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.

Inheritance

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

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