    Console.WriteLine("Transmission type: {0}", transmission);
    Console.WriteLine("Car make: {0}", make);
    Console.WriteLine("-----");
}

public void rev()
{
    for(int i = 0; i < 3; ++i)
    {
        Console.WriteLine("rev");
    }
}
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)

```

```

{
    //throw error
    Car Mustang = new Car(5);
    Truck Chevy = new Truck(10);

    Console.WriteLine("Mustang tank size: {0} gallons", Mustang.carTank);
    Console.WriteLine("Chevy tank size: {0} gallons", Chevy.truckTank);

}
}

```

```

filled the tank
filled the tank
Car gas tank out of range!
Mustang tank size: 0 gallons
Truck tank out of range!
Chevy tank size: 0 gallons

```

So, for this example we passed in the values to the constructor then the value was assigned to a private variable, from there the property did the rest. The setter method set the value that passed in and the getter method performed a check to determine which value to return. Now, for some homework! Play with the values that were passed in for the car and truck tank. In the above example, we set the values to throw the message if the tank was too small. What happens when the values are too big? Or, what happens when the tank is the proper size and the other value is out of range?

The last example was manipulating what was read from the property by adding logic to the getter method. What happens when we try adding logic to the setter method? Let's assume our drunk boss gets drunk one day and wants to double the size of every tank for every car and every truck. After trying unsuccessfully to convince him to go to an AA meeting we comply. After pondering our life choices and how we're going to do this we decided to use our property's setter method to multiply all values and multiply them by 2. So, after some typing, we come up with the following code,

```

//Truck
using System;

namespace ConsoleApp2

```

```

{
    class Truck : Vehicle
    {

        private int TruckTank;
        public int truckTank
        {
            set
            {
                TruckTank = value * 2;
            }

            get
            {
                if (TruckTank < 20 || TruckTank > 40)
                {
                    Console.WriteLine("Truck tank out of range!");
                    return 0;
                }
                else
                {
                    return TruckTank;
                }
            }
        }

        public Truck(int tankSize)
        {
            truckTank = tankSize;
        }

        public int getNumOfWheels(int numberOfWheels = 4)
        {
            return numberOfWheels;
        }

        public void TruckBed(int length)
        {
            Console.WriteLine("Bed Length: " + length);
        }

    }
}

//car
using System;

```

```
namespace ConsoleApp2
{
    class Car : Vehicle
    {
        private int CarTank;

        public int carTank
        {
            set
            {
                CarTank = value * 2;
            }
            get
            {
                if (CarTank < 10 || CarTank > 20)
                {
                    Console.WriteLine("Car gas tank out of range!");
                    return 0;
                }
                else
                {
                    return CarTank;
                }
            }
        }
        public Car(int tankSize)
        {
            carTank = tankSize;
        }

        public void SetNumberOfPassengers(ref int numOfPassengers)
        {
            numOfPassengers = 4;
        }

        public void carChar(string make, int numberOfDoors = 4, string transmission = "Automatic")
        {
            Console.WriteLine("Number of doors: {0}", numberOfDoors);
            Console.WriteLine("Transmission type: {0}", transmission);
            Console.WriteLine("Car make: {0}", make);
            Console.WriteLine("-----");
        }

        public void rev()
        {
            for(int i = 0; i < 3; ++i)
            {
                Console.WriteLine("rev");
            }
        }
    }
}
```

```

    }
}

}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            //throw error
            Car Mustang = new Car(10);
            Truck Chevy = new Truck(15);

            Console.WriteLine("Mustang tank size: {0} gallons", Mustang.carTank);
            Console.WriteLine("Chevy tank size: {0} gallons", Chevy.truckTank);

        }
    }
}

```

```

filled the tank
filled the tank
Mustang tank size: 20 gallons
Chevy tank size: 30 gallons

```

As can be seen, we passed in 10 for the car's tank size and 15 for the trucks tank size and we got 20 and 30 respectively. If you notice the properties in both classes, we took the value and multiplied it by 2.

The moral of this story is the set methods assign a value to the property and the logic there should be relegated to just that, the logic that assigns values. The get method should only handle the retrieval of the assigned values. It is very tempting to mix and match these responsibilities. However, for clean code just remember that if the logic is for reading a value it goes in the get method and if it is used for setting a value the logic goes in the set method.

Properties don't need both a get and set method. You can set a

property to only read or only write. In practice, it is more common to make a property read-only as opposed to write-only. If you're a Java dork this is going to look very familiar as it resembles true getter and setter methods. A read-only property will only contain the get method while a write-only will contain only the set method. To see the read-only property in action let's create a new class called properties and add a private string variable called bossName and a property called GetBossName like so,

```
//properties
namespace ConsoleApp2
{
    class properties
    {
        private string bossName = "drunk boss";

        public string GetBossName
        {
            get { return bossName; }
        }
    }
}
```

To use this new class, we're going to modify the Main method like so,

```
//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {

            properties boss = new properties();
            Console.WriteLine("Bosses Name: {0}", boss.GetBossName);

        }
    }
}
```

Run the program and we should get the following output,

```
Bosses Name: drunk boss
```

Alright, the drunk boss looks accurate to me!

Now, we can turn our attention to setter methods. Consider the following,

```
//properties
using System;

namespace ConsoleApp2
{
    class properties
    {
        private string bossName = "drunk boss";

        public string GetBossName
        {
            get { return bossName; }
        }

        public string setBossName
        {
            set { bossName = value; }
        }

        public void DisplayBossName()
        {
            Console.WriteLine("Boss name now: {0}", bossName);
        }
    }
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            properties boss = new properties();
            boss.setBossName = "drunk stoned boss";
            boss.DisplayBossName();
        }
    }
}
```



```
Boss name now: drunk stoned boss
```

This time we added `setBossName` which is the write-only property, pretty straightforward. We also added a simple `DisplayBossName` method that does nothing but shows the change in the boss's name. To assign the new name in the `Main` method we assigned a string to the property as if it were a simple class variable, and when we ran the program, we got the expected results. All things considered pretty simple.

Now, that we have an idea of read and write-only properties let's deduce some rules. First, and this is an opinion, believe it or not, I'm also a dreaded Java dork, and as such I like to name my read and write-only properties with the prefix `get` and `set`. I feel it makes the code more maintainable in the long run. You technically don't have to do that, but I recommend that you do it, your colleges and stoned bosses will thank you. Second, in our example, we wrote a method to retrieve the changes the setter made, in real-life you may not have to do this and if you do you might as well just create a regular property. A good way to conceptualize read and write only properties is with the following:

- Setter Method = Data going into the class
- Getter Method = Data coming out of the class

Our read and write-only examples are pretty bare. However, just like with traditional properties logic can be added to read and write-only properties.

Properties, much like regular methods, can be virtual. This means that we can create a virtual property in a base class and override the `get` and `set` logic in the child class. This is a very handy feature when there are different requirements for a given property across the child classes. Virtual properties are declared and overridden just like methods are, with the `virtual` and `override` keywords respectively. To see how virtual properties work let's look at the following example,

```
//Vehicle
using System;

namespace ConsoleApp2
{
```

```

class Vehicle
{
    private int _numberOfCylinders;
    public virtual int numberOfCylinders { get; set; }

    //constructor
    public Vehicle()
    {
        Console.WriteLine("filled the tank");
    }

    public void engine()
    {
        Console.WriteLine("vroom");
    }

    public void breaks()
    {
        Console.WriteLine("stop");
    }

    public void transmission(int rpms)
    {
        Console.WriteLine("RPMs: " + rpms);
    }
}

//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        private int TruckTank;
        private int _numberOfCylinders; //property variable

        //virtual property
        public override int numberOfCylinders
        {
            set
            {
                _numberOfCylinders = value;
            }

            get

```

```

    {
        if (_numberOfCylinders < 6)
        {
            Console.WriteLine("invalid number of cylinders");
            return 0;
        }
        else
        {
            return _numberOfCylinders;
        }
    }
}

public int truckTank
{
    set
    {
        TruckTank = value * 2;
    }

    get
    {
        if (TruckTank < 20 || TruckTank > 40)
        {
            Console.WriteLine("Truck tank out of range!");
            return 0;
        }
        else
        {
            return TruckTank;
        }
    }
}

    public Truck(int tankSize)
    {
        truckTank = tankSize;
    }

    public int getNumOfWheels(int numberOfWheels = 4)
    {
        return numberOfWheels;
    }

    public void TruckBed(int length)
    {
        Console.WriteLine("Bed Length: " + length);
    }
}

```

```

    }
}

//car
using System;

namespace ConsoleApp2
{
    class Car : Vehicle
    {
        private int CarTank;
        private int _numberOfCylinders;//property variable

        public override int numberOfCylinders
        {
            set
            {
                _numberOfCylinders = value;
            }

            get
            {
                if(_numberOfCylinders > 4)
                {
                    Console.WriteLine("invalid number of cylinders");
                    return 0;
                }
                else
                {
                    return _numberOfCylinders;
                }
            }
        }
    }

    public int carTank
    {
        set
        {
            CarTank = value * 2;
        }
        get
        {
            if (CarTank < 10 || CarTank > 20)
            {
                Console.WriteLine("Car gas tank out of range!");
                return 0;
            }
        }
    }
}

```

```

        else
        {
            return CarTank;
        }
    }
}

public Car(int tankSize)
{
    carTank = tankSize;
}

public void SetNumberOfPassengers(ref int numOfPassengers)
{
    numOfPassengers = 4;
}

public void carChar(string make, int numberOfDoors = 4, string transmission = "Automatic")
{
    Console.WriteLine("Number of doors: {0}", numberOfDoors);
    Console.WriteLine("Transmission type: {0}", transmission);
    Console.WriteLine("Car make: {0}", make);
    Console.WriteLine("-----");
}

public void rev()
{
    for(int i = 0; i < 3; ++i)
    {
        Console.WriteLine("rev");
    }
}
}

//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Car mustang = new Car(13);
            Truck ford = new Truck(20);

```

```

        Console.WriteLine("Valid cylinders for a truck: {0}", ford.numberOfCylinders = 8);
        Console.WriteLine("Valid cylinders for a car: {0}", mustang.numberOfCylinders = 6);
        Console.WriteLine("-----");
        Console.WriteLine("Invalid cylinders for a truck: {0}", ford.numberOfCylinders = 4);
        Console.WriteLine("Invalid cylinders for a car: {0}", mustang.numberOfCylinders = 2);
    }
}

```

```

filled the tank
filled the tank
Valid cylinders for a truck: 8
Valid cylinders for a car: 6
-----
Invalid cylinders for a truck: 4
Invalid cylinders for a car: 2

```

So, in this case, we declared a virtual property in the vehicle class which is the base class for the car and truck classes. Both the car and truck override the property and have their implementation of the getter logic. In short, if a truck has less than 6 cylinders the program returns 0 and displays a message where if a car has less than 4 cylinders the program returns 0 and displays a message.

## Expression Bodied Members and the => operator

Suppose you need a simple method that returns the product of two numbers like the following,

```

public static int product(int x, int y)
{
    return x * y;
}

```

When you think about it for a second this is a lot of code for such a simple method. You have to use a set of curly braces, the return keyword, and all the declaration code. Luckily, if you're using C# 6 or above you can use what is called Expression Bodied members. The basic syntax for an Expression Bodied member is as follows,

```

member => expression

```

So, if we were to use Expression Bodied member for the product method, we would use the following,

```
public static int product(int x, int y) => x * y;
```

Essentially, the method is declared the same way as a normal method; however, there are no curly braces or return statements.

Expression Bodied members are a shorthand way of writing members. Expression Bodied members were introduced in C# 6 and were greatly expanded upon in C#7. Expression Bodied members can be used with the following,

- Methods
- Constructors
- Destructors
- Property setter
- Property getter
- Indexers

In other words, if has a logical implementation you can use Expression Bodies.

Expression Bodies members are syntax sugar to simplify simple expressions. You want to use these expressions to make code more readable and easier to understand. If you're doing something simple that can fit one line such as computing a math equation, performing simple operations on strings, and so on. If the logic is complicated and can't fit on one line avoid using these expressions; however, I strongly recommend using Expression Bodied members if you're logic is simple enough to logically fit on one line.

### **What goes in a method, how long should it be?**

Many newbies don't know how long a method should be or what should go into a method. I have seen methods that have one check and call another method and I have seen methods that are hundreds of lines long that do everything under the sun. So, with all that being said how long should the method be? The answer to that is it varies, yes, a hundred-line method is acceptable if it is well written. A better question to focus on is what a

method should do.

In short, a method should do one thing and one thing only. A while back I came across a rule of thumb for methods that says a method should be definable in one sentence without the word and. If the word and appears in the method's definition you need to turn everything after the word and into a method of its own. Consider the following pseudo-code,

```
//this method sets the buttons color and text  
method format button(btnColor, btnText)  
    turn button color btnColor;  
    set button message to btnText;
```

Changing a button's text and color at the same time is an excellent method. It does everything we need. Now, let's assume that after an all-night bender our boss just wants to add a new button and that button's text never changes only the color. This kind of leaves us in a pickle because we already have a method that changes the color and the text so at best half the code is going to be repeated. That's no good we're programmers we don't repeat ourselves. A better solution would be as follows,

```
//this method sets the buttons text  
method format buttonText(btnText)  
    set button message to btnText;  
  
//this method set the button color  
method format buttonColor(btnColor)  
    set button color to btnColor;
```

By breaking, the first method into two smaller methods that can be defined without the word and we free ourselves up and our program becomes more robust. Now, if we just want to just change the color we call the buttonColor method, on the other hand, if we want to just change the text we just call the buttonText method, and if we want to change both the color and the text we call both methods.

This sentence trick is more or less a rule of thumb. Depending on how you word the sentence can justify a larger method that should be broken into smaller methods or you could end up abstracting a method so much you might end up with 40 tiny methods that should be rolled into one larger



method. As I said this is a rule of thumb, not a hard-set principle. To use this technique effectively you have to be fair with yourself and your wording. Don't just use wording tricks as a means to justify having a 10,000 line method or having 10,000 methods that only perform an if check. Use your best judgment and this as a guiding principle.

# CHAPTER 3: CLASSES

## PART 2 AND OTHER WEIRD STUFF

Thus far we looked at classes, the four pillars of object-oriented programming, and a bunch of other stuff. You're on your way to becoming a cool programmer, by that I means an employed programmer. However, C# is that gift that keeps on giving. There's way more to C# than just classes. Oh no, Alice, this rabbit hole is much, much deeper. We're about to dive into a magical world of classes that are designed only to be base classes, organization containers for a large project, and much, much more.

### Structs

If you're one of those dorky Java programmers or a newbie there is another class like structure that you probably never heard of called a struct. Structs are the older, dumber brothers of classes. Structs are an old data type that originated from the C programming language many, many eons ago, like the 1970s. In terms of C#, structs are a lot like classes. In terms of C#, a struct is almost like a class; however, there are differences. In practice, the biggies that you'll more or less notice are as follows:

- Structs can't use inheritance
- You cannot initiate an array
- Structs do not support constructors without parameters
- You create an instance of a struct without the new keyword
- Members cannot be virtual or abstract

Though structs have some functionality of class; traditionally, structs are used to declare variables of different types that are related. Usually, the rule of thumb is structs are only used in smaller programs. I disagree with that rule and have used structs in very large industrial programs with great effect.

However, I will say that whatever a struct can do a class can do better, but like everything else, a struct is a tool and has its place in C#.

Let's assume that our boss wants to modify the program so that it keeps track of the miles per gallon, the miles on the vehicle, and the vehicle's value. We could create a class to handle the data and inherit it. This is a possible solution. Another solution would be to use a struct like the following,

```
struct vehicleData
{
    public double MPG;
    public int miles;
    public double value;
}
```

Now, remember that to initiate the struct you do not have to use the new keyword. So, with that the following syntax can be used,

```
vehicleData MustangData;
vehicleData ChevyData;
```

The above snippet is basically the same thing as creating class objects. We use the struct variables to access the attributes the same way we would in a class. Consider the following,

```
//main
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            //reference variable
            vehicleData MustangData;

            //assign values
            MustangData.miles = 3;
            MustangData.MPG = 26.5;
            MustangData.value = 45000;
```

```

//display data
Console.WriteLine("Mustang miles: {0}", MustangData.miles);
Console.WriteLine("Mustang MPG: {0}", MustangData.MPG);
Console.WriteLine("Mustang value: {0}", MustangData.value);

    }
}

```

```

Mustang miles: 3
Mustang MPG: 26.5
Mustang value: 45000

```

As can be seen, in terms of accessing and retrieving attributes are almost the same as with classes.

Structs can be declared either in a class or in a namespace. I've seen many developers opt to embed structs in classes. By embedding the struct the struct is encapsulated in the class and becomes a class attributes and is accessed with the following syntax,

```
<object_variable>.<struct_variable>.<attribute>
```

As I stated before you can also declare a struct in a namespace much the same way you declare a class. To do this in Visual Studios you want to create an empty code file and include the following code,

```

using System;

namespace ConsoleApp2
{
    struct vehicleData
    {
        public double MPG;
        public int miles;
        public double value;
    }
}

```

This is the way the struct was created for the example. Technically you don't have to declare the struct in its own code file and you don't have to declare it in a class either. I've seen structs declared inside of class files but outside of classes. This to me can lead to issues and I have seen issues arise for this, mainly the struct can be hard to find and it can clutter the class files. My

advice is if you declare a struct outside of a class it is usually best to declare them in a code file.

## **Abstract Classes**

Suppose this, you're a programmer. You're typing away on your keyboard looking like a super spy, you're typing, and typing, and smiling. The testers walk by and look at you with envy in their eyes wishing they could be you. Then bam, your world shatters into a thousand pieces. You have multiple classes that all need a method with the same name but with different functionalities. Then you realize that the method is so unique to each of the child classes that each of the classes needs its own implementation of the method. Then you realize that the base class you're working on is never instantiated. At this point, testers start laughing at you, the room starts spinning, reality breaks apart, and the great god Cthulhu appears wearing an I love software testers shirt. What do you do? How do you fix reality and logic? Do you just surrender to Cthulhu or worse do you implement the method and override it as a tester would? Or maybe do you use an abstract class or interface? I would personally use an abstract class or interface depending on the situation.

So, what is an abstract class? An abstract class is a special type of class that supports methods that are declared but not implemented. Abstract classes are meant to be base classes and as such, they cannot be instantiated on their own, and for the methods to be consumed the abstract class must be inherited. In short, an abstract class is useless on its own as it is meant as a base class for another class.

The best way to think of an abstract class is as a framework class. Abstract classes generally have a mix of implemented and non-implemented methods. For the most part the implemented methods are defined enough to be used by any class that inherits the abstract class and the non-implemented methods are usually so generic that each class that derives from the abstract class will each need its implementation of the method.

Our vehicle class from our earlier examples is a prime candidate for an abstract class. Our program will never need to instantiate the vehicle class and when we get down to it the only thing our Vehicle class does is act as a framework class for our Car class and our Truck class. So, let's start over and gut everything out of the Vehicle class and turn it into an abstract class

while we're at it let's also start over with the Truck and Car class. When you're done gutting those classes let's modify the engine method to be an abstract method like the following:

```
namespace ConsoleApp2
{
    abstract class Vehicle
    {
        public abstract void engine();

        public void breaks()
        {
            Console.WriteLine("stop");
        }

        public void transmission(int rpms)
        {
            Console.WriteLine("RPMs: " + rpms);
        }
    }
}
```

Notice in the class declaration line that the keyword `abstract` comes before the keyword `class`. This tells C# that this is an abstract class, can support methods, and cannot be instantiated. Now, let's look at the engine method. As can be seen right after our access specifier is the keyword `abstract`. This tells C# that this is an abstract method, it currently has no implementation, and it must be defined in the inherited classes. Also, notice that abstract methods end in a semicolon. The semicolon is a little gotcha, that can bite you in the long run so be careful not to forget it.

Now, let's prep our Truck class. To make things easy on ourselves let's go ahead and gut the old code from the previous examples out and modify the class to look like the following:

```
//Truck
using System;

namespace ConsoleApp2
{
    class Truck : Vehicle
    {
        public override void engine()
```

```
{  
    Console.WriteLine("rumble");  
}  
}
```

Notice we have to declare the engine method with the override keyword like we did when we were overriding virtual methods. In essence, we are doing the same thing here, so even if a method is abstract, we still have to use the override keyword. Finally, for the Truck class, the engine method is implemented to write “rumble” to the console.

Moving on to the car class, we’re also going to gut the old code out and modify it to match the following:

```
//car  
using System;  
  
namespace ConsoleApp2  
{  
    class Car : Vehicle  
    {  
        public override void engine()  
        {  
            Console.WriteLine("vroom");  
        }  
    }  
}
```

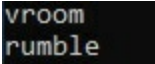
Pretty much the same syntax as before; however, this time the engine method is implemented to say “vroom”.

Finally, the main file should be modified to be like the following:

```
//main  
using System;  
  
namespace ConsoleApp2  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Car mustang = new Car();  
            Truck chevy = new Truck();  
        }  
    }  
}
```

```
        mustang.engine();
        chevy.engine();
    }
}
```

After running the program, the following should be outputted to the console.



```
vroom
rumble
```

As can be seen, the mustang, which is a car object, says vroom while the chevy, which is a truck object says rumble.

There are catch 22s when it comes to abstract methods. An abstract method has to be implemented in every derived class. A program will not compile if there are unimplemented abstract methods in the child class(es). If you're using Visual Studios a red squiggle will show under the child class's name. If you right-click the squiggle and select Quick Actions and Refactoring you will be met with an option to automatically implement the abstract method(s). It's a handy shortcut, and it's one that I use all the time.

### Helper Classes: The good, the bad, and the ugly

Not matter what object-oriented language you are using you can create what is called a helper class. Helper classes, outside of being the bane of existence for many developers, are little classes that for lack of a better word everyone dumps their junk in. Being a little more specific helper classes are regular, static classes that contain methods and other attributes that are used across multiple classes but don't fit well into any of the classes. Generally, helper classes are usually labeled as utility classes and are used to augment code.

Consider the following two classes,

```
//cat
using System;

namespace ConsoleApp2
{
    class Cat
    {
        public void displayString(string message)
```



```

        {
            Console.WriteLine(message);
        }
    }
}

//spaceship
using System;

namespace ConsoleApp2
{
    class spaceship
    {
        public void displayString(string message)
        {
            Console.WriteLine(message);
        }
    }
}

```

As you can see both classes have two methods that do the same thing. This is technically not the end of the world, but suppose the boss gets drunk again and wants the message displayed three times. To implement the change, we would have to change both methods. That's kind of okay when there are only two methods and the program is small but what about large programs that have the same method across multiple classes? That's not going to be an easy change. Besides we're programmers we don't repeat ourselves. We could think about using inheritance but there isn't an is-a relationship between a cat and a spaceship. We could also think about using an interface but there's no common model between a cat and a spaceship. We could also put the displayString method in its own class and use composition but would there really be that strong of a has-a relationship between a displayString and a cat or spaceship? This is where things get subjective quick. In my opinion, considering we have a cat and a spaceship we're always going to have a very weak has a relationship and though we could get away with using composition it's going to be more or less forced. As such, I would personally create a helper class like the following,

```

using System;

namespace ConsoleApp2
{
    static class utility

```

```
{  
    public static void displayString(string message)  
    {  
        Console.WriteLine(message);  
    }  
}  
}
```

Helper classes have a bad rap. Many developers consider them evil anti-patterns because they often get bloated with helper methods, they break OOP principles such as the single responsibility principal, they don't follow the blueprint rule for classes, they are impossible to test, are generally considered more procedural, and the list goes on and on. Regardless, most programs of any significant size will usually have helper classes. Helpers are usually considered sloppy programming and should for the most part be avoided.

So, is it ever okay to use a helper class? The answer is maybe. I have personally used helper classes to great success. When used properly, and when you get your team to agree to avoid using helper classes as a dumping pit, helper classes are okay to use. Generally, when I consider writing a helper class is when I start seeing the same method popping up in multiple classes maybe even multiple projects, and the methods don't fit any of the classes well. I also write them when I'm trying to develop specialized libraries for things like robots that require a lot of special math. When using helper classes, you don't want the class to hold any data, in fact, you want the methods to perform their function and end. If you're not writing a library or class to augment existing code, your best bet is to go back to the drawing board and try to use composition or inheritance.

## Interfaces

Let's assume our boss now owns a chop shop, he needs more employees besides us and the testers, not that the testers count. Just our luck our boss has added three new types of jobs to his roster and now he needs a whole new program that can keep track of who's doing what.

As we're designing our new software, we realize that all employees have certain attributes that are the same. All the employees have a job duty, have to come into work at a certain time, have to get paid, and have to commit crimes; however, since every job is different all of these duties

change from job to job. So, as we're designing we figure we could use an abstract class, but for what we're doing that's overkill. Instead a better option is that we use an interface.

Interfaces are a lot like abstract classes; however, an abstract class is a class and an interface is an interface. By definition interfaces are contracts. Interfaces do not allow you to implement methods only declare them, all the methods are virtual, you cannot instantiate an interface, and classes do not inherit interfaces. At first glance interfaces may seem a bit useless; however, they help use model things. When you use an interface, you're telling C# "I'm going to use this interface to model this thing and in return, I promise to implement everything that isn't implemented."

Interfaces are handy with problems like the one we're experiencing in our chop shop employee program. We have the basic blueprint of what an employee should be but since every job is different, we don't know how those job duties are going to be implemented in the program. As such, we can use the interface to establish what the outline can be and we can fill in the blanks when the interface is consumed. So, let's look at what the basic syntax for an interface is,

```
interface <Iname>
{
    //code
}
```

Interfaces do have a naming convention that you technically don't have to follow but is a good idea to follow if you do. It is considered a good idea to start the interface's name with a capital I. The I signals to another developer that they're working with an interface and that ultimately they're dealing with a model of something.

Now that we have an idea of what an interface looks like, let's look to see how interfaces are implemented,

```
class <name> : <Interface_name>
{
    //code
}
```

As can be seen in the example, the same syntax that is used for inheritance is

also used to implement interfaces. If you use Visual Studios a red line will appear under the interface's name. This is normal and it means that the interface's methods are not implemented. The methods can be automatically implemented by right-clicking the interface's name and click Quick Action and Auto Refactorings and then selecting the implement interface option. After you do that your code should look like the following,

```
//employee Interface
namespace chopShop
{
    interface Iemployee
    {
        public void getPaid();
        public void ArrivalTime();
        public void crimes();
    }
}

//BigBoss
using System;

namespace chopShop
{
    class BigBoss : Iemployee
    {
        public void ArrivalTime()
        {
            throw new NotImplementedException();
        }

        public void crimes()
        {
            throw new NotImplementedException();
        }

        public void getPaid()
        {
            throw new NotImplementedException();
        }
    }
}

//Programmer
using System;

namespace chopShop
{
```

```

class programmer : Iemployee
{
    public void ArrivalTime()
    {
        throw new NotImplementedException();
    }

    public void crimes()
    {
        throw new NotImplementedException();
    }

    public void getPaid()
    {
        throw new NotImplementedException();
    }
}

//ShopHand
using System;

namespace chopShop
{
    class ShopHand : Iemployee
    {
        void Iemployee.ArrivalTime()
        {
            //throw new NotImplementedException();
        }

        void Iemployee.crimes()
        {
            throw new NotImplementedException();
        }

        void Iemployee.getPaid()
        {
            throw new NotImplementedException();
        }
    }
}

//ShopManager
using System;

namespace chopShop
{
    class ShopManager : Iemployee
    {

```

```

    public void ArrivalTime()
    {
        throw new NotImplementedException();
    }

    public void crimes()
    {
        throw new NotImplementedException();
    }

    public void getPaid()
    {
        throw new NotImplementedException();
    }
}

//Tester
using System;

namespace chopShop
{
    class Tester : Iemployee
    {
        public void ArrivalTime()
        {
            throw new NotImplementedException();
        }

        public void crimes()
        {
            throw new NotImplementedException();
        }

        public void getPaid()
        {
            throw new NotImplementedException();
        }
    }
}

```

Notice that in the interface we did not use the virtual keyword, nor did we use the override keyword in the classes that implement the interface. This goes back to all methods in an interface being virtual. When a method is declared C# just assumes that it is virtual and as such, we don't have to mark it as virtual. By the same extension when we implement methods that are derived from an interface C# assumes we are overriding them and we don't have to

use the override keyword. Also, when you auto implement the interface's methods C# will include this line of code,

```
throw new NotImplementedException();
```

You don't need that line of code for everything to work and it can be removed.

Now, we can start filling in the methods for each class. For our program, we're only going to modify the class methods to display text to the console. So, we're going to modify our classes like so,

```
//BigBoss
using System;

namespace chopShop
{
    class BigBoss : Iemployee
    {
        public void ArrivalTime()
        {
            Console.WriteLine("3pm");
        }

        public void crimes()
        {
            Console.WriteLine("DUI");
            Console.WriteLine("Assault and Battery");
            Console.WriteLine("Drug Usage");
            Console.WriteLine("Terrorism");
            Console.WriteLine("Illegal Gambling");
            Console.WriteLine("Resisting Arrest");
            Console.WriteLine("Back Child Support");
        }

        public void getPaid()
        {
            Console.WriteLine("Every day");
        }
    }
}

//Programmer
using System;

namespace chopShop
{
    class programmer : Iemployee
    {
```

```

        public void ArrivalTime()
        {
            Console.WriteLine("9am");
        }

        public void crimes()
        {
            Console.WriteLine("hacking");
        }

        public void getPaid()
        {
            Console.WriteLine("salary");
        }
    }
}

//ShopHand
using System;

namespace chopShop
{
    class ShopHand : Iemployee
    {
        void Iemployee.ArrivalTime()
        {
            Console.WriteLine("5am");
        }

        void Iemployee.crimes()
        {
            Console.WriteLine("Car chopping");
        }

        void Iemployee.getPaid()
        {
            Console.WriteLine("hourly");
        }
    }
}

//ShopManager
using System;

namespace chopShop
{
    class ShopManager : Iemployee
    {
        public void ArrivalTime()
        {
            Console.WriteLine("6am");
        }
    }
}

```



```

    }

    public void crimes()
    {
        Console.WriteLine("car theft");
    }

    public void getPaid()
    {
        Console.WriteLine("Salary");
    }
}
}

//Tester
using System;

namespace chopShop
{
    class Tester : Iemployee
    {
        public void ArrivalTime()
        {
            Console.WriteLine("3am");
        }

        public void crimes()
        {
            Console.WriteLine("hacking");
        }

        public void getPaid()
        {
            Console.WriteLine("they pay us");
        }
    }
}

```

From this point it the same old object game. For example, suppose we want to retrieve the big boss's information, all we need to do is create an object and call the methods like so,

```

using System;

namespace chopShop
{
    class Program
    {

```

```

static void Main(string[] args)
{
    BigBoss boss = new BigBoss();

    Console.WriteLine("Boss's Arrival Time");
    boss.ArrivalTime();
    Console.WriteLine("-----");
    Console.WriteLine("Boss's crimes");
    boss.crimes();
    Console.WriteLine("-----");
    Console.WriteLine("Boss's pay");
    boss.getPaid();
    Console.WriteLine("-----");
}
}

```

```

Boss's Arrival Time
3pm
-----
Boss's crimes
DUI
Assault and Battery
Drug Usage
Terrorism
Illegal Gambling
Resisting Arrest
Back Child Support
-----
Boss's pay
Every day
-----

```

Since interfaces are not inherited from, a class can implement as many interfaces as needed, there is no limit. So, let's reexamine our bigBoss. A boss is not a regular employee. They are an employee and as such, they have all the attributes of an employee so we can partially model a boss after an employee. However, to properly model a boss you have to give them the ability to hire employees, fire employees, and be utterly useless. So, to add these extra attributes we're going to make another interface called IBoss and add these extra attributes like so,

```

//Boss interface
namespace chopShop
{
    interface IBoss
    {
        public void hireEmployees();
    }
}

```

```
        public void fireEmployees();
        public void BeUseless();
    }
}
//BigBoss
using System;

namespace chopShop
{
    class BigBoss : Iemployee, IBoss
    {
        public void ArrivalTime()
        {
            Console.WriteLine("3pm");
        }

        public void BeUseless()
        {
            Console.WriteLine("zzzzzzz");
        }

        public void crimes()
        {
            Console.WriteLine("DUI");
            Console.WriteLine("Assault and Battery");
            Console.WriteLine("Drug Usage");
            Console.WriteLine("Terrorism");
            Console.WriteLine("Illegal Gambling");
            Console.WriteLine("Resisting Arrest");
            Console.WriteLine("Back Child Support");
        }

        public void fireEmployees()
        {
            Console.WriteLine("You're fired!!!");
        }

        public void getPaid()
        {
            Console.WriteLine("Every day");
        }

        public void hireEmployees()
        {
            Console.WriteLine("You're hired");
        }
    }
}

//Main
using System;
```

```
namespace chopShop
{
    class Program
    {
        static void Main(string[] args)
        {
            BigBoss boss = new BigBoss();

            Console.WriteLine("Boss's Arrival Time");
            boss.ArrivalTime();
            Console.WriteLine("-----");
            Console.WriteLine("Boss's crimes");
            boss.crimes();
            Console.WriteLine("-----");
            Console.WriteLine("Boss's pay");
            boss.getPaid();
            Console.WriteLine("-----");
            Console.WriteLine("Boss is hiring");
            boss.hireEmployees();
            Console.WriteLine("-----");
            Console.WriteLine("Boss is firing");
            boss.fireEmployees();
            Console.WriteLine("-----");
            Console.WriteLine("Boss man doing what he does best");
            boss.BeUseless();
        }
    }
}
```

```
Boss's Arrival Time
3pm
-----
Boss's crimes
DUI
Assault and Battery
Drug Usage
Terrorism
Illegal Gambling
Resisting Arrest
Back Child Support
-----
Boss's pay
Every day
-----
Boss is hiring
You're hired
-----
Boss is firing
You're fired!!!
-----
Boss man doing what he does best
zzzzzzz
```

Okay, surprise it works. Now, pay attention to this line of code,

```
class BigBoss : Iemployee, IBoss
```

As can be seen, we have Iemployee and IBoss. We use the same syntax that we use to inherit but we are separating the interfaces with a comma. Essentially, you use a comma to separate the interfaces that you are implementing. You also use the same syntax when you are inheriting a class and implementing interfaces.

## Delegates

If you're a C++ programmer you've probably heard of function pointers. C# has a similar feature called a delegate. A delegate in C# can call any method that matches its return type and argument lists. In my experience I've seen delegate used mainly for two reasons, the first is to fire multiple methods at the same time which is known as a multicast delegate and the second is pass around methods like they are variables to other methods.

Delegates are created with the following syntax

```
<access_specifier> delegate <return_type> name(<args>);
```

So if we wanted to create a delegate called testDel() that can call any method with a return type of void and no arguments, we would use the following,

```
public delegate void testDel();
```

Now, that we can create a delegate let's see one in action in our chop shop program. For this example, let's make a delegate called bonus that has a return type of int and takes an int type called salary as a bonus. For simplicity, we are going to put everything in the Main method.

```
namespace chopShop
{
    public delegate int bonus(int salary);
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Now, let's assign a method for the delegate to call. To start let's create a method called employee that takes a value called salary and returns that value times 2. So, we would have the following,

```
namespace chopShop
{
    public delegate int bonus(double salary);
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

```
    public static int employeeBonus(int salary) => salary * 2;
}
}
```

Pretty straightforward, right? Now, we can assign the employeeBonus method to the delegate by creating an object to the delegate like so,

```
namespace chopShop
{
    public delegate int bonus(int salary);
    class Program
    {
        static void Main(string[] args)
        {
            bonus b = new bonus(employeeBonus);

        }

        public static int employeeBonus(int salary) => salary * 2;
    }
}
```

As you can see when we create the object variable, we pass in the name of the method we want to assign. To invoke the method, we use the following syntax,

```
b(argument_value);
```

So, if we wanted to pass in 2000 for the salary we would use,

```
b(2000);
```

After putting everything together we would get the following,

```
using System;

namespace chopShop
{
    public delegate int bonus(int salary);
```

```

class Program
{

    static void Main(string[] args)
    {

        bonus b = new bonus(employeeBonus);
        Console.WriteLine(b(2000));

    }

    public static int employeeBonus(int salary) => salary * 2;
}

```

Which in turn would yield an output of,

4000

Okay, okay so this isn't the coolest thing in the world, and in fact, it's a bit of a convoluted way of calling a method. So, to spice things up a bit let's take a look at Multicast delegates.

A multicast delegate is a delegate that references more than one method when it is invoked. Multicast delegates call multiple methods when the delegate is invoked. There is a little gotcha when dealing with multicast delegates. If you have multiple methods that return a value all the return values except the last value will be ignored. We'll explore this more in a bit but for now, let's change the delegates and method to have a void return type and display the calculated. As such modify your program to match the following,

```

using System;

namespace chopShop
{
    public delegate void bonus(int salary);
    class Program
    {

        static void Main(string[] args)
        {

```



```

        bonus b = new bonus(employeeBonus);
        b(2000);

    }

    public static void employeeBonus(int salary) => Console.WriteLine(salary * 2);
}

```

Now let's make another method called manager that displays the salary multiplied by three like the following,

```

using System;

namespace chopShop
{
    public delegate void bonus(int salary);
    class Program
    {

        static void Main(string[] args)
        {

            bonus b = new bonus(employeeBonus);
            b(2000);

        }

        public static void employeeBonus(int salary) => Console.WriteLine(salary * 2);
        public static void managerBonus(int salary) => Console.WriteLine(salary * 3);
    }
}

```

Now, we can add the new, managerBonus method to the delegate with the following

```

using System;

namespace chopShop
{
    public delegate void bonus(int salary);
    class Program
    {

```

```

    static void Main(string[] args)
    {

        bonus b = new bonus(employeeBonus);
        b += managerBonus; //add method
        b(2000);

    }

    public static void employeeBonus(int salary) => Console.WriteLine(salary * 2);
    public static void managerBonus(int salary) => Console.WriteLine(salary * 3);
}

```

The magic in this example lies in the

```
b += managerBonus; //add method
```

line of code. The += signals to C# that we are adding a new method to the delegate, so when we run this, we get the following,

```
4000
6000
```

As you can deduce, the employee bonus is 4000 and the manager bonus is 6000, and we called both methods and passed in the value all at one shot. Notice what I just said, we passed in all the values at one shot. When using multicast delegates, you can only pass in one value for all the methods. Since we passed in 2000, that value gets passed to all the methods the delegate invokes.

Now, as I stated before if you have multiple methods that return a value the delegate will ignore all the returns excepts for the last one. So, let's see what happens when we modify the methods to have a return type. To make everything be compatible change the return type of the delegate and the methods to int. When you're done your program should look like the following,

```

using System;

namespace chopShop

```

```

{
    public delegate int bonus(int salary);
    class Program
    {

        static void Main(string[] args)
        {

            bonus b = new bonus(employeeBonus);
            b += managerBonus; //add method
            int returnVal = b(2000);
            Console.WriteLine(returnVal);

        }

        public static int employeeBonus(int salary) => salary * 2;
        public static int managerBonus(int salary) => salary * 3;
    }
}

```

Running this will produce,

6000

What we got was the returned value of the last method assigned to the delegate, in this case, the manager's bonus.

Okay, so we can add methods to a delegate, but you know what things change. What if we don't need a method anymore? Well we can always remove a method from the delegate with

-=

So, let's say the board finally figures out that the managers make way too much money and they are as worthless as recycled toilet paper, and they want to cut their bonus. To modify our program to accommodate the change all we have to do is the following modification to the program.

```

using System;

namespace chopShop
{

```

```

public delegate int bonus(int salary);
class Program
{
    static void Main(string[] args)
    {
        bonus b = new bonus(employeeBonus);
        b += managerBonus; //add method
        b -= managerBonus; //remove method
        int returnVal = b(2000);
        Console.WriteLine(returnVal);
    }

    public static int employeeBonus(int salary) => salary * 2;
    public static int managerBonus(int salary) => salary * 3;
}

```

In this case, we used the -= on the managerBonus method and after running the program we get,

4000

The 4000 just so happens to be the employee bonus, and since we know that when there is more than one method with a return type the returned value reflects the last method called we can safely deduce that the managerBonus method has been removed.

Now that we have a decent grasp on multicast delegates let's move on to something really cool. Suppose we have a method that loads values to an array. Under certain conditions it'll load a value multiplied by 3 while under other conditions it'll need to load a value multiplied by 4. To accomplish this we would need to make two separate methods, one that multiplies by 4 and the other that multiplies by 3. Now for such a simple change in writing, two different methods are overkill. Enter the world of delegates, again. Delegates allow us to pass methods as arguments to other methods. So, to solve our little problem let's make two methods called times4 and times3. These methods will both return an int value. Let's also make delegate called multiplier of type int. When you're finished your program should look like the following,

```

using System;

namespace chopShop
{
    public delegate int multiplier (int val);
    class Program
    {

        static void Main(string[] args)
        {

        }

        public static int times4(int val) => val * 4;
        public static int times3(int val) => val * 3;
    }
}

```

Now, we're going to add a method called load. Load is going to have a void return type and is going to take an array of numbers and an argument of type multiplier called m. It is not fair to call m a type as m is going to be a method that matches the multiplier delegate's signature. The whole purpose of the load method is to take the passed-in array and perform the passed-in method on it. Once the operations have been performed on the array it'll load them into a new array and display the elements of the new array. As such, when all is said and done the load method should look like the following,

```

public static void load (int [] arr, multiplier m)
{
    int [] loadedArray = new int[10];

    for(int i = 0; i < 9; ++i)
    {
        loadedArray[i] = m(arr[i]);
    }

    for(int i = 0; i < loadedArray.Length; ++i)
    {
        Console.WriteLine(loadedArray[i]);
    }
}

```

```
}
```

As can be seen in the second argument, `m`, we declare it as if was a normal type; however, we are using a delegate name where we would normally declare a datatype. First, let's pass in the `times3` method and see what happens. To do this run the following,

```
using System;

namespace chopShop
{
    public delegate int multiplier (int val);
    class Program
    {

        static void Main(string[] args)
        {

            int[] arr = new int[] { 0, 1, 2, 3, 5, 6, 7, 8, 9 };
            load(arr, times3);

        }

        public static int times4(int val) => val * 4;
        public static int times3(int val) => val * 3;

        public static void load (int [] arr, multiplier m)
        {
            int [] loadedArray = new int[10];

            for(int i = 0; i < 9; ++i)
            {
                loadedArray[i] = m(arr[i]);
            }

            for(int i = 0; i < loadedArray.Length; ++i)
            {
                Console.WriteLine(loadedArray[i]);
            }
        }
    }
}
```

If all goes well you should get the following output,

```
0
3
6
9
15
18
21
24
27
0
```

So, in this case, we passed in a function that multiplied each element in the array by three, and as can be seen, each element in the array was multiplied by three. Now, let's pass in the method that multiplies each element by four. Modify the program to match the following,

```
using System;

namespace chopShop
{
    public delegate int multiplier (int val);
    class Program
    {

        static void Main(string[] args)
        {

            int[] arr = new int[] { 0, 1, 2, 3, 5, 6, 7, 8, 9 };
            load(arr, times4);

        }

        public static int times4(int val) => val * 4;
        public static int times3(int val) => val * 3;

        public static void load (int [] arr, multiplier m)
        {
            int [] loadedArray = new int[10];

            for(int i = 0; i < 9; ++i)
            {
                loadedArray[i] = m(arr[i]);
            }
        }
    }
}
```

```

        for(int i = 0; i < loadedArray.Length; ++i)
        {
            Console.WriteLine(loadedArray[i]);
        }
    }
}

```

If all goes well you should get the following output,

```

0
4
8
12
20
24
28
32
36
0

```

Delegates can be declared in a class; however, if you're passing in a method that method does not have to be declared in a class. For example, let's move the load method and the multiplier delegate to the BigBoss class and create a like so,

```

//BigBoss
using System;
using System.Security.Cryptography.X509Certificates;

namespace chopShop
{
    class BigBoss : Iemployee, IBoss
    {
        public delegate int multiplier(int val);
        public void ArrivalTime()
        {
            Console.WriteLine("3pm");
        }

        public void BeUseless()
        {
            Console.WriteLine("zzzzzzz");
        }

        public void crimes()
        {
            Console.WriteLine("DUI");
        }
    }
}

```



```

        Console.WriteLine("Assault and Battery");
        Console.WriteLine("Drug Usage");
        Console.WriteLine("Terrorism");
        Console.WriteLine("Illegal Gambling");
        Console.WriteLine("Resisting Arrest");
        Console.WriteLine("Back Child Support");
    }

    public void fireEmployees()
    {
        Console.WriteLine("You're fired!!!");
    }

    public void getPaid()
    {
        Console.WriteLine("Every day");
    }

    public void hireEmployees()
    {
        Console.WriteLine("You're hired");
    }

    public void load(int[] arr, multiplier m)
    {
        int[] loadedArray = new int[10];

        for (int i = 0; i < 9; ++i)
        {
            loadedArray[i] = m(arr[i]);
        }

        for (int i = 0; i < loadedArray.Length; ++i)
        {
            Console.WriteLine(loadedArray[i]);
        }
    }
}

```

Now, let's modify our main program to match the following,

```

using System;

namespace chopShop
{
    class Program
    {

```

```

static void Main(string[] args)
{
    BigBoss bigBoss = new BigBoss();
    int[] arr = new int[] { 0, 1, 2, 3, 5, 6, 7, 8, 9 };
    bigBoss.load(arr, times3);
    Console.WriteLine("-----");
    bigBoss.load(arr, times4);
}

public static int times4(int val) => val * 4;
public static int times3(int val) => val * 3;
}

```

As can be seen, we removed all references to the load method and the delegate from the Main method but we left the time3 and times4 methods that we are going to pass to the load method that is now in the BigBoss class. When the program is ran the output should match this,

```

0
3
6
9
15
18
21
24
27
-----
0
4
8
12
20
24
28
32
36
0
0
1
2
3
5
6
7
8
9
-----
0
1
2
3
5
6
7
8
9

```

As you can see, the methods that we passed in were in a separate class and they were static, and everything still ran fine. At a practical level passing in a method as a parameter is the same as passing a variable around, the same basic rules apply.

In essence, delegates allow us to pass around methods as if they were

variables. This is a very handy trick to know and know well. I can't count the number of times that passing functions to arguments has saved me. There are times where the operation you need to perform on something will vary depending on a certain condition. As such, before you move on, I suggest playing around with passing methods and becoming familiar with the concept.

## Namespaces

If you look at any of the past examples, you'll see the keyword `namespace`. However, they are much more than just pretty keywords and boilerplate code. Namespaces are organizational tools that are used to organize projects. Generally, namespaces are used to organize large projects, but they can be used anywhere organization is needed. The best way to think of namespaces is as containers that hold any of the following,

- Classes
- Structs
- Other namespaces
- Interfaces
- Delegates

In other words, everything we have looked at thus far can go into a namespace. As can be seen from the above list the items listed are basic building blocks of C# programs and each of these must be embedded in a namespace.

The basic syntax for declaring a namespace is as follows:

```
namespace <name>
{
    //code
}
```

For the most part, I've never found it necessary to create a namespace from scratch. The only determining factor for what goes into what namespace is determined by `<name>` and that's it. Usually, what I do is just create a class file and change the namespace. The only time I create namespaces from scratch is when I create a struct. Now, that may differ for you, it depends on what you're working on and what you're trying to accomplish. So, if you

start creating namespaces be careful with what you put for the name. It is easy to accidentally add things to the wrong namespace and not realize it until problems come up.

If you include something in a namespace it can't be used outside of the namespace unless the namespace is imported. Namespaces are imported with the using keyword. To demonstrate namespace let's take or VehicleData struct and move it into a namespace called salesInformation.

```
namespace salesInformation
{
    struct vehicleData
    {
        public double MPG;
        public int miles;
        public double value;
    }
}
```

Now let's try to access like we did before,



The type or namespace name 'vehicleData' could not be found (are you missing a using directive or an assembly reference?)  
[Show potential fixes](#) (Alt+Enter or Ctrl+.)

As can be seen, the struct is not be found. Now, let's see what happens when we modify the file the Main method is in with the following line of code,

```
using salesInformation;
```

After you modify the file the Main method is in it should look like the following,

```
//main
using System;
using salesInformation; //import namespace

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
```

```

//reference variable
vehicleData MustangData;

//assign values
MustangData.miles = 3;
MustangData.MPG = 26.5;
MustangData.value = 45000;

//display data
Console.WriteLine("Mustang miles: {0}", MustangData.miles);
Console.WriteLine("Mustang MPG: {0}", MustangData.MPG);
Console.WriteLine("Mustang value: {0}", MustangData.value);

    }
}
}

```

After running the program, you should get the following just as before,

```

Mustang miles: 3
Mustang MPG: 26.5
Mustang value: 45000

```

Importing namespaces is vital when you use a library that isn't automatically included. When using libraries, it is often common to have namespaces inside of namespaces and importing them takes a little extra work. Importing embedded namespaces is a lot like peeling the layers on an onion. You have to import all the namespaces up to the namespace you're targeting. For example, say you have a namespace called Ford inside a namespace called Truck like the following,

```

using System;

namespace Truck
{
    namespace Ford
    {
        class F150
        {
            public void engine()
            {
                Console.WriteLine("rumble");
            }
        }
    }
}

```

```
}
```

To be able to access the F150 class you would need to import the namespaces like so,

```
using Truck.Ford;
```

# CHAPTER 4: TYPING

If you're a programmer at least one sweaty, greasy interviewer has asked you about typing. These greasy bums ask this because one, it makes them feel good and two a lot of people don't understand typing. Typing is kind of weird in my opinion and too most, myself included, it is very confusing. Many developers that I've encountered over the years don't know, or really care for that matter, if they are programming in weakly typed, strongly typed, dynamically typed, or statically typed language. Generally, in my experience, the little kinks of typing are often chopped up to being kinks of the language itself. However, as I stated before the difference between a good programmer and an okay programmer is stuff like this. Before we start diving into typing I need to add a little disclaimer. Believe it or not, typing is a concept that many new developers don't get and there's a good reason. There is no universally agreed-upon definition of typing. Definitions of typing are like opinions, everyone has one. Really when you think about it, typing is an opinion, and for this book, you're going to get mine.

There are four different types of typing. There are dynamically, statically, weakly, and strongly typed languages. All four of these types often get lumped together by new programmers and I've often heard newbies argue which typing system is the best. In reality, each typing system will belong to one of two groups and the best way to think of these groups is as follows:

- 1) Weakly typed / Strongly typed
- 2) Dynamically typed / Statically typed

Generally speaking, each language will get a characteristic from group 1 and a characteristic from group 2. In other words, weakly typed is the opposite of strongly typed and vice versa while dynamically typed is the opposite of statically typed and vice versa. For the most part, a programming language will fall into one of four possible categories, see table 4.1

Weakly and Statically Typed	Strongly and Statically typed
Weakly and Dynamically Typed	Strongly and Dynamically typed

## Table 4.1

Now it's time for a second disclaimer. There isn't a language that 100% fits any of these categories. As can be seen, a language is either weakly or strongly typed and either dynamically or statically typed. Well, that's all fine and dandy but what does all that mean? As I stated before, that's a matter of opinion.

### Strongly Typed and Weakly typed Languages

To begin our adventure into typing let's start by looking at group 1, strongly and weakly typed languages. A strongly typed language is a language that enforces type safety. With strongly typed languages a variable is tied to a specific data type. A strongly typed language also enforces a strict set of rules on operations performed on variables. In layman's terms, a strongly typed language provides a lot of safety in terms of data typing. For example, an integer variable cannot be combined with a string variable. Consider the following pseudo-code example,

```
Integer a = 10;  
String b = "banana";  
Print(combine(a, b));//error will be thrown.
```

Examples of strongly typed languages are Java and Python amongst many others.

On the other hand, a weakly typed language does not provide the type safety that a strongly typed language does. Think of a weakly typed language as the opposite of a strongly typed language. Weakly typed languages don't care about data types when performing operations. Just as we saw an error be thrown with the last example based on a strongly typed language the following will be outputted in a weakly typed language.

```
Integer a = 10;  
String b = "banana";  
Print(combine(a, b));//banana10 will be printed
```

Examples of weakly typed languages are PHP and JavaScript.

So, there are pros and cons to both strongly and weakly typed systems. For starters, a strongly typed system is going to be way more stringent on what you can and can't do, for example, comparisons. A



strongly typed language is not going to let you compare an integer value to a double value. However, the rigidity of this type of system does promote type safety, meaning you don't have to worry about accidentally comparing a boolean value to a string value.

In terms of weakly typed systems, you do get a lot more freedom. However, this freedom does come with a cost. Yeah, you can combine a string and an integer which in some cases might be a good thing but at the same time, it might not. Debugging a piece of software written in a weakly typed system can be an utter nightmare. You might think you know what a datatype is when you're operating but oh too often will that operation return unexpected results stemming from something you thought was one datatype but turned out to be another datatype.

## Statically Typed Language vs Dynamically Typed Language

Now that we've got an idea of what strongly and weakly typed languages are let's move our attention to dynamically and statically typed languages. To start with let's look at what a statically typed language is. If you've ever used Java, C++, or so on you've used static typing. Consider the following snippet,

```
int var1;
```

In this case, we are explicitly telling C# what kind of data we want each of the variables to hold. Take var1 for example, we are declaring var1 as an integer and for the entire life of var1 it will only be able to hold integers. In short, we cannot assign a value that is not integer to var1.

So, what would happen if we did try to assign a value of a different data type to a variable in a statically typed language? Well, things wouldn't go well. Consider the following example,

```
int var1;  
var1 = 3.25; //Uh oh!
```

In this snippet, we are telling the compiler/interpreter that we want var1 to be an integer, then on the next line, we're telling the compiler/interpreter to assign a value of type double to the variable. This causes sparks to shoot out

of your computer, the compiler/interpreter throws up its hands, and worst of all your program won't run. All this drama is caused because in statically typed languages this form of type checking is done at compile time

Now, on the other hand, there is dynamic typing. If you've ever used a language like Python or JavaScript, you're already familiar with dynamic typing. In dynamically typed languages the type is inferred during run time. Consider, the following:

```
var1 = 2
```

In this code snippet, we are not explicitly declaring a data type. In this case, the compiler or interpreter has to figure out the data type when the program runs. What will happen is when that line of code is run the compiler/interpreter will give the variable the data type of the assigned value, in this case, var1 will be assigned an integer data type.

There are some pros and cons to using a dynamically typed language over a statically typed language and vice versa. However, in my opinion, there are two biggies in terms of development that should be addressed. Statically typed languages take a bit more upfront thought and design work in terms of figuring out data types; however, when something goes wrong, and trust me things will go wrong, they are much easier to debug. On the other hand, variables are much easier to declare, and it takes much less effort to figure out what type a variable should be. However, the main con to dynamically typed languages stems from the same fact that you're not declaring a data type. This means that you may not be completely sure what the data type of a variable is, and since types are being assigned at runtime you may not realize you have a bug until an edge situation occurs. As with everything else choosing between a dynamically typed language and statically typed language boils down to your needs.

## **Fudging it: what is the difference between implicit, explicit, and dynamic typing in C#**

Modern programming is seeing the lines blurred between everything, and many traditionally stringently static languages are allowing dynamic typing and vice versa. Traditionally in C#, you declare a data type to

a variable at the time it is initialized. However, starting with C# 3.0 that started to change, enter the world of C#'s typing. Now, implicit, explicit, and dynamic typing are and aren't the same thing as the type systems we explored above. The definitions hold the same meaning but when you're developing a piece of software in C# the way you think of these concepts changes a little.

Traditionally in C#, you declare a variable with a specific type when the variable is first declared.

```
int var1;
```

This is what is called explicit typing. In this context, we are explicitly telling C# that we want var1 to be an integer and only hold integer values. For primitive data types such as ints, doubles, and so on explicit typing is the way to go. However, if you have something strange like a long class name or there is no way of knowing if you're going to get 2 or 2.2 back you may want to consider using what is known as implicit typing.

Starting with C# 3.0 the var keyword was introduced. The var keyword signals to the C# compiler that it needs to figure out the data type of a variable at compile time. What the C# compiler will do is look at the data type of the value assigned to the variable and assign that data type to the variable. Consider the following syntax,

```
var var1 = 2;
```

In this case, we are telling the C# compiler right off the bat that we have a variable called var1 and we're not sure of the data type. We're also telling the compiler that we're assigning the value of 2 to var1 and as such C# figures that var1 must be an integer type. Wow, that's a mouth full.

Now there are a few gotchas with the var keyword in C#. First, once a datatype is assigned to a variable using the var keyword it cannot be changed. Considering the above var example, since we assigned a 2 to var1 and var1 got assigned the datatype of integer it cannot be changed. The second gotcha is that a variable declared with var must be initialized as soon as it is declared. This little gotcha means that you cannot use the var keyword as a method argument type. The final little gotcha is that you can but really shouldn't use var to declare primitive data types. Using var with

primitives can make a program very hard if not impossible to maintain. Using var with primitive data types is a lazy and bad way of programming. For me the only time I use the var keyword is when there's a long or weird class name that I'm creating an object for.

The next form of typing we need to look at is dynamic typing in C#. As with dynamically typed languages like Python, C# has a keyword that allows the developer to define a variable's data type during runtime. This keyword is the dynamic keyword. The syntax for using the dynamic keyword is as follows:

```
dynamic var1 = 2.5;
```

The above line of code in practice does the same as the example with the var keyword. We're telling C# that we have a variable we don't know the type of yet and to assign the data type of 2.5 to var1. For those of you who have never used dynamic or implicit typing in C# the difference between the var and dynamic keywords maybe a bit confusing. At first glance they seem to be doing the same thing. However, there is a world of difference. When a variable is declared with the dynamic keyword the type is inferred when that line of code is running. Unlike implicit typing with the var keyword where you have to initialize the variable at the same time, when the variable is declared with the dynamic keyword you can initialize the variable after it is declared. This means that you can use the dynamic keyword for argument types in methods. However, there is a major gotcha with dynamic. The data type of a variable declared with dynamic can change throughout the program's lifecycle. I call this a gotcha, but I have used this feature with great success in the past. However, since the datatype can change, it can be accidentally changed or changed on fringe cases which in turn can make the program difficult to debug.

Between implicit and dynamic typing I tend to favor dynamic. Now, that doesn't mean it is the best option or the best option for what you're trying to accomplish. I found that dynamic typing is an excellent option when passing objects such as objects to methods.

# CHAPTER 5: ARRAYS

Usually, arrays are the first topic covered in most programming books; however, if you're reading this book you probably know what a basic array is. If you don't it's a data structure that holds multiple values. Essentially, it's one variable that holds multiple values. A program of any significant size or functionality is going to have to utilize arrays. In fact, there's no getting around arrays in the modern programming world.

## Common Array

The most common type of array that you're going to run across is a single dimension array. If you're not familiar with the term dimensions don't worry that's coming. The most common type of array is going to have the following syntax,

```
int[] x = new int [] { 1, 2, 3, 4 };
```

Here we have an integer array named x that is initialized with four integer values. Arrays are accessed with their index number, note that arrays are zero-indexed as such,

```
using System;
namespace chopShop
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] x = new int [] { 1, 2, 3, 4 };
            Console.WriteLine("Index 0: {0}", x[0]); // 1
            Console.WriteLine("Index 1: {0}", x[1]); // 2
            Console.WriteLine("Index 2: {0}", x[2]); // 3
            Console.WriteLine("Index 3: {0}", x[3]); // 4
        }
    }
}
```

will produced

```
Index 0: 1
Index 1: 2
Index 2: 3
Index 3: 4
```

Values can be assigned to a specific index with the following,

```
using System;
namespace chopShop
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] x = new int [] { 1, 2, 3, 4 };
            x[0] = 500; // re-assign x[0] to 500
            Console.WriteLine("Index 0: {0}", x[0]); // now 500
            Console.WriteLine("Index 1: {0}", x[1]); // 2
            Console.WriteLine("Index 2: {0}", x[2]); // 3
            Console.WriteLine("Index 3: {0}", x[3]); // 4
        }
    }
}
```

Okay, so this was a crash course to arrays, now let's get to the fun stuff.

## Looping: Foreach

C# has a very easy and effective way of looping through an array called the foreach loop. If you need to loop through an array the best way to do that is with a foreach statements. In practice, the foreach statement is a lot like a for loop; however, unlike a traditional for loop it will only iterate through arrays or collections (we'll tackle these later) and does not need a counter variable. The basic syntax for a foreach loop is as follows,

```
foreach(<data_type> <variableName> in <array> )
{
    // do stuff to <variableName>
}
```

So, let's see how the foreach would work on our first array example.

```
using System;
namespace chopShop
```

```

{
    class Program
    {
        static void Main(string[] args)
        {
            int[] x = new int [] { 1, 2, 3, 4 };
            foreach (int i in x)
            {
                Console.Write(i);
            }
        }
    }
}

```

1234

As can be seen pretty much what you would expect from a for loop.

## Multidimensional Arrays

C# also supports multidimensional arrays. Multidimensional arrays can best be thought of as rows and columns. In theory, you can have as many dimensions as you want; however, the more dimensions you start adding the harder the array is to understand, at least in my opinion. However, no rule limits the number of dimensions you can have.

Let's look at declaring a multidimensional array,

```

int[,] x = new int [a, b]; //2d
int[,,] x = new int[a, b, c]; //3d
int[,,,] x = new int[a, b, c, d]; //4d

```

or if you are initializing the array on start up

```

int[,,,] arr4d1 = new int[,,,]//3d
{
    {
        {
            { 1, 2},
            { 3, 4}
        }
    }
};

```

If you're initializing on startup just make sure the number of commas matches on both sides.

Of course, these are integer arrays and like normal arrays, you can declare them as any data type you wish. Now, notice the number of commas in the square brackets and the number of dimensions, there is always one less comma. In other words, there are always

$$\text{Number of Commas} = \text{rank} - 1$$

Notice the word rank in the above equation. In programming, rank refers to the number of dimensions an array has. For example, a 3d array has a rank of three while a 10d array would have a rank of 10. In all, this is a handy little equation that has always served me well in the past.

Now, multidimensional arrays, much like our boss's life choices, can get very confusing very quickly. It can be very difficult to figure out which element is in which dimension, so the best way to think about these dimensions is as a hierarchy of curly brackets when you initialize the array at declaration. In short, the number of curly bracket levels is going to equal the rank, so to declare and initialize an [1,1,1,2] array, which is a four dimension array we need it to be four curly brackets deep with the last set of curly brackets having two elements like so,

```
int[,,] arr4d = new int[,,]
{
    {
        {
            { a, b }
        }
    }
};
```

Here we have a 4-dimensional array. If you notice starting at the first curly bracket and count to the bracket with elements there are four. If we wanted to make a [2,3,4] we would need three levels of curly brackets with the last two levels having two sets of curly brackets like the following,

```
int[,] arr3d = new int[,]
{
    {
        {1,2,3,4},
        {5,6,7,8},
        {9,10,11,12}
    },
}
```



```
{  
    {13,14,15,16},  
    {17,18,19,20},  
    {21,22,23,24}  
},  
};
```

Accessing the elements is the same as with a normal array only with extra indexes. Working off the above example let's say we want to access the 19. The 19 is in position [1,2,2] as such the following line of code should retrieve the 19

```
Console.WriteLine(arr3d[1, 1, 2]);
```

19

Also, like regular 1d arrays, it is not necessary to initialize a multidimensional array at startup. For example, we can declare the same array in the last example with the following syntax,

```
int[,.] arr3d = new int[2, 3, 4];
```

From this point pretty much, everything is the same with 1d arrays. You loop through higher-dimensional arrays, you assign values the same way, and so on. As such we're going to wrap up multidimensional arrays.

## Jagged Arrays

Imagine this, you're tasked with creating a way of organizing the chop shop's monthly income, and the boss's main business's main income. However, there's a problem, considering your boss the books are less than perfect to say the least, as such you never know if the income for that month is going to be for the chop shop or the main business. As such, you need a organizing the data so that it can be easily inserted into the main business or the chop shop data structure. After thinking about it for a minute you decide that the chop shop should be its own array that has one element for each month and another array that also has one element for each month that will represent the earnings for the main business. Now, you have to decide how you're going to distinguish between sorting the data for the two businesses. You could use some convoluted logic that uses a series of control statements

that decided which is the proper array for the data or a struct, but this is something a tester would do. A much simpler method of storing and sorting the data would be to use a jagged array.

Jagged arrays are arrays that hold other arrays as opposed to data such as integers or strings. Jagged arrays are declared with the following syntax,

```
<data_type>[][] jaggedArray = new <data_type>[<num_of_indexes>][];
```

So, if we wanted to declare a jagged array with two elements of type integer, we would use the following syntax,

```
int[][] jaggedArray = new int[2][];
```

Jagged arrays can be loaded two ways. The first way is like assigning data to an index in a regular 1d array.

```
int[] array1 = new int []{ 1, 2, 3 };// JaggedArray[0]  
int[][] jaggedArray = new int[2][];//declare the jagged array  
jaggedArray[0] = array1;//assign array to [0]
```

we can also initialize each element in the jagged array with an array like so,

```
int[][] jaggedArray = new int[2][];//declare the jagged array  
jaggedArray[0] = new int[] { 1, 2, 3, 4 };  
jaggedArray[1] = new int[] { 1, 2, 3, 4, 5, 6 };
```

Okay, so all this is fine and dandy, but how does it make life easy for us and help us solve our problem with easily entering and retrieving profits for the month? Well, let's look at some code, first let's create two arrays, chopShop and mainBusiness like so,

```
double[] chopShop = new double[] { 100, 200, 350, 3260, 30000, 30430, 12500, 16700, 900, 100,  
1530, 13580 };  
double[] mainBusiness = new double[] { 1200, 1450, 2000, 2400, 22400, 5000, 12300, 289, 230, 5,  
22353, 10000 };
```

next we're going to create a two-element jagged array called profits,

```
double[][] profits = new double[2][];
```

Now, like in the first jagged array example, we're going to assign chopShop to profits[0] and mainBusiness to profits[1].

```
profits[0] = chopShop;  
profits[1] = mainBusiness;
```

As of this point much like the first example, the arrays are loaded into the jagged array. Now we need to retrieve the data

One way to getting the data from the jagged array is by looping. Jagged array looping is quite simple. Since a jagged array is an array of arrays you can use the foreach loop to cycle through the elements of the jagged array. So, if we wanted to cycle through jaggedArray[1] and display the output to the console we would use something like so,

```
int[][] jaggedArray = new int[2][]; //declare the jagged array  
jaggedArray[0] = new int[] { 1, 2, 3, 4 };  
jaggedArray[1] = new int[] { 1, 2, 3, 4, 5, 6 };  
  
foreach (int x in jaggedArray[1])  
    Console.WriteLine(x);
```

123456

So, applying this to our profits array we can retrieve the chop shop's monthly profits along with the main business's monthly profits with the following,

```
Console.WriteLine("chop shop");  
foreach (int x in profits[0])  
    Console.WriteLine(x);  
  
Console.WriteLine("main business");  
foreach (int x in profits[1])  
    Console.WriteLine(x);
```

Which in turn will yield the following output,

```
chop shop
100
200
350
3260
30000
30430
12500
16700
900
100
1530
13580
main business
1200
1450
2000
2400
22400
5000
12300
289
230
5
22353
10000
```

Surprise, surprise we can now see all the profits for the year.

Now, this is a start, but for this to be useful we have to have the ability to add the data in month by month and we have to be able to retrieve each month's data individually. Do we need fancy logic? No! Jagged arrays still have use covered and this is where jagged arrays really started to behave like multidimensional arrays.

Inserting and retrieving data from individual elements is much like inserting and retrieving data from a multidimensional array. Consider the following code,

```
double[][] profits = new double[2][];
double value = profits[x][y];
```

Much like in the first example we created a jagged array named profits which holds two arrays. On the next line, we are accessing a single value and assigning it to the variable value. Pretty simple. Now, the value we are accessing is the y element in x array.

So, how do we apply all this to our example, well let's say we want to access the main business's profits for August. To pull this data we would use the following,

```
Console.WriteLine("August Profits: {0}", profits[1][7]);
```

The following code snippet will output,

```
August Profits: 289
```

Notice that we used

```
profits[1][7]
```

we used the 7<sup>th</sup> index for August because any array in C# is 0 indexed. Essentially, all we did was tell C# to take the array in index 1 and take the 7<sup>th</sup> index of that array. So, in short, this is just like a multidimensional array.

Assigning a value to specific indexes works the same way. Suppose we want to change the main business's August profits from 289 to 4567. To do this we would do the same thing that we would do with a normal array,

```
profits[1][7] = 4567;  
Console.WriteLine("August Profits: {0}", profits[1][7]);
```

This code will output,

```
August Profits: 4567
```

In short, once you get the hang of the first set of square brackets choosing the array and the second set selecting the index of the array you've mastered jagged arrays. Generally, I like to use jagged array for scenarios like the one we worked through. A lot of new programmers like to opt for fancy logic that uses control statements to pick the proper array to load the data to; however, using jagged arrays will usually gain you points on an interview or with your boss in a code review. Overall, I recommend playing with and

mastering jagged arrays as they are, what I would call, a gem of the C# language.

# CHAPTER 6:

## COLLECTIONS

At this point, we've established that our boss is degenerate, drunk, incompetent, and can't make up his mind. As we've seen time and again our program has to be flexible, so counting on our boss to give us a decent ballpark figure for of the number of variables we need is laughable. Since our boss is so incompetent at life and his mind changes with every sip of whisky, we need a flexible way of adding elements to an array-like structure. To do this we're going to use what I consider one of the most powerful features of any programming language, collections.

Collections are one of my favorite aspects of any programming language. I personally nerd out over the types of collections a programming language has to offer. For those of you reading this that have never heard of a collection before, think of collections as arrays on steroids. C# has a lot of cool built-in methods that allow developers to do a whole variety of operations on collections. For example, C# has built-in methods to add elements, delete elements, return the size of a collection, and so much more.

Not only are elements very useful they are also common interview questions. I can't count the number of times I've been asked about the difference between this collection and that collection, or how this collection works versus the way that collection works. So, this is why I decided to dedicate a whole chapter to collections. By no means will every type of collection be covered. The collections that will be covered are commonly used collections and collections that will often be inquired about in interviews. I've blown and seen many interviews blown for answering collection questions wrong, so don't take the following lightly.

In modern programming, it is becoming much more common, and some would say better practice to use a collection over an array. Arrays are still a vital part of programming and aren't going anywhere anytime soon but they do have their limitations compared to collections. With all that said,

when should you use an array over a collection? Well, that's not a black and white clear-cut answer as it depends on what you're trying to accomplish. My general rule of thumb is if there isn't an absolute predefined number of elements and I know I'm not going to need to add and remove elements it is best to use a collection.

C# supports two types of collections, a generic collection, and a non-generic collection. A generic collection is type-safe meaning it can only hold values of a specific, predetermined data type, and a non-generic collection is not type-safe and can hold any data type. Due to the type safety many developers favor generic collections over non-generic collections. To use generics you need to import the `System.Collections.Generic` namespace with using `System.Collections.Generic`. To be able to use a non-generic collection you need to import the `System.Collection` with the following line of code using `System.Collections`. If you're a newbie it might be tempting to import `System.Collections.Generic` and assume you're going to be able to access non-generic members; however, you're not. To be able to access non-generic members you have to explicitly import the `System.Collections` namespace. Now, if you're using a later version of Visual Studios all you have to do is start declaring the name of the collection and Visual Studios will automatically import the namespace. So, with all that in mind let's start exploring.

## **Generic Collections**

C# support multiple generic collections that are as follows:

- `List<Type>`
- `Dictionary<Type of Key, Type of Value>`
- `Queue<Type>`
- `Stack<Type>`
- `HashSet<Type>`
- `SortedList<Type of Key, Type of Value>`

Each type of collection serves its purpose and has its little quirks and is handy for a certain task. The following section is going to breakdown how each of these collections works and some of the more useful methods and properties associated with the collection.



## Lists

To start off with let's take a look at Lists. This is probably the most used collection type in C# and it's my personal favorite. It is probably the closest you're going to come to a general-purpose, generic collection. In fact, in most of the C# code I have seen, lists are more prevalent than arrays. To declare a list, you use the following syntax,

```
List<type> variable_name = new List<type>();
```

As I'm sure you've already figured "type" is the data type of the collection. Now, let's make an empty list of type integer,

```
List<int> li = new List<int>();
```

Another way to initialize an array on declaration is with the following,

```
List<int> li = new List<int> { 0, 1, 2 };
```

With this syntax, the values 0, 1, and 2 are automatically added to the list. Notice when that the parentheses are dropped when you declare a list on the declaration, this is a little detail that I always forget.

Now, that we know how to declare and automatically initialize a list we need to a way to add elements to the list. To do this, we use the Add method. The Add method is pretty self-explanatory, all you have to do is append Add to the list name. The only gotcha you have to worry about is making sure you pass in a value of the proper type in the method. So, with that in mind let's add some elements to our list!

```
li.Add(1);  
li.Add(2);
```

It is important to note that like with arrays list is zero-indexed. So, to pull the values we can use the following,

```
Console.WriteLine(li[0]);  
Console.WriteLine(li[1]);
```

running the above lines of code will render the following output,

1  
2

Now that we know how to add elements to a List we need to learn how to delete elements from a list. C# offers multiple ways to remove elements. Two of the most common methods used to remove an element are with the Remove and the RemoveAt. Remove removes the first occurrence of value from the list. On the other hand, RemoveAt removes the element at the specified index. To clarify this let's look at some examples. First, let's create and initialize a list called test.

```
List<int> li = new List<int> { 0, 1, 2, 3, 2 };
```

Notice that in this list we have two occurrences of the integer 2. If we run a RemoveAt(2) the first 2 will be removed. Now, let's see the RemoveAt(2) in action.

```
li.Remove(2);  
  
foreach (int i in li)  
    Console.WriteLine(i);
```

0  
1  
3  
2

Originally our list consisted of 0,1,2,3,2. Then after we ran a RemoveAt(2) on the list our list changed to 0,1,3,2. As can be seen, the first occurrence of the 2 in the List was removed from the list. So, diving in a little bit deeper, RemoveAt will search the entire list for a value that you passed in. Once it finds the value it will remove it from the list and stop searching.

The final way to remove elements from a list that we are going to look at is the Clear method. The clear method is the nuclear option. When you perform Clear on a list you are removing all the elements from the list and what you're left with is a blank list with nothing in it. Clear can be used with the following syntax,

```
li.Clear();
```

You have to be careful when you use the Clear method. If you accidentally use Clear and then try to access any of the elements you will be met with an error and your program may not compile. The Clear method is very handy, but it can lead to some headaches if you're not careful, so pay attention to how you use it.

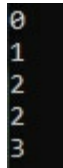
The next handy method that List<type> supports is the Sort method. As the name suggests the sorts the elements in your list. The best way to explore this method is to play around with it.

```
List<int> li = new List<int> { 0, 1, 2, 3, 2 };

li.Sort(); //sort the list

//display data
Console.WriteLine(li[0]);
Console.WriteLine(li[1]);
Console.WriteLine(li[2]);
Console.WriteLine(li[3]);
Console.WriteLine(li[4]);
```

This snippet will produce the following output,



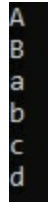
As would be expected the elements in the List are sorted numerically.

Now, what do you think would happen if we had a list of chars or strings? How, would you expect the list to be sorted?

```
List<char> test = new List<char> { 'a', 'A', 'b', 'B', 'c', 'd' };

test.Sort();

//display data
Console.WriteLine(test[0]);
Console.WriteLine(test[1]);
Console.WriteLine(test[2]);
Console.WriteLine(test[3]);
Console.WriteLine(test[4]);
Console.WriteLine(test[5]);
```



```
A
B
a
b
c
d
```

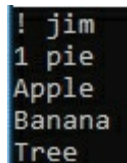
As can be seen when a list of chars is sorted capital letters are first, then lowercase letters, and all of which are sorted alphabetically. The reason for this is because the Sort method sorts the chars by their ASCII code. As such, you can expect uppercase characters, followed by lowercase letters. Special characters can also be sorted; however, some special characters are sorted before numbers characters, and some are sorted after lowercase letters. If you're not familiar with how characters are going to be sorted refer to an ASCII table.

Now that leaves us with strings. Let's look at the following code and output,

```
List<string> test = new List<string> { "Banana", "Apple", "Tree", "1 pie", "! jim" };

test.Sort();

//display data
Console.WriteLine(test[0]);
Console.WriteLine(test[1]);
Console.WriteLine(test[2]);
Console.WriteLine(test[3]);
Console.WriteLine(test[4]);
```



```
! jim
1 pie
Apple
Banana
Tree
```

As can be seen, the Sort method treats string almost the same way as it treats chars. If two or more strings have the same characters in the same position Sort will look at the next character and sort accordingly. However, a major difference is that when sorting strings with special characters. Strings with special characters will be sorted first, followed by numbers, lowercase letters, and finally, uppercase letters. A whole book can be dedicated solely to all the methods that accompany lists.

When it comes to List the last operation, we're going to look at isn't a method but a property. The property in question is the count property. The Count property, as the name suggests, gets the number of elements in the list.

```
List<string> test = new List<string> { "Banana", "Apple", "apple"};  
int size = test.Count;  
Console.WriteLine(size);
```

The snippet renders the following output,

3

As can be seen, we have three elements in the in List and the output from the count property reads three. Very simple, very easy to understand, and very useful especially for loops.

## Dictionaries

Dictionaries differ from List as a dictionary is a key-value pair. If you've ever used a Hashmap you have the basics of a dictionary, if you haven't don't worry, we're going to cover Hashmaps when we cover non-generic collections later on. In short, a dictionary is a generic Hashmap. The key-value pairs are type-safe and can only accept values that are of the declared type.

Dictionaries are declared in the same way as a List. To declare an empty dictionary the following syntax is used.

```
Dictionary<Key Type, Value Type> dic = new Dictionary<Key Type, Value Type>();
```

Much like a List a dictionary can be initialized on declaration with the following syntax.

```
Dictionary<int, string> dic = new Dictionary<int, string>  
{  
    {1, "Apple"},  
    {2, "Banana" }  
};
```

As can be seen, the key-value pairs are declared inside of curly braces which

are again declared inside of curly braces.

Accessing an element is different than accessing an element in an array or List. Since a dictionary is a key-value pair, an element or value is accessed with the key.

```
dictionary_name[key]
```

So, let's see this in action,

```
Dictionary<int, string> dic = new Dictionary<int, string>
{
    {1, "Apple"},
    {2, "Banana" }
};

Console.WriteLine(dic[1]);
Console.WriteLine(dic[2]);
```

This code will produce the following output.

```
Apple
Banana
```

Elements can be added to a dictionary the same way they are added to a list with the Add method. The only difference is that when you use the Add method dictionaries you have to pass in both a key and value pair of the proper types. To demonstrate let's make and initialize a dictionary of type <string, string> and add an employee and manager entry.

```
Dictionary<string, string> dic = new Dictionary<string, string>();

dic.Add("employee", "Steve");
dic.Add("manager", "Larry");

Console.WriteLine("Manager: " + dic["manager"]);
Console.WriteLine("Employee: " + dic["employee"]);
```

Let's take a second to analyze the code. We've established that to access a value we use the key's value which in the last example was an integer. However, this time we have a dictionary of type <string, string> this means that we have to pass in a string that matches one of the keys to retrieve the value as can be seen in the second and third line.

```
Manager: Larry  
Employee: Steve
```

As with Lists we need to turn our attention to removing entries from a dictionary. The Remove method works the same way the Remove method for a list works. The only difference between the two is instead of passing in the index of the element you want to delete as you would with a list you pass in the key-value for a dictionary. So, to demonstrate this we're going to revisit our employee dictionary. In this example, we are going to remove the manager entry from the dictionary.

```
Dictionary<string, string> dic = new Dictionary<string, string>();  
  
dic.Add("employee", "Steve");  
dic.Add("manager", "Larry");  
  
Console.WriteLine("Manager: " + dic["manager"]);  
Console.WriteLine("Employee: " + dic["employee"]);  
  
dic.Remove("manager");//remove manager  
Console.WriteLine("Manager: " + dic["manager"]); //will not run
```

In this case, the program will not run because we removed the manager entry, and then we tried to access that dictionary entry.

Much like the count property that List support, dictionaries also have properties. One such property is known as the Keys property. The Keys property creates a collection containing all the keys in the dictionary. To create the collection, you use the following syntax,

```
Dictionary<Key Type, Value Type>.KeyCollection key = dic.Keys;
```

What's important to note here is that the key and value types for the collection must match those of the original dictionary. When you're supporting legacy code or modifying a program that you didn't write the var keyword is an excellent way to create the collection.

```
var key = dic.Keys;
```

There is another property similar to the Key property that creates a collection of values. The collection for the values is created the same way

the key collection is made.

```
Dictionary<string, string>.ValueCollection values = dic.Values;
```

And, just like with a key collection the var keyword drastically simplifies creating the collection.

```
var value = dic.Values;
```

Dictionaries also have a Clear method and a count property that work the same as ones for the list collections and as such we're not going to waste time diving into examples. Also, much like Lists, dictionaries are very rich in terms of methods and properties. Another way to access the keys and the values with the KeyValuePair<type, type> struct. It is common to use the KeyValuePair struct in a foreach loop to access the keys and values. For example,

```
Dictionary<string, string> dic = new Dictionary<string, string>();  
dic.Add("employee", "Steve");  
dic.Add("manager", "Larry");  
foreach(KeyValuePair<string, string> x in dic)  
    Console.WriteLine("key: {0} value: {1}", x.Key, x.Value);
```

```
key: employee value: Steve  
key: manager value: Larry
```

In short, using the KeyValuePair struct allows us to store the dictionary's keys and values in a collection-like data structure. The keys and values are stored in the order they are added to the dictionary.

## Queues

At first glance, a queue may seem a lot like a list but they are worlds different. The most obvious difference is that queues are FIFO (First In First Out). This means the first element stored in the queue is the first element to be retrieved, elements in the queues can't be retrieved with the index like lists, values are not preloaded the same way as with lists, and once an element is consumed from the queue it is removed from the queue. However, declaring an empty queue is the same as declaring a list as can be seen,



```
Queue<Type> queue_name = new Queue<Type>();
```

Now, as I stated before initializing a queue at declaration is a little bit different than a list or dictionary. To initialize a queue, you use the following syntax,

```
Queue<type> q = new Queue<type>(new type[] { val1, val2, val3 });
```

As can be seen, you have to pass in the `new type[]` along with the values in curly braces into the constructor. This is a little detail that is easy to forget, and it's a killer for many people that are ever asked to preload a queue on a whiteboard challenge.

Now, there are two similarities with lists that should be covered. The `Clear` method and the `Count` property behave the same way as they do with Lists. The `Clear` method removes all of the elements in the Queue and the `Count` property gets the number of elements in the Queue. So, if you're familiar with how these two methods work with Lists you know how they work with Queues.

Unlike a List structure, there is no `Add` method. To add new elements to the queue you use the `Enqueue` method. `Enqueue` is a lot like the `Add` method used in List and Dictionaries. The `Enqueue` method adds an object to the end of the queue, which is a fancy way of saying that the element that was added with the `Enqueue` method is the last in the queue. To demonstrate let's create a queue called `q` and preloaded it with three values, display the number of elements in the queue, use the `Enqueue` method to add an extra value, and display the output again.

```
Queue<double> q = new Queue<double>(new double[] { 1.1, 2.2, 3.3 });  
Console.WriteLine(q.Count); //count is 3  
q.Enqueue(4.1); //add an new element to the queue  
Console.WriteLine(q.Count); //count is 4
```

As can be seen in the screenshot below, the `Enqueue` method behaves just like the `add` method in a list or dictionary.



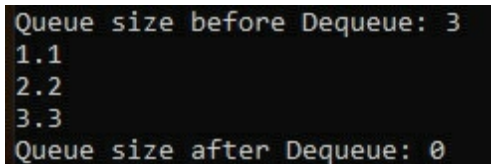
```
3  
4
```

To retrieve a value from a queue you use the `Dequeue` method. Now,

there are a few major gotchas with the Dequeue method. First, since queues are FIFO Dequeue will pull the oldest elements. Secondly, Dequeue removes the element from the queue. So, you have to be very careful and pay very close attention to the queue after you use the Dequeue method on it. To demonstrate the Dequeue method we're going to use the Queue we created in the last example and we're going to get the number of elements in the Queue, use a loop to display the elements retrieved from the Queue using the Dequeue method, and finally get the number of elements in the Queue.

```
Queue<double> q = new Queue<double>(new double[] { 1.1, 2.2, 3.3 });
Console.WriteLine("Queue size before Dequeue: " + q.Count);
while (q.Count > 0)
    Console.WriteLine(q.Dequeue());
Console.WriteLine("Queue size after Dequeue: " + q.Count);
```

Now, let's examine the output from the code snippet.



```
Queue size before Dequeue: 3
1.1
2.2
3.3
Queue size after Dequeue: 0
```

We started with a Queue that had three elements, we looped through the Queue and displayed the output, and in doing so each iteration removed the displayed element from the Queue until the Queue was emptied and the loop was broken. We verified that the Queue was empty by looking at the number of elements left in the Queue which was zero.

Now let's look at the Peek method. The Peek method will retrieve only the first element in the Queue which means it retrieves the first element added to the Queue. However, unlike the Dequeue method, the element is not removed from the Queue. Simple enough right? Consider, the following example,

```
Queue<double> q = new Queue<double>(new double[] { 1.1, 2.2, 3.3 });
Console.WriteLine(q.Peek());
Console.WriteLine("Current number of elements in Queue: " + q.Count);
```

After running the code what element from the Queue do you expect to be displayed? Let's analyze the output.

```
1.1
Current number of elements in Queue: 3
```

Just as we would expect, the Peek method returned the first value-added to the Queue and the current number of elements in the Queue is still three. As such, we were able to retrieve the first element added to the Queue without removing any of the elements.

## Stacks

A Stack is a lot like a Queue except for this datatype it is LIFO (Last In First Out). In other words, a Stack is a Queue in reverse. This means the last or newest element added to the Stack is the first one retrieved. Much like a Queue a Stack, an empty stack is created the same way.

```
Stack<type> s = new Stack<type>();
```

The syntax for initializing a Stack at declaration is also the same as with a Queue.

```
Stack<type> s = new Stack<type>(new type[] { val1, val2, val3 });
```

Also, like Queues, Stacks have a Clear method and a count property that behave the same way as we've seen in the past. However, most of the associated Stack methods are different than the methods associated with Queues but serve the same underline function.

To start with let's look at the Push method. The Push method is Stack's version of the Queue's Enqueue method.

```
Stack<int> s = new Stack<int>(new int[] { 1, 2, 3 });
Console.WriteLine("Before Push Stack size: " + s.Count);
s.Push(4);
Console.WriteLine("After Push Stack size: " + s.Count);
```

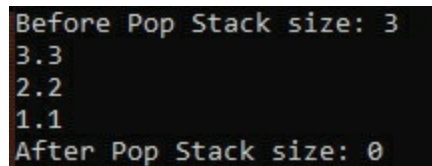
The code renders,

```
Before Push Stack size: 3
After Push Stack size: 4
```

As can be seen, the Stack's Push method behaves the same way as the Enqueue method for Queues.

Stacks also have a method similar to the Dequeue method. The Pop method behaves the same way the Dequeue method does. Much like Dequeue, Push will access the elements and then remove them from the Stack. However, since a Stack is LIFO, Push will access the newest element first and then remove it from the Stack. To demonstrate the Push method we're going to basically, use the same program we used to demonstrate the Dequeue method. We're going to create a preloaded Stack, use the count property to get the number of elements, display all the elements in the Stack using the Push method, and finally display the number of elements again. The resulting output should be relatively the same except in reverse order.

```
Stack<double> s = new Stack<double>(new double[] { 1.1, 2.2, 3.3 });
Console.WriteLine("Before Pop Stack size: " + s.Count);
while (s.Count > 0)
    Console.WriteLine(s.Pop());
Console.WriteLine("After Pop Stack size: " + s.Count);
```



```
Before Pop Stack size: 3
3.3
2.2
1.1
After Pop Stack size: 0
```

```
Stack<double> s = new Stack<double>(new double[] { 1.1, 2.2, 3.3 });
Console.WriteLine(s.Peek());
```

3.3

## HashSet

A HashSet is a lot like a List; however, unlike a List, duplicate entries are not allowed, and you cannot directly access elements like a List or array. A HashSet will automatically remove duplicate elements. You can add a duplicate entry but it will be removed as soon as it is added. Outside of automatically removing duplicates, there isn't much of a difference. You declare an empty HashSet the same way as a List, you initialize a HashSet at declaration the same way you would a List, the Add method is used the same way, the Clear method works the same way, and the count property behaves the same way as well. So, for the most part, if you know how to use a list you can use a HashSet. Consider the following code example,

```
HashSet<double> ht = new HashSet<double> { 1.1, 2.2, 3.3, 1.1 };
```

```
foreach (var x in ht)
    Console.WriteLine(x);
```

After examining the output we can see that the extra 1.1 was removed from the list.

```
1.1
2.2
3.3
```

There is also a slight difference in behavior when it comes to the remove method compared to its behavior with a List that should be noted. Since a HashSet does not have any duplicate elements the Remove method will completely remove the element for the collection. So, in a HashSet, the Remove method behaves more like a delete method.

## SortedList

A SortedList is another key-value pair similar to a dictionary. When working with a SortedList it is important to remember that there is a generic version and a non-generic version of the collection which will be covered later on. So, when you're working with a SortedList pay attention to the type of SortedList you are using. The difference between a SortedList and a dictionary is very simple. A generic SortedList will automatically sort the entries based on the keys in ascending order.

An empty SortedList is declared the same way a Dictionary is with the following syntax,

```
SortedList<Key_type, Value_type> sl = new SortedList<Key_type, Value_type>();
```

Also, a SortedList is initialized at declaration the same way.

```
SortedList<Key_type, Value_type> sl = new SortedList<Key_type, Value_type>
{
    {Key1, Value1 },
    {Key2, Value2 },
    {Key3, Value3 },
    {Key4, Value3 }
};
```

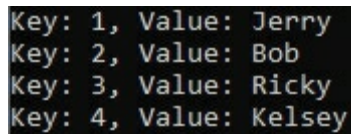
Outside, of the SortedList automatically sorting the keys in ascending order, the major methods and properties associated with SortedList are the same as with a Dictionary. Therefore, the Add method adds new key-value pairs to the SortedList, Clear operates the same, as does the count

property. Elements are also accessed the same way as with Dictionaries, with the key. SortedLists do not contain a KeyCollection nor a ValueCollection; however, the KeyValuePair struct operates the same way. Consider the following SortedList

```
SortedList<int, string> sl = new SortedList<int, string>
{
    {2, "Bob" },
    {1, "Jerry" },
    {4, "Kelsey" },
    {3, "Ricky" }
};

foreach (KeyValuePair<int, string> x in sl)
    Console.WriteLine("Key: {0}, Value: {1}", x.Key, x.Value);
```

Notice how the keys in the SortedList are in no particular order. If this were a dictionary the entries would stay in the same order that they were added, but since this is a SortedList they will be sorted. Let's look at what the code above outputs,



```
Key: 1, Value: Jerry
Key: 2, Value: Bob
Key: 3, Value: Ricky
Key: 4, Value: Kelsey
```

As can be seen, the entries are sorted based on the key's value and, as promised, in ascending order.

## Non-Generic Collections

Now that we've explored the generic collections let's explore non-generic collections. As stated before, a non-generic collection is not type-safe. This means, that a non-generic can accept entries of any datatype. Extreme caution should be used when using a non-generic collection due to the lack of type safety. Since non-generics can accept entries of any data type it is possible, and common, to accidentally add elements of unexpected data types. For this reason, generic collections are often favored over non-generics.

Since we've already explored generics, we've already gained a good understanding of the more commonly used non-generics.

- ArrayLists

- SortedList
- Stack
- Queue
- Hashtable

For these non-generic, there is a generic counterpart. Most of the methods and properties are the same between generics and non-generics as well.

Generics	Non-Generics
List	ArrayList
SortedList	SortedList
Stack	Stack
Queue	Queue
Dictionary	Hashtable

**Table 6.1**

For the sake of job searching, I would study this table and memorizing the generics and their non-generic counterparts. With all that being said let's look at these collections in action.

## ArrayList

ArrayList work the same way as Lists. Declaring an empty ArrayList is accomplished with the following syntax,

```
ArrayList al = new ArrayList();
```

As can be seen, there are no data type declarations. Declaring an ArrayList, or any non-generic for that matter is as easy as declaring a normal class object.

Initializing the ArrayList on declaration is done just like a List with the following syntax

```
ArrayList al = new ArrayList{ val1, val2, val3, ...};
```

Adding new elements to an ArrayList is just as simple as adding new elements to a List. All you have to do is use the Add method. However, since there is no type checking you can have multiple data types in the

collections. Consider the following example,

```
ArrayList al = new ArrayList();  
  
al.Add(1);  
al.Add("string");  
al.Add(3.3);  
  
foreach (var x in al)  
    Console.WriteLine(x);
```

```
1  
string  
3.3
```

As can be seen, we created an ArrayList and added three different elements with three different datatypes to it.

Accessing individual elements is also the same as with a List or an array. Suppose we want to display the element that contains a string, since ArrayLists are zero-index all we would have to do is use the following syntax,

```
Console.WriteLine(al[1]);
```

```
string
```

As can be seen, elements in an ArrayList are accessed individually in the same way as an array or List.

## SortedList

A non-generic and generic sorted list are essentially the same thing with the only difference between the two is that a generic SortedList can hold keys and values of any type where the keys and values of a generic SortedList are type specific. Since a generic SortedList and a non-generic SortedList are essentially the same thing with different typing rules you have to be careful and be clear of what type of SortedList you declare or are asked to declare. A non-generic SortedList is declared with the following syntax,

```
SortedList sl = new SortedList();
```

Adding an element to a SortedList is just like adding an element to a generic



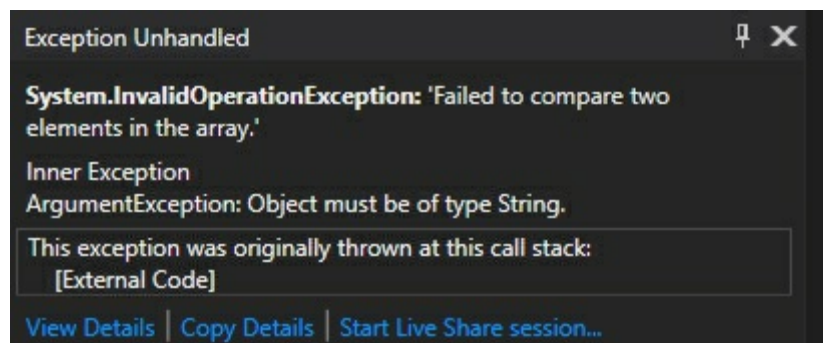
SortedList. As such, you can opt to add elements individually with the Add method or you can load the SortedList like the following,

```
SortedList sl = new SortedList();
sl.Add("manager", "Bob");
sl.Add("employee", "John");

SortedList sl2 = new SortedList()
{
    {"manager", "Bob" },
    {"employee", "John" }
};
```

Non-generic SortedLists do have a little gotcha that you need to keep in mind. Though you can use any datatypes you wish for the key and value, they all have to be consistent. The key's datatype is locked in when the first set is added. If we tried to do something like the following, we'll get an error,

```
SortedList sl2 = new SortedList()
{
    {"manager", "Bob" },
    { 1, "John" }
};
```



In this case, we started with a string for both the key and the value data type. In the next line, we tried to add an integer data type for the key and a string data type for the value. Now, this is only true for the key's data type, the data type for the value can be any type. For example, the following will run without issue,

```
SortedList sl2 = new SortedList()
{
```

```
{ "manager", "Bob" },  
{ "employee", 1 },  
{ "other", 22.3 }  
};
```

Accessing a SortedList is done with the following syntax,

```
SortedList_name[key_name]
```

Looping through a SortedList with a foreach can be accomplished with the following,

```
foreach(var x in SortedList_Name.Keys)  
    Console.WriteLine(sl[x]);
```

I've found it best when using a foreach loop to enumerate over a SortedList to use the var keyword to determine the SortedList key's data type. You don't have to use the var keyword; however, it does simplify things. Now, when trying to access the values as you would do under normal circumstances you want to grab the key like in the example.

Much like the generic version of the SortedList the entries are sorted by the keys. Consider the following,

```
SortedList sl2 = new SortedList()  
{  
    { "manager", "Bob" },  
    { "employee", 1 },  
    { "other", 22.3 }  
};  
  
foreach(var x in sl2.Keys)  
    Console.WriteLine(sl2[x]);
```

```
1  
Bob  
22.3
```

As can be seen, the values are sorted alphabetically by the keys.

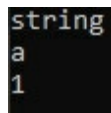
Much like the generic version of the SortedList all the methods are the same. Since we've already seen those in action we're going to move on to the next type of collection.

## Generic Stack

A generic stack is almost the same as a non-generic stack. Elements are retrieved in a LIFO manner the Push method adds an element to the stack while the Pop method removes an element from the stack. However, unlike a generic stack, the non-generic counterpart is not type specific and as such, each element that you add to the stack can have a different data type. For example, consider the following code snippet,

```
Stack s1 = new Stack();  
s1.Push(1);  
s1.Push('a');  
s1.Push("string");  
  
foreach (var x in s1)  
    Console.WriteLine(x);
```

If you study the data types of the elements that are being added to the stack s1 you'll see that we've added an integer, a char, and a string all with no problems. If you run this snippet, you'll get the following output,



```
string  
a  
1
```

Much like any of the other generics, we have explored thus far you can load the stack at initialization much like any other structure. As such, if you understand a generic stack you understand the non-generic counterpart.

## Queues

Much like generic stacks and their non-generic counterparts, non-generic Queues behave in much the same way as their generic counterparts. Much like the generic counterpart, the Enqueue method adds an element to the queue while the Dequeue removes an element from the queue. Consider the following example,

```
Queue q1 = new Queue();  
q1.Enqueue(1);  
q1.Enqueue('a');  
q1.Enqueue("string");  
  
foreach (var x in q1)  
    Console.WriteLine(x);
```

Running the above will produce the following,

```
1  
a  
string
```

As can be seen, the non-generic queue behaves just like its generic counterpart. The elements are retrieved in a FIFO manner.

## Hashtables

I don't use Hashtables a lot during my day-to-day programming. I don't like Hashtables due to their lack of type safety. Hashtables allow you to mix and match the datatypes of both the key and the value. Consider the following example,

```
Hashtable h1 = new Hashtable();  
h1.Add(1, "one");  
h1.Add("2", 2);  
  
Console.WriteLine(h1["2"]);  
Console.WriteLine(h1[1]);
```

After running the snippet should see the following,

```
2  
one
```

In short, much like the other non-generics, the Hashtable behaves the same as its generic counterpart.

# CHAPTER 7: GENERICS

## THE REALLY COOL STUFF

Thus far we have explored generic and non-generic collections, arrays, and so much more. You my lucky friend are well on your way to becoming an employed, tax-paying, productive member of society. However, there is still one more facet of C# that you must master on your way to gainful employment at a hopefully not mediocre company, and that is generics.

### **Generics Methods**

So, as I'm sure you've already figured, at least if you're older than two, is that most people can't make up their minds. Well you might think you're safe with programming but you're not, a lot of times programmers and users can't make up their minds either. Sometimes you have to pass a double to a method while other times you might have to pass one of those pesky integers. So, what's a hotshot to do? We've established we can't use the var keyword because that's illegal in C#. However, we could use the dynamic keyword which is a handy trick if you're passing around GUI elements such as buttons. The dynamic keyword has a drawback though, the variable's data type that was declared with the dynamic keyword is at risk of being changed accidentally. So, what are we to do? Well, enter the world of generics.

Generics are kind of weird at first and take a little getting used to, so to start let's create a new project and look at a simple example. To start with let's look at the syntax to declare a method that takes generic data types as arguments. When you declare a generic data type you can use any letter or series of letters you want, but it is common practice to use T to denote your first generic type. So, with that in mind let's make a method called to test and have it take in a variable x of type T as an argument. We're going to call this method from the Main method, so it'll have to be static as well. The syntax to accomplish this will be as follows,

```
public static void test<T>(T x)
```

Notice the T in the angle brackets. Those angle brackets tell C# that this method takes in variables of a yet-to-be-determined type. Also notice in the argument list that instead of placing a data type we're placing T. Again, this signals to C# that the arguments are going to take the data type of whatever T is. So, let's see this in action.

```
using System;

namespace GenericTest
{
    class Program
    {
        static void Main(string[] args)
        {
            test(1.2);

            public static void test<T>(T x)
            {
                Console.WriteLine(typeof(T));
            }
        }
    }
}
```

```
System.Double
```

As can be seen, we passed in a double and the type comes back as double. Now, what happens if we pass in an integer or a string?

```
using System;

namespace GenericTest
{
    class Program
    {
        static void Main(string[] args)
        {
            test(1.2);
            test(2);
            test("string");
        }

        public static void test<T>(T x)
```

```

    {
        Console.WriteLine(typeof(T));
    }
}

```

```

System.Double
System.Int32
System.String

```

Well, it looks like the argument's datatype was matched to the data type of whatever we passed in.

So, this is fine and dandy, but what if we have more than one data type? Well, we can simple add a U type in the angle brackets like the following,

```

using System;

namespace GenericTest
{
    class Program
    {
        static void Main(string[] args)
        {
            test(1.2 , 2);
            test(2, "string");
            test('c', 1.2f);
        }

        public static void test<T, U>(T x, U y)
        {
            Console.WriteLine("type x: {0}, type y: {1}", typeof(T), typeof(U));
        }
    }
}

```

```

type x: System.Double, type y: System.Int32
type x: System.Int32, type y: System.String
type x: System.Char, type y: System.Single

```

As can be seen, by adding the U to the generics list we are now able to pass in two different data types.

So, this is great, but when would you use this in the real world?

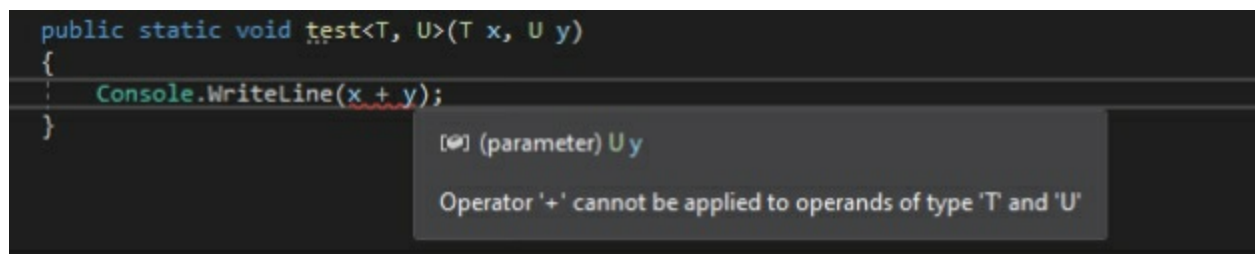
There are a lot of answers to that question. Generally, you don't use generics for math, for example suppose you want to add two generic types like the following

```
using System;

namespace GenericTest
{
    class Program
    {
        static void Main(string[] args)
        {
            test(2, "hired");
        }

        public static void test<T, U>(T x, U y)
        {
            Console.WriteLine(x + y);
        }
    }
}
```

If you try to type in the above code you're going to get the following,



```
public static void test<T, U>(T x, U y)
{
    Console.WriteLine(x + y);
}
```

[?] (parameter) U y

Operator '+' cannot be applied to operands of type 'T' and 'U'

You get this error because C# doesn't know that what the types for T and U are. For all C# knows you might be asking it to add a number to an impossible string.

Circling back to the original question what are generic types good for when you can't even do math with them? Well, the answer to that is it's up to you; however, I liked to use generics when I needed to do things like pass different structs or classes to methods for operations such as XML serialization or other type neutral operations.

## Generic classes



Much like methods, a class can also be passed a generic type and that type will reflect across the whole class. This may seem a bit odd at first but think of generics collections. Collections such as Lists are, at their hearts, nothing more than generic classes.

Declaring a generic class is quite simple, all you have to do is include <T> after the class name. Consider the following example

```
using System;
using System.Collections.Generic;
using System.Text;

namespace chopShop
{
    class GenericClass<T>
    {
    }
}
```

Creating an object to a generic class is exactly like creating an object to a List or other generic collection. Consider the following,

```
GenericClass<int> gc = new GenericClass<int>();
```

In this case, we are saying that for the reference gc anywhere C# sees a T in the GenericClass to assume that the data type is an int. Let's demonstrate this by creating a method call dataType and an extra object called gc2. When all is said and done your code should look like the following,

```
using System;

namespace chopShop
{
    class GenericClass<T>
    {
        public void DataType<T>(T x)
        {
            Console.WriteLine(typeof(T));
        }
    }
}

using System;
```

```

namespace chopShop
{
    class Program
    {
        static void Main(string[] args)
        {
            GenericClass<int> gc = new GenericClass<int>();
            GenericClass<string> gc2 = new GenericClass<string>();
            gc.DataType(2);
            gc2.DataType("string");
        }
    }
}

```

Once you run the Main method, you'll get the following output,

```

System.Int32
System.String

```

As can be seen, it behaves like a normal generic method.

## Conclusion

As of now, if you understand all the information in the book you should have a pretty good grasp on the more advanced concepts of the C# programming language and more importantly programming in general. Now, as I have stated before C# is the language that keeps on giving, and as such you should consider you're journey as a C# programmer, and a programmer in general, just beginning. What I will say is this book was more or less just a crash course into the more advanced topics you will face in your day to day life as a programmer. However, what I hope you gain from this book is a good backbone into C# and programming in general so you can quit your job and no longer have to work as a drunk loser and hopefully get a better job where only have to work for your standard drunk.