Advanced Data Types

C# Programming

Arrays and Collections

There are limitations with arrays ArrayLists

- Arrays are generally Fixed Length
  - Need to be managed
  - Manual resizing
  - Manual re-arrangement

ArrayList

.NET Framework 1.0 shipped with ArrayList

- Any type can be added to an ArrayList
  - All types inherit from System.Object
- But ...
  - No type safety
  - Boxing and Unboxing of Value Types hits performance

```
ArrayList names = new ArrayList() {"Fred", "Mina", "Bob", 99};
```

- Can you spot the problem?

Generics

Allow Data Types to be defined

- Without pre-defining the types they use or work with
- Containing generic algorithms that apply to any type

Generics can apply to

- Data Types
- Functions
- Interfaces
- Delegates

Framework libraries Generics

Many classic generic types are part of System.Collections.Generic

- List<T>
- Dictionary<TKey, TValue>
- Stack<T>

Generic List

Generics use one or more type placeholders eg T

```
List<string> names = new List<string> {
    "Fred", "Amy", "Mina", "Tam", "Baz"
};
```

- Specifies a List of strings
- Populated with names

```
names.Add("Dhami"); // Type safe! Add expects a string argument
```

Generics are Type Safe

```
List<string> names = new List<string> {
    "Fred", "Amy", "Mina", "Tam", "Baz", 99
}; // Compile Error!! - Can't Add int to String List
```

- Specifies a List of strings
- Populated with names

```
names.Add(99); // Compile Error!! - Invalid Argument
```

Dictionary

Dictionary is a hash table of Key-Value pairs

- Elements not stored using linear indexes
- Hashes are calculated based on keys

```
Dictionary<string, Car> cars = new Dictionary<string, Car>();
cars.Add("ABC123", new Car("Audi", "R8"));
```

Elements can be retrieved by key

```
> Car myCar = cars["ABC123"];
> myCar.Make
'Audi'
```

Static Data

Most Data declared in a type is instance data

- A type instance must be created first
- Each instance holds its own instance data

Data can also be defined as static

- static data is shared among all instances of the type
- Accessible from the type name

Static Data - Example

Below there are two variables defined

- nextReg - static
- registration - instance

```
class Car {
    public static int nextReg;
    public string registration;
    public Car() {
        registration = $"ABC{++nextReg:000}";
    }
}
```

Accessing Instance Data

To access instance variables

- Create an instance of a type
- Access the public fields of the instance of the type

```
> Car myCar = new Car();
> myCar.registration
'ABC001'
```

- We had to create an instance of Car first!

Accessing Static Data

To access static variables

- Just access the public fields of the of the type itself

```
> Car.nextReg;
1
```

- We just used the name of the class
- No instance of Car had to be created!

Tuples

Tuples group data items together

- Simple, Lightweight
- Easy To Use
- Read Only

Define

- Data and Field Names
- Not Methods!

Example - Tuple

Default Tuples

```
(string, int, string) info = ("Fred", 21, "Star Wars);
Console.WriteLine($"{info.Item1} is {info.Item2} and has a favourite movie called {info.Item3}");
```

Named Field Tuples

```
(string Name, int Age, string Movie) info = ("Fred", 21, "Star Wars);
Console.WriteLine($"{info.Name} is {info.Age} and has a favourite movie called {info.Movie}");
```

Enumerated Types

C# Programming Concepts

Classifications via lists

```
/*
    Car Types
    1 : Road Car
    2 : Racing Car
    3 : Rally Car
    4 : F1 Car
*/

int carType = 3;
```

Why might this approach be a problem?

Enumerations

Enumerated Types Define the allowed values

```
enum CarType
{
    RoadCar,
    RacingCar,
    RallyCar,
    F1Car
}
```

Enum defines a new data type

```
CarType carType = CarType.RoadCar;
```

Enums are stored as integers

- Each enum value is implicitly assigned an integer value starting from
- enum values can be explicitly assigned

```
enum CarType {
    RoadCar = 0,
    RacingCar = 1,
    RallyCar = 2,
    F1Car = 3
}
```

Enums can be cast to and from integers

- enum to int

```
> CarType carType = CarType.RoadCar;
> int c = (int) carType;
> c
0
```

- int to enum

```
> int c = (int) 3;
> CarType carType = (CarType) c;
> carType
F1Car
```

Converting enums ToString()

Converting ToString returns the name of enum value

```
> CarType carType = CarType.RoadCar;
> carType.ToString()
RoadCar
```

To return a list of value names or number

```
string[] carTypeNames = Enum.GetNames(typeof(CarType)); // Names
int[] carTypeNames = Enum.GetValues(typeof(CarType)); // Numbers
```

Parsing enums from Strings

An string literal can be parsed into an enum using Enum.Parse

```
string cType = "F1Car";
CarType carType =  (CarType) Enum.Parse(typeof(CarType), cType)
```

- Must cast to enum type - above we cast to (CarType)

Simplifying Enums

Enums can be declared as static

- Allows user to use enum members with prefixing with enum type name

```
using static CarType; // Place with other using statements
...
// CarType carType = CarType.RoadCar;
CarType carType = RoadCar; //  No need to prefix with CarType !
Console.WriteLine(carType.ToString());
RoadCar
```