README

Objects

What is Object Oriented Programming (OOP)

Objects, as you might have guessed, are the foundation of OOP, Object-Oriented Programming. They allow us to create reusable, modular code that makes our projects more comfortable to maintain and extend. But how is that different from well-composed functions and procedural programming?

OOP and the creation of objects, bundles data, and the behaviors that modify that data into neat packages called objects, whereas, in procedural programming, the data and the behaviors that modify data do not belong to a single data structure. Consider the follow procedural programming example:

```
using System;

class Program
{
    static void Main()
    {
        Player[] players = new Player[3];
        Random random = new Random();

        players[0] = new Player {Name = "Ollie", X = 0, Y = 0};
        players[1] = new Player {Name = "Van", X = 0, Y = 0};
        players[2] = new Player {Name = "Lucy", X = 0, Y = 0};

        for (int i = 0; i < players.Length; i++)
        {
            MovePlayer(players[i], (X: random.Next(0, 10),
Y:random.Next(0, 10)));
            Console.WriteLine(players[i]);
        }
    }

    /**
     * MovePlayer is a method defined inside our Program class that
takes two arguments:
     * The player we wish to move, and the new coordinates, as a
```

```
Tuple, to which we want
     * to move the player.
     *
     * @see https://docs.microsoft.com/en-us/dotnet/csharp/tuples
     */
    static void MovePlayer(Player player, (int X, int Y) coords)
    {
        player.X = coords.X;
        player.Y = coords.Y;
    }
}

/**
 * Player class contains data that defines what a player is.
 * for our purposes, the player has a name and X and Y coordinates
 * that represent the position the player occupies on a game board
 */
class Player
{
    public string Name;
    public int X;
    public int Y;

    public override string ToString()
    {
        return $"Player {Name} is at the location {X}, {Y}";
    }
}
```

The class player has data that defines what the player is:

- Name
- X coordinate
- Y coordinate

However, the behavior of moving a player is defined outside of the player class. This is an example of procedural programming. To this point in our class, most of you have been writing procedural code.

From here on out, we will work on bundling our data and the behaviors that modify that data into a single class to help make our classes more reusable and maintainable.

A more object-oriented approach to creating the Player class would be to define the MovePlayer method inside of the Player class as follows:

```
class Player
{
    public string Name;
    public int X;
    public int Y;

    public void MovePlayer(int x, int y)
    {
        /**
         * Debug.Assert() runs only when the code is being executed
in
         * debug mode and can be used to catch conditions in your
code
         * that might cause bugs. In our case, the game board is
10x10
         * and any value of x or y outside 0-9 is invalid.
         *
         * @see https://docs.microsoft.com/en-
us/dotnet/api/system.diagnostics.debug.assert?view=netcore-3.1
         */
        Debug.Assert(x >= 0 && y < 10);
        Debug.Assert(y >= 0 && y < 10);
        X = x;
        Y = y;
    }

    public override string ToString()
    {
        return $"Player {Name} is at the location {X}, {Y}";
    }
}
```

Try to incorportate the new player class into the program above.

1. What changed in your code?
2. Are there advantages you can see the new Player class?
3. Assume you were writing a library that was meant to be used by several board games at your company. Each game had the concept of a Player with a name and coordinates. What would you have to do in each new board game if a Player could not move itself?
4. Can you think of any disadvantages to a Player being able to move itself?

What is an object?

Objects are data abstractions that capture:

1. an internal representation through data members. In C# these data members are called properties.
   - it might help to think of properties as member variables; meaning, variables that are members of the class in which they are defined. A member variable cannot be accessed without first referencing the containing class.
2. an interface for interacting with the object. In C# this interface is implemented with methods.
   - it might help to think of methods as member functions; meaning, functions that are members of the function in which they are defined. A member function cannot be accessed without first referencing the containing class.

Objects have a type, data, and sometimes behaviors to modify the data:

```
/**
 * The following class definition has the type BaseballPlayer.
 * We say that any object created using this class is 'of type
BaseballPlayer'
 */
class BaseballPlayer
{
    /**
     * Properties that represent the data that defines what a
BaseballPlayer is.
```

```
     */
    public string Name;
    public int AtBats;
    public int Hits;

    /**
     * The interface for interacting with the BaseballPlayer
implemented as methods.
     * SwingAtPitch does not represent the actual game of baseball.
This method assumes
     * that a batter only gets one pitch before we increment his/her
stats.
     *
     * @see https://en.wikipedia.org/wiki/Baseball_rules#Batting
     */
    public bool SwingAtPitch()
    {
        Random random = new Random();
        float battingAverage = GetBattingAverage();
        bool isHit = random.NextDouble() <= battingAverage;
        IncrementStats(isHit);
        return isHit;
    }

    public float GetBattingAverage()
    {
        return Hits/AtBats;
    }

    private void IncrementStats(bool isHit)
    {
        ++AtBats;
        if (isHit) {
            ++Hits;
        }
    }
}
```

Programmers create new types by defining a class. The class definition is a blueprint from which we can create new instances of an object. We can be certain that every new object instance has the same members (properties and methods).

After creating (or instantiating) a new object, we can assign values to, and access values of objects members using the member access expression (dot notation).

```
/**
 * Class definition.
 */
class BaseballPlayer
{
    //class implementation
}

/**
 * Object instantiation using the BaseballPlayer blueprint.
 */
BaseballPlayer catcher = new BaseballPlayer();

/**
 * Property assignment using the member access expression
 */
catcher.Name = "Buster Posey";
catcher.AtBats = 4575;
catcher.Hits = 1380;

/**
 * Interface usage using the member access expression
 */
catcher.GetBattingAverage();
```

This example works but it's a little clunky. We are creating a new object and then assigning property values. We can do this in one step by defining a constructor inside of our BaseballPlayer class.

A constructor is called called everytime a new obect is instantiated. We can use the constructor to initalize the values of the object properties. The modified class definition and usage looks like:

```
class BaseballPlayer
{
```

```
    public string Name;
    public int AtBats;
    public int Hits;

    public BaseballPlayer(string name, int atBats, int hits)
    {
        Name = name;
        AtBats = atBats;
        Hits = hits;
    }

    //Remaining class implementation
}

BaseballPlayer catcher = new BaseballPlayer("Buster Posey", 4575,
1380);
catcher.GetBattingAverage();
```

The base Object class and overriding methods to make your classes more meaningful

Every object in C# inherits functionality from a base class called the Object class. We will discuss inheritance later this week. For today, just know that any class we create can access methods defined in the base Object class. The methods in the Object class are defined as `virtual` methods; meaning, that any class derived from the Object class (all classes) can `override` the functionality of the virtual methods. Lets see what this means instead of talking about it more.

```
using System;

namespace Baseball
{
    class Game
    {
        static void Main()
        {
            BaseballPlayer shortStop = new BaseballPlayer("Buster
Posey", 4575, 1380);
```

```
            Console.WriteLine(shortStop);
            //expected output: Baseball.BaseballPlayer: Buster Posey
has hit safely in 1380 of his/her 4575 AB's.
        }
    }

    /**
     * Notice the colon followed by Object.
     * This is how you would explicitly define that BaseballPlayer is
derived from Object.
     * However, because all C# classes are derived from Object you do
not need to explicitly define
     * the relationship. It is implied that BaseballPlayer is a
derivative of Object; therefore, it
     * is best practice to leave out the colon and Object. I have
chosen to explicitly define the
     * the relationship here to highlight that ToString is a virtual
method defined in the Object class.
     */
    class BaseballPlayer : Object
    {
        public string Name;
        public int AtBats;
        public int Hits;

        public BaseballPlayer(string name, int atBats, int hits)
        {
            Name = name;
            AtBats = atBats;
            Hits = hits;
        }

        public float GetBattingAverage()
        {
            return Hits / AtBats;
        }

        /**
        * The override keyword is used to extend the functionality
        * of the Object classes ToString method. Here we are calling
the
        * Object classes ToString method by accessing it through the
```

```
'base' keyword.
        * The Object classes ToString method returns the fully
qualified name of the class; meaning,
        * the namespace and classname in the format
namespace.classname.
        */
        public override string ToString()
        {
            return $"{base.ToString()}: {Name} has hit safely in
{Hits} of his/her {AtBats} AB's.";
        }
    }
}
```

In the coming days, we will discuss, use, and override virtual methods as we more deeply explore object inheritance.

Look up the documentation for the Object class

1. What other virtual methods are defined in the object class?
2. What does the GetHashCode method do and how does it help determine equality?