Collections/Generics

Collection Types are specialized classes for data storage and retrieval.

- These classes provide support for stacks, queues, lists, and hash tables.
- Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc.

Namespaces:

- System.Collection

e.g. ArrayList, Hashtable, SortedList, Stack, Queue

- System.Collection.Generic

- Generic collection is strongly typed (type safe), that you can only put one type of object into it.
- This eliminates type mismatches at runtime, giving better performance.

e.g. List, Dictionary

public List<Person> Children = new List<Person>();

List<Person> is read aloud as "list of Person "
We must ensure the collection is initialized to a new instance of a list of Person before we can add items to it otherwise the field will be null and it will throw runtime exceptions.

TasK: Print the names of bob's children using foreach loop.

Passing parameters to methods/overloading

Methods can have parameters passed to them to change their behavior.

```
public string SayHello()
{
 return $"{Name} says 'Hello!'";
}

public string SayHelloTo(string name)
{
 return $"{Name} says 'Hello {name}!'";
}
WriteLine(bob.SayHello());
WriteLine(bob.SayHelloTo("Emily"));
```

We can change the name of the second method as SayHello (string name) and still get the same output. This is called **method overloading**. This is allowed because the methods each have a different signature. A method signature is a list of parameter types that can be passed when calling the method (as well as the type of the return value).You can see the Console.WriteLine() method having different output depending on the parameters passed to it.

Partial Classes

- In C#, a class definition can be divided over multiple files.
- It is helpful for large classes with many methods.
- Used by Microsoft in some cases to separate automatically generated code from user written code.
- If class definition is divided over multiple files, each part is declared as a partial class.

Imagine we want to add statements to the Person class that are automatically generated by a tool like an object-relational mapper that

reads schema information from a database. If the class is defined as partial, then we can split the class into an auto-generated code file and a manually edited code file.

e.g.

Change the name of the person class as: public partial class Person

Add a new cs file named ‘PersonAutoGen.cs’ partial class file, type the following:

```
//properties and indexers

namespace OopClassLib
{
 public partial class Person

// a property defined using C# 1 - 5 syntax
public string Origin
{
 get
 {
 return $"{Name} was born on {HomePlanet}";
 }
}
// two properties defined using C# 6+ lambda expression syntax
```

```
public string Greeting => $"{Name} says 'Hello!'";

public int Age => System.DateTime.Today.Year - DateOfBirth.Year;

}
```

A property is simply a method (or a pair of methods) that acts and looks like a field when you want to get or set a value, thereby simplifying the syntax.

Defining settable properties

public string FavoriteIceCream { get; set; } // auto-syntax

Although you have not manually created a field to store the person's favorite ice cream, it is there, automatically created by the compiler for you. Sometimes, you need more control over what happens when a property is set. In this scenario, you must use a more detailed syntax and manually create a private field to store the value for the property.

e.g.

```
private string favoritePrimaryColor;

public string FavoritePrimaryColor

{

 get

 {

 return favoritePrimaryColor;

 }

 set
```

```
{
switch (value.ToLower())
{
case "red":
case "green":
case "blue":
favoritePrimaryColor = value;
break;
default:
throw new System.ArgumentException(
$"{value} is not a primary color. " +
"Choose from: red, green, blue.");
```

Now, in the main method of program.cs file, type:

```
sam.FavoriteIceCream = "Chocolate Fudge";
WriteLine($"Sam's favorite ice-cream flavor is {sam.
FavoriteIceCream}.");
sam.FavoritePrimaryColor = "Red";
WriteLine($"Sam's favorite primary color is {sam.
FavoritePrimaryColor}.");
```

Indexers allow the calling code to use the array syntax to access a property. For example, the string type defines an indexer so that the calling code can access individual characters in the string individually.

```
// indexers in Children object
public Person this[int index]
{
 get
 {
 return Children[index];
 }
 set
 {
 Children[index] = value;
 }
}
```

In main method:

```
sam.Children.Add(new Person { Name = "Charlie" });
sam.Children.Add(new Person { Name = "Ella" });
WriteLine($"Sam's first child is {sam.Children[0].Name}");
WriteLine($"Sam's second child is {sam.Children[1].Name}");
WriteLine($"Sam's first child is {sam[0].Name}");
WriteLine($"Sam's second child is {sam[1].Name}");
```

Good Practice: Only use indexers if it makes sense to use the square bracket, also known as array syntax. As you can see from the preceding example, indexers rarely add much value.

Delegates and Events

Events are often described as actions that happen to an object. For example, in a user interface, Button has a Click event, click being something that happens to a button. Another way of thinking of events is that they provide a way of exchanging messages between two objects.

Events are built on delegates.

One way to call a methos is via its name with passing arguments. The other way to call or execute a method is to use a delegate. If you have used languages that support function pointers, then think of a delegate as being a type safe method pointer. In other words, a delegate contains the memory address of a method that matches the same signature as the delegate so that it can be called safely with the correct parameter types.

For example, imagine there is a method in the Person class that must have a string type passed as its only parameter, and it returns an int type,

```
public int MethodIWantToCall(string input)

{

 return input.Length; // it doesn't matter what this does

}
```

We can call this method on an instance of Person named p1 like this:

```
int answer = p1.MethodIWantToCall("Frog");
```

Alternatively, we can define a delegate with a matching signature to call the method indirectly. Note that the names of the parameters do not have to match. Only the types of parameters and return values must match, as shown in the following code:

```
delegate int DelegateWithMatchingSignature(string s);
```

Now, we can create an instance of the delegate, point it at the method, and finally, call the delegate (which calls the method!), as shown in the following code:

```
// create a delegate instance that points to the method
var d = new DelegateWithMatchingSignature(p1.MethodIWantToCall);
// call the delegate, which calls the method
int answer2 = d("Frog");
```

You are probably thinking, "What's the point of that?" Well, it provides flexibility.

For example, we could use delegates to create a queue of methods that need to be called in order. Queuing actions that need to be performed is common in services to provide improved scalability.

Another example is to allow multiple actions to perform in parallel. Delegates have built-in support for asynchronous operations that run on a different thread, and that can provide improved responsiveness.

Inheritance

Code reusability feature provided via inheritance. New Classes called derived classes are created from existing classes called base/super classes.

e.g.

```
public class Class A
{
}
public class Class B : A
```

{

}

Create a new derived class Employee as below:

```
public class Employee : Person //syntax to inherit from base class
{
        }
```

In the main method:

```
Employee john = new Employee
{ Name = "John Jones",
DateOfBirth = new DateTime(1990, 7, 28) };
john.WriteToConsole();
```

Output: John Jones was hired on 23/11/14

Note that the Employee class has inherited all the members of Person.

Hiding members

So far, the WriteToConsole() method is being inherited from Person, and it only outputs the employee's name and date of birth as defined in the Person class. We might want to change what this method does for an employee:

{

```
public string EmployeeCode { get; set; }
public DateTime HireDate { get; set; }
public void WriteToConsole()
{
WriteLine(format:
"{0} was born on {1:dd/MM/yy} and hired on {2:dd/MM/yy}",
arg0: Name,
arg1: DateOfBirth,
arg2: HireDate);
}
}
```

Write in the main method:

```
Employee john = new Employee
{
 Name = "John Jones",
 DateOfBirth = new DateTime(1990, 7, 28)
};
john.WriteToConsole();
```

Output: John Jones was born on 28/07/90 and hired on 01/01/01

Overriding members

Rather than hiding a method, it is usually better to override it. You can only override if the base class chooses to allow overriding, by applying the **virtual/override** keyword:

Write in main method:

WriteLine(john.ToString());

Output: OopClassLib.Employee

The ToString method is inherited from System.Object so the implementation returns the namespace and type name.

```
// overridden methods in Person class
public override string ToString()
{
 return $"{Name} is a {base.ToString()}";  //base allows access to the derived class
}
```

Sealed Class/Method

You can prevent someone from inheriting from your class by applying the sealed keyword to its definition. No one can inherit from Scrooge McDuck

```
public sealed class ScroogeMcDuck
{
}
```

An example of sealed in .NET Core is the string class. Microsoft has implemented some extreme optimizations inside the string class that could be negatively affected by your inheritance; so, Microsoft prevents that.

You can prevent someone from further overriding a virtual method in your class by applying the sealed keyword to the method. No one can change the way Lady Gaga sings, as shown in the following code:

using static System.Console;

```
namespace OopClassLib
{
 public class Singer
 {
 // virtual allows this method to be overridden
 public virtual void Sing()
 {
 WriteLine("Singing...");
 }
```

```
public class LadyGaga : Singer
 {
 // sealed prevents overriding the method in subclasses
 public sealed override void Sing()
 {
 WriteLine("Singing with style...");
  }
 }
```

Polymorphism

Two ways to change the behavior of an inherited method- We can

hide it using the new keyword (known as non-polymorphic inheritance), or we can override it (known as polymorphic inheritance).

In the Employee class, add statements to override the ToString method

so it writes the employee's name and code to the console.

```
public override string ToString()
{
 return $"{Name}'s code is {EmployeeCode}";
}
```

In the main method:

```
Employee aliceInEmployee = new Employee
 { Name = "Alice", EmployeeCode = "AA123" };
Person aliceInPerson = aliceInEmployee;
aliceInEmployee.WriteToConsole();
aliceInPerson.WriteToConsole();
WriteLine(aliceInEmployee.ToString());
WriteLine(aliceInPerson.ToString());
```

When a method is hidden with new, the compiler is not smart enough to know that the object is an Employee, so it calls the WriteToConsole method in Person.

When a method is overridden with virtual and override, the compiler is smart

enough to know that although the variable is declared as a Person class, the object itself is an Employee class and, therefore, the Employee implementation of ToString is called.

Implicit casting

An instance of a derived type can be stored in a variable of its base type (or its base's base type, and so on). When we do this, it is called implicit casting.

Person aliceInPerson = aliceInEmployee;

Explicit casting: Going the other way is an explicit cast, and you must use parentheses around the type you want to cast into as a prefix to do it:

Employee explicitAlice = aliceInPerson; //not allowed

Employee explicitAlice = (Employee)aliceInPerson; //allowed

Interfaces:

Interfaces are a way of connecting different types together to make new things.

Think of them like the studs on top of LEGO™ bricks, which allow them to "stick" together, or electrical standards for plugs and sockets.

If a type implements an interface, then it is making a promise to the rest of .NET that it supports a certain feature.

Common interfaces

Here are some common interfaces that your types might need to implement:

Interface	Method(s)	Description
`IComparable`	`CompareTo(other)`	This defines a comparison method that a type implements to order or sort its instances.
`IComparer`	`Compare( first, second)`	This defines a comparison method that a secondary type implements to order or sort instances of a primary type.
`IDisposable`	`Dispose()`	This defines a disposal method to release unmanaged resources more efficiently than waiting for a finalizer.
`IFormattable`	`ToString( format, culture)`	This defines a culture-aware method to format the value of an object into a string representation.
`IFormatter`	`Serialize( stream, object)` and `Deserialize(stream)`	This defines methods to convert an object to and from a stream of bytes for storage or transfer.
`IFormat Provider`	`GetFormat(type)`	This defines a method to format inputs based on a language and region.

Comparing objects of a class type:

1. In the Main method, add statements that create an array of Person instances and writes the items to the console, and then attempts to sort the array and writes the items to the console again, as shown in the following code:

```
Person[] people =
{
 new Person { Name = "Simon" },
 new Person { Name = "Jenny" },
 new Person { Name = "Adam" },
 new Person { Name = "Richard" }
};
WriteLine("Initial list of people:");
foreach (var person in people)
{
 WriteLine($"{person.Name}");
}
WriteLine("Use Person's IComparable implementation to
sort:");
Array.Sort(people);
foreach (var person in people)
{
 WriteLine($"{person.Name}");
}
```

2. Run the application, and you will see the following runtime error:

Unhandled Exception: System.InvalidOperationException: Failed

to compare two elements in the array. --->

System.ArgumentException: At least one object must implement

IComparable.

As the error explains, to fix the problem, our type must implement

IComparable.

3. In the Person class, after the class name, add a colon and enter IComparable<Person>, as shown in the following code:

public class Person : IComparable<Person>

Visual Studio will draw a red squiggle under the new code to warn you

that you have not yet implemented the method you have promised to. It can

write the skeleton implementation for you if you click on the light bulb and

choose the Implement interface option.

Change the skeleton code as below:

```
public int CompareTo(Person other)
{
 return Name.CompareTo(other.Name);
}
```

Good Practice: If anyone will want to sort an array or collection of

instances of your type, then implement the IComparable interface.

A language feature introduced in C# 8.0 is **default implementations** for an interface.

e.g.

```
public interface IPlayable
 {
 void Play();

 void Pause();
 }
```

Implementation:

```
public class DvdPlayer : IPlayable
 {
 public void Pause()
 {
 WriteLine("DVD player is pausing.");
 }
 public void Play()
 {
 WriteLine("DVD player is playing.");
 }
 }
```