#### GIVE IT A TRY #1

#### Start with a new console project.

#### Change the using statement at the top of the file from System.Collections.Generic to System.Collections.

#### Create a new class named ClientRecord which implements the IEqualityComparer interface.

#### This interface has two methods: public bool Equals (object a, object b) and public int GetHashCode (object o).

#### Two member variables: clientName of type string and clientId of type int. Create two properties for these variables.

#### In the set accessor for clientName, make sure that value is not null and not the empty string.

#### In the set accessor for clientId, make sure that value is greater than 0.

#### Override the ToString() method to display the clientId followed by the clientName (e.g. "15: Kevin");

#### Implement the Equals() and GetHashCode() methods.

#### For Equals() assume that if either or both objects are null, they’re not equal.

#### For GetHashCode(), just call the clientId.GetHashCode() method.

#### Write code to test the program. Ensure equal objects have the same hash codes.

#### Bonus: write a polymorphic method to compare IEqualityComparer objects.

#### EXERCISE #1

|  |  |
| --- | --- |
| **WHAT TO DO** | **HOW TO DO IT** |
| Create an interface and then implement it with two different classes. | The **IFormatter** interface is very simply. It contains a single method called **Format** that takes no arguments and returns **void**.  The **PhoneNumber** class implements the **IFormatter** interface. Phone numbers will have the format of **(NNN) NNN-NNNN** where **N** is a digit. This class will have the following features:   * A **private** member variable of type **int** called **areaCode**. * A **private** member variable of type **int** called **exchangePrefix**. * A **private** member variable of type **int** called **phoneNumber**. * A **private** member variable of type **string** called **formattedPhoneNumber**. * A constructor that takes a single **string** of digits (NNNNNNNNNN) and parses it apart into the three components that make up a phone number. These parts will be stored in the **private** member variables **areaCode**, **exchangePrefix**, and **phoneNumber**. Also, set **formattedPhoneNumber** to **null**.   + To parse a **string**, use the **Substring()** function of the **string**. The first parameter is the starting index in the string (remember strings a 0-based) and the second parameter is the number of characters to parse. For example, if you wanted to parse characters 7 through 10 of a string named **sampleString**, you would write a statement like **sampleString.Substring(6, 4);**. Once you’ve parsed out the target sub-substring, call the **int.Parse()** method passing that number in to get the integer value. For example, **int n = int.Parse(sampleString.Substring(6,4));** * Implement the **Format()** method as **public** and set the **formattedPhoneNumber** to the appropriate format using the **private** member variables. * Override the **ToString()** method. First check to see if **formattedPhoneNumber** is **null**. If it is, call the **Format()** method. Then return **formattedPhoneNumber** to the caller.   The **SocialSecurityNumber** class also implements the **IFormatter** interface. Social Security Numbers will have the format of **NNN – NN – NNNN** where **N** is a digit. This class will have the following features:   * A **private** member variable of type **int** called **groupOne**. * A **private** member variable of type **int** called **groupTwo**. * A **private** member variable of type **int** called **groupThree**. * A **private** member variable of type **string** called **formattedSsn**. * A constructor that takes a single **string** of digits (NNNNNNNNN) and parses it apart into the three components that make up a social security number. These parts will be stored in the **private** member variables **groupOne**, **groupTwo**, and **groupThree**. Also, set **formattedSsn** to **null**. * Implement the **Format()** method as **public** and set the **formattedSsn** to the appropriate format using the **private** member variables. * Override the **ToString()** method. First check to see if **formattedSsn** is **null**. If it is, call the **Format()** method. Then return **formattedSsn** to the caller.   The **TestInterface** class will test the **SocialSecurityNumber** and **PhoneNumber** classes discussed above. This class will have the following features:   * A **private** **static** method called **FormatRecord** that takes an argument of type **IFormatter** and returns **void**. This method will call the argument’s **Format()** method and then return to the caller. * A **private** **static** method called **GetRecordType** that takes no arguments and returns a **string**. This method will display the menu to the user and obtain their choice. The choice is returned to the caller. Menu options are 1 = create a phone number record, 2 = create a social security record, and 0 = exit. * The **Main()** method which will declare two variables. The first is a local variable of type **bool** named **keepGoing** that is set to **true**. This variable will be used to run the **while** loop. The second is a **string** called **userChoice**. * The **Main()** method will have **while** loop that checks if **keepGoing** is **true**. If it is, the **while** loop will do the following in order:   + Call **GetRecordType()** to display the menu and get the user’s choice.   + Use a **switch** statement on the user’s choice.   + For choice 1 prompt the user for a **string** that represents the phone number, create a new object of type **PhoneNumber**, call **FormatRecord()** with this object, and display a message showing the formatted phone number.   + For choice 2, prompt the user for a **string** that represents the social security number, create a new object of type **SocialSecurityNumber**, call **FormatRecord()** with this object, and display a message showing the formatted social security number.   + For choice 0, set **keepGoing** variable to **false**. This will force the **while** loop to end. |

#### EXERCISE #2

|  |  |
| --- | --- |
| **WHAT TO DO** | **HOW TO DO IT** |
| Implement the **ICloneable** interface in two classes and demonstrate polymorphism using the interface. | Introduction:  The **ICloneable** interface can be found in the **System** namespace. It contains a single method named **Clone**. The purpose of this method is to copy an existing object to a new object. If any changes are made to one of the instances, the other instance will retain its original values. The signature is:  **Object Clone();**  This exercise will instruct you on how to create two classes that implement this interface. You will also create a class to test the interface by using polymorphism.  Create a class named **Square** implementing the **ICloneable** interface. This class will have the following members:   * A **private** member variable named **\_side** of type **double**. (Note that it is common practice to name class member variables with a preceding underscore to distinguish them from local variables.) * A constructor that takes a **double** and saves it by calling **Side**. (This property will be implemented in the next step). * A read/write property named **Side** that allows access to **\_side**. This property returns a **double**. * A read-only property named **Perimeter** that returns the perimeter of the square. To compute the perimeter: **Side \* 4**. This property returns a **double**. * A read-only property named **Area** that returns the area of the square. To compute the area: **Side \* Side**. This property returns a **double**. * The **public** **Clone()** method that returns an **object**. This method will do two things.   + It will create a new **Square** instance passing in **this.Side** into the constructor.   + It will return the new **Square** instance.   Create a class named **Circle** implementing the **ICloneable** interface. This class will have the following members:   * A **private** member variable named **\_radius** of type **double**. * A constructor that takes a **double** and saves it by calling **Radius**. (This property will be implemented in the next step). * A read/write property named **Radius** that allows access to **\_radius**. This property returns a **double**. * A read-only property named **Circumference** that returns the circumference of the circle. To compute the circumference: **(Radius\*2) \* Math.PI**. This property returns a **double**. * A read-only property named **Area** that returns the area of the circle. To compute the area: **(Radius\*Radius) \* Math.PI**. This property returns a **double**. * The **public** **Clone()** method that returns an **object**. This method will do two things.   + It will create a new **Circle** instance passing in **this.Radius** into the constructor.   + It will return the new **Circle** instance.   Create a class named **TestCloning**, which will contain the **Main()** method. This class will have the following members:   * A **private** **static** method named **Copy** that takes a single argument of type **ICloneable** and returns an **object**. This method will call the **Clone()** method on the argument that was passed in. The **Clone()** method returns an **object**. Then this method will return the **object** back to the caller. * Implement the **Main()** method:   + Create a **Square** object providing the size of one of its sides.   + Display the square’s side, perimeter, and area.   + Clone the square by calling the **Copy()** method. This will return an **object**. You will need to cast this **object** to a new **Square()**.   + Display the copied square’s side, perimeter, and area.   + Change the side of one of the squares and display the data of both again.   + Following the same procedure for creating two **Circle** objects. Remember to cast the **object** returned from **Copy()** to a new **Circle**.   Notice that the **Copy()** method will take any object that implements the **ICloneable** interface. You can create more classes that implement this interface and pass them all to **Copy()**. |