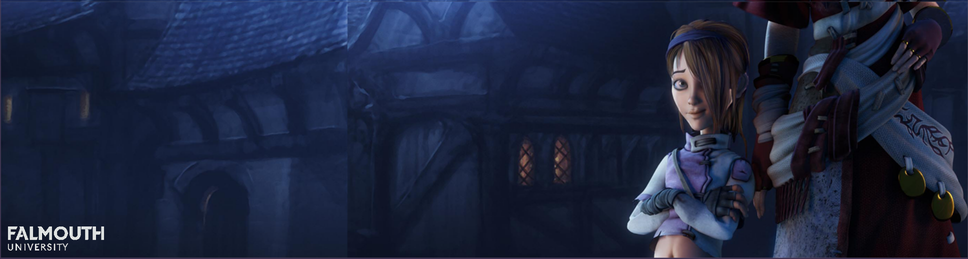

COMP110: Principles of Computing

9: Data Structures II

Worksheet 6

Due **Monday 25th November**

FALMOUTH UNIVERSITY

Generics in C#

The problem

- Suppose we want to define a `Pair` class to store two values
- Something like this...

```
class PairOfInts
{
    public int first;
    public int second;

    public PairOfInts(int f, int s)
    {
        first = f;
        second = s;
    }
}
```

The problem

- This is fine if we just want pairs of `int`s
- To store a pair of `string`s we would need another class:

```
class PairOfStrings
{
    public string first;
    public string second;

    public PairOfStrings(string f, string s)
    {
        first = f;
        second = s;
    }
}
```

The problem

- This quickly gets repetitive!
- We could just store a pair of `object`s — in C# `object` can store values of any type

```
class PairOfObjects
{
    public object first;
    public object second;

    public PairOfObjects(object f, object s)
    {
        first = f;
        second = s;
    }
}
```

- However this doesn't let us impose type safety

The solution

- **Generics** are a feature of C# which let us pass types as "parameters"

```
class Pair<ElementType>
{
    public ElementType first;
    public ElementType second;

    public PairOfObjects(ElementType f, ElementType s)
    {
        first = f;
        second = s;
    }
}
```

- `ElementType` can be any type

The solution

- When we instantiate the generic class, we pass in the type in angle brackets:

```
Pair<int> pairOfInts = new Pair<int>(12, 34);
Pair<string> pairOfStrings = new Pair<string>("hello",
    "world");
```

Multiple parameters

- Generics can take multiple parameters:

```
class Pair<Type1, Type2>
{
    public Type1 first;
    public Type2 second;

    public PairOfObjects(Type1 f, Type2 s)
    {
        first = f;
        second = s;
    }
}
```

```
Pair<int, string> x = new Pair<int, string>(123, "
    hello");
```

Why generics?

- Generics let us write type safe code which can be adapted to data of different types
- Standard libraries in .NET and Unity make use of generics for e.g. container types
- Similar to **templates** in C++

FALMOUTH UNIVERSITY

Basic data structures in C#

Classes and interfaces

- A **class** in C# defines constructors, destructor, methods, properties, fields, ...
- An **interface** defines methods and properties which a class can implement
- An interface is a little like a fully abstract class
- A class in C# can only **inherit** from one **class**, but can **implement** several **interfaces**

IEnumerable

- Most container types in C# implement the `IEnumerable<ElementType>` interface
- Anything implementing `IEnumerable` can be iterated over with a **`foreach`** loop

Arrays

```
int[] myArray = new int[10];
int[] anotherArray = new int[] { 123, 456, 789 };
```

- `int[]` is an array of `int`s
- Size of the array is set on initialisation with `new`
- Array **cannot change size** after initialisation
- Use `myArray[i]` to get/set the `i`th element (starting at 0)
- Use `myArray.Length` to get the number of elements

Multi-dimensional arrays

```
int[,] myGrid = new int[20, 15];
```

- `int[,]` is an 2-dimensional array of `int`s
- Use `myArray[x, y]` to get/set elements
- Use `myArray.GetLength(0)`, `myArray.GetLength(1)` to get the “width” and “height”
- Similarly `int[,,]` is a 3-dimensional array, etc.

Lists

```
using System.Collections.Generic;

List<int> myList = new List<int>();
List<int> anotherList = new List<int> { 1, 2, 3, 4 };
```

- Like a list in Python, but can only store values of the specified type (here `int`)
- Has similar time complexity properties to Python lists
- Append elements with `myList.Add()`
- Get the number of elements with `myList.Count`

Strings

```
string myString = "Hello, world!";
```

- `string` can be thought of as a container
- In particular, it implements `IEnumerable<char>`

Strings are immutable

- Strings are **immutable** in C#
- This is also true in Python, but not in all programming languages
- But wait... we change strings all the time, don't we?

```
string myString = "Hello ";
myString += "world";
```

- This isn't changing the string, it's creating a new one and throwing the old one away!
- Hence building a long string by appending can be slow (appending strings is $O(n)$)
- C# has a **mutable** string type: `StringBuilder`

Dictionaries

- Dictionaries are **associative maps**
- A dictionary maps **keys** to **values**
- Takes two generic parameters: the **key type** and the **value type**
- A dictionary is implemented as a **hash table**

Using dictionaries

```
var age = new Dictionary<string, int> {
    ["Alice"] = 23,
    ["Bob"] = 36,
    ["Charlie"] = 27
};
```

Access values using `[]`:

```
Console.WriteLine(age["Alice"]);  // prints 23
age["Bob"] = 40;        // overwriting an existing item
age["Denise"] = 21;     // adding a new item
age.Add("Emily", 29); // adding a new item -- error if already present
```

Iterating over dictionaries

- `Dictionary<Key, Value>` implements `IEnumerable<KeyValuePair<Key, Value>>`
- `KeyValuePair<Key, Value>` stores `Key` and `Value`

```
foreach (var kv in age)
{
    Console.WriteLine("{0} is {1} years old", kv.Key,
        kv.Value);
}
```

- (Note the `var` keyword — automatically determines the appropriate type to use for a variable)
- Dictionaries are **unordered** — avoid assuming that `foreach` will see the elements in any particular order!

Hash sets

- Sets are **unordered** collections of **unique** elements
  - Sets **cannot** contain **duplicate** elements
  - Attempting to `Add` an element already present in the set does nothing
- `HashSet`s are like `Dictionary`s without the values, just the keys
- Certain operations on sets scale better on average than the equivalent operations on lists:

Operation	List	Hash Set
Add element	Append: $O(1)$ Insert: $O(n)$	$O(1)$
Delete element	$O(n)$	$O(1)$
Contains element?	$O(n)$	$O(1)$

Using sets

```
var numbers = new HashSet<int>{1, 4, 9, 16, 25};
```

Add and remove members with `Add` and `Remove` methods

```
numbers.Add(36);
numbers.Remove(4);
```

Test membership with `Contains`

```
if (numbers.Contains(9))
    Console.WriteLine("Set contains 9");
```

Linked lists

Linked list

- Composed of a number of **nodes**
- Each node contains:
  - An **item** — the actual data to be stored
  - A pointer or reference to the **previous node** in the list (null for the first item)
  - A pointer or reference to the **next node** in the list (null for the last item)

- In C#: `LinkedList<ElementType>`

Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ [1]
Pop	$O(1)$	$O(1)$ [1]
Index lookup	$O(1)$	$O(n)$
Count elements	$O(1)$	$O(n)$
Insert	$O(n)$	$O(1)$ [2]
Delete	$O(n)$	$O(1)$ [2]

[1] If we already have a reference to the last node
[2] If we already have a reference to the relevant node

Workshop

- The **Microsoft .NET documentation** has pages about all the data structures we have seen today (and more)
- **Read** the documentation for these data structures
- **Copy and experiment** with the sample code provided to ensure that you understand these data structures