



FALMOUTH  
UNIVERSITY



COMP110: Principles of Computing

# 9: Data Structures II

# Worksheet 6

Due **Monday 25th November**

# Generics in C#



# The problem

# The problem

- Suppose we want to define a `Pair` class to store two values

# The problem

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

# 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



# The problem

- ▶ This is fine if we just want pairs of `ints`

# The problem

- ▶ This is fine if we just want pairs of `ints`
- ▶ To store a pair of `strings` we would need another class:

# The problem

- ▶ This is fine if we just want pairs of `ints`
- ▶ To store a pair of `strings` 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

# The problem

- ▶ This quickly gets repetitive!

# The problem

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

# The problem

- ▶ This quickly gets repetitive!
- ▶ We could just store a pair of **objects** — 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;
    }
}
```

# The problem

- ▶ This quickly gets repetitive!
- ▶ We could just store a pair of **objects** — 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

# The solution

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

# 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;
    }
}
```

# 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

# The solution

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

# 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



# Multiple parameters

- ▶ Generics can take multiple parameters:

# 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?

# Why generics?

- ▶ Generics let us write type safe code which can be adapted to data of different types

# 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

# 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++

# Basic data structures in C#



# Classes and interfaces



# Classes and interfaces

- ▶ A **class** in C# defines constructors, destructor, methods, properties, fields, ...

# Classes and interfaces

- ▶ A **class** in C# defines constructors, destructor, methods, properties, fields, ...
- ▶ An **interface** defines methods and properties which a class can implement

# 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

# 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

# IEnumerable

- ▶ Most container types in C# implement the `IEnumerable<ElementType>` interface

# 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 };
```



# Arrays

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

- `int[]` is an array of `ints`

# Arrays

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

- ▶ `int[]` is an array of `ints`
- ▶ Size of the array is set on initialisation with `new`

# Arrays

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

- ▶ `int[]` is an array of `ints`
- ▶ Size of the array is set on initialisation with `new`
- ▶ Array **cannot change size** after initialisation

# Arrays

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

- ▶ `int[]` is an array of `ints`
- ▶ 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)

# Arrays

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

- ▶ `int[]` is an array of `ints`
- ▶ 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];
```

# Multi-dimensional arrays

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

- ▶ `int[,]` is an 2-dimensional array of `ints`

# Multi-dimensional arrays

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

- ▶ `int[,]` is an 2-dimensional array of `ints`
- ▶ Use `myArray[x, y]` to get/set elements



# Multi-dimensional arrays

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

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

# Multi-dimensional arrays

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

- ▶ `int[, ]` is an 2-dimensional array of `ints`
- ▶ 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 };
```

# 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`)

# 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

# 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()`

# 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!";
```



# Strings

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

- ▶ `string` can be thought of as a container

# 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

- ▶ Strings are **immutable** in C#

# Strings are immutable

- ▶ Strings are **immutable** in C#
- ▶ This is also true in Python, but not in all programming languages

# 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";
```

# 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!

# 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  $O(n)$ )



# 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

- ▶ Dictionaries are **associative maps**

# Dictionaries

- ▶ Dictionaries are **associative maps**
- ▶ A dictionary maps **keys** to **values**

# Dictionaries

- ▶ Dictionaries are **associative maps**
- ▶ A dictionary maps **keys** to **values**
- ▶ Takes two generic parameters: the **key type** and the **value type**

# 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  
};
```

# 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

# Iterating over dictionaries

- ▶ `Dictionary<Key, Value>` implements  
`IEnumerable<KeyValuePair<Key, Value>>`

# Iterating over dictionaries

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

# 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);
}
```

# 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)

# 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

# Hash sets

- Sets are **unordered** collections of **unique** elements



# Hash sets

- ▶ Sets are **unordered** collections of **unique** elements
  - ▶ Sets **cannot** contain **duplicate** elements

# 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

# 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

# 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:

# 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};
```

# 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);
```

# 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

# Linked list

- ▶ Composed of a number of **nodes**

# Linked list

- ▶ Composed of a number of **nodes**
- ▶ Each node contains:

# Linked list

- ▶ Composed of a number of **nodes**
- ▶ Each node contains:
  - ▶ An **item** — the actual data to be stored

# Linked list

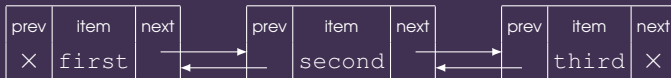
- ▶ Composed of a number of **nodes**
- ▶ Each node contains:
  - ▶ An **item** — the actual data to be stored
  - ▶ A pointer or reference to the **previous node** in the list (null for the first item)

# 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)

# Linked list

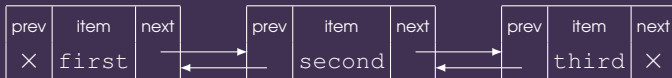
- ▶ Composed of a number of **nodes**
- ▶ Each node contains:
  - ▶ An **item** — the actual data to be stored
  - ▶ A pointer or reference to the **previous node** in the list (null for the first item)
  - ▶ A pointer or reference to the **next node** in the list (null for the last item)





# 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

---

<sup>1</sup>If we already have a reference to the last node

<sup>2</sup>If we already have a reference to the relevant node

# Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ <sup>1</sup>

---

<sup>1</sup>If we already have a reference to the last node

<sup>2</sup>If we already have a reference to the relevant node

# Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ <sup>1</sup>
Pop	$O(1)$	$O(1)$ <sup>1</sup>

---

<sup>1</sup>If we already have a reference to the last node

<sup>2</sup>If we already have a reference to the relevant node

# Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ <sup>1</sup>
Pop	$O(1)$	$O(1)$ <sup>1</sup>
Index lookup	$O(1)$	$O(n)$

---

<sup>1</sup>If we already have a reference to the last node

<sup>2</sup>If we already have a reference to the relevant node

# Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ <sup>1</sup>
Pop	$O(1)$	$O(1)$ <sup>1</sup>
Index lookup	$O(1)$	$O(n)$
Count elements	$O(1)$	$O(n)$

---

<sup>1</sup>If we already have a reference to the last node

<sup>2</sup>If we already have a reference to the relevant node

# Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ <sup>1</sup>
Pop	$O(1)$	$O(1)$ <sup>1</sup>
Index lookup	$O(1)$	$O(n)$
Count elements	$O(1)$	$O(n)$
Insert	$O(n)$	$O(1)$ <sup>2</sup>

<sup>1</sup>If we already have a reference to the last node

<sup>2</sup>If we already have a reference to the relevant node

# Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ <sup>1</sup>
Pop	$O(1)$	$O(1)$ <sup>1</sup>
Index lookup	$O(1)$	$O(n)$
Count elements	$O(1)$	$O(n)$
Insert	$O(n)$	$O(1)$ <sup>2</sup>
Delete	$O(n)$	$O(1)$ <sup>2</sup>

<sup>1</sup>If we already have a reference to the last node

<sup>2</sup>If we already have a reference to the relevant node



# Workshop

For each of the following containers:

`array`, `List`, `LinkedList`,  
`Dictionary`, `HashSet`,  
`SortedDictionary`,  
`SortedList`, `SortedSet`,  
`Stack`, `Queue`

And each of the operations →,  
use the Microsoft .NET  
documentation and  
experimentation to answer the  
following:

- ▶ Does this operation make sense / is it possible?
- ▶ If so, how do you do it?
- ▶ What is the time complexity?
- ▶ Add an element at the (beginning, *i*, end)
- ▶ Remove an element at the (beginning, *i*, end)
- ▶ Get the (first, *i*th, last) element
- ▶ Find if a specific element is present
- ▶ Get the (smallest, largest) element
- ▶ Get the number of elements
- ▶ Copy the container
- ▶ Reverse the container
- ▶ Sort the container
- ▶ Randomly shuffle the container
- ▶ Print a string listing the contents