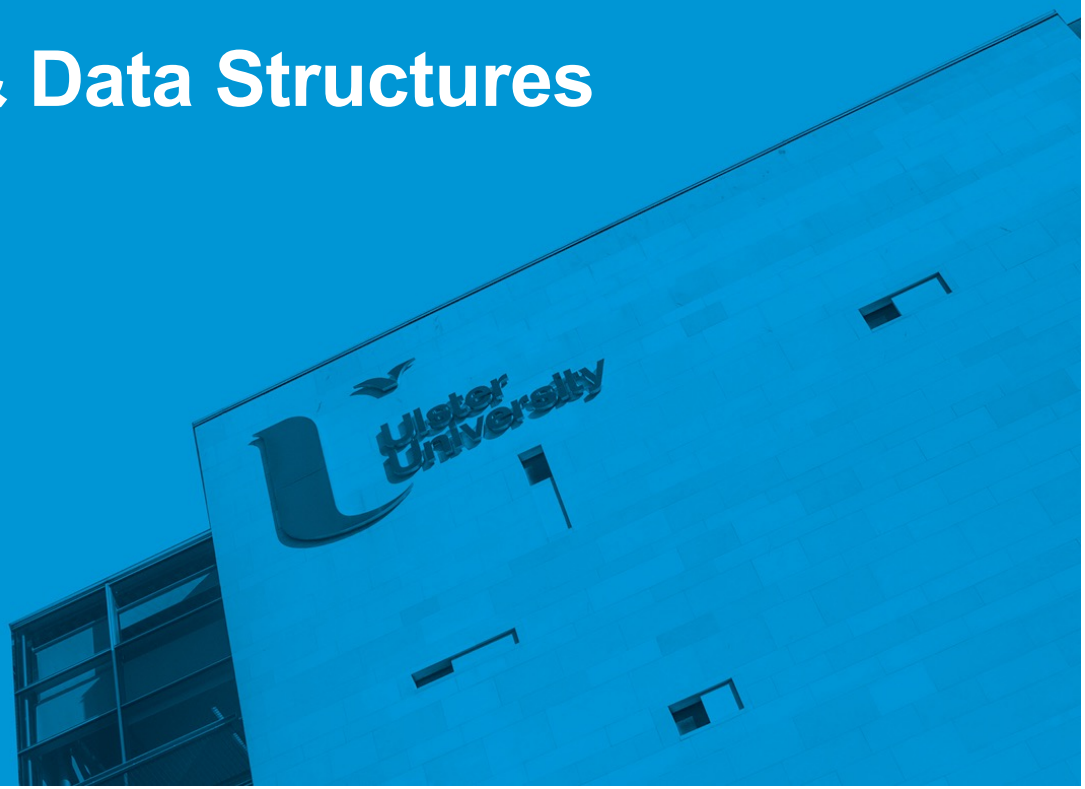
Ulster University

COM498 Algorithms & Data Structures

2.2 The Bag ADT



The ADT Bag

- A collection is an object that groups other objects and provides various services (add, remove, retrieve, query)
- Different collections exist for different purposes whose behaviours are specified abstractly and can differ according to the collection

- Definition of a bag:
  - A finite collection of objects in no particular order
  - Objects have the same or related data types
  - Can contain duplicate items
- Possible behaviors:
  - Get number of items
  - Check for empty
  - Add, remove objects

Bag Behaviours

- Since the bag contains a finite number of objects, one of its behaviours would be:
  - Get the number of items currently in the bag
- A related behavior:
  - See whether the bag is empty
- Adding and removing objects:
  - Add an object to the bag
  - Remove an object from the bag
  - Remove an occurrence of a particular object
  - Remove all objects
- Content-related behaviours:
  - Test whether the bag contains a particular object
  - Count the number of times a certain object occurs
  - List all objects that are in the bag

Specifying a Bag

Ulster University

- Before we can implement a bag in Java we need to describe its data and specify in detail the methods (corresponding to the bag's behaviours)
- Name the methods, choose their parameters, decide on their return types and write comments describing their effects on the bag's data
- First identified behaviour ***"Get the number of items currently in the bag"*** may be described as a method signature as:

```
int getCurrentSize()
```

- Next behaviour ***"See whether the bag is empty"*** may be described as:

```
boolean isEmpty()
```

- Next behaviour ***"Add a given object to the bag"*** *leads to some design decisions . . .*

Specifying a Bag

Ulster University

- We want to add a given object to the bag: we can name the method and give it a parameter to represent the new item:

```
add(newItem)
```

- We might be tempted to make this a *void* method but what do we do if the bag is full and we attempt to add a new entry?
- Some options that we can take if add() cannot complete its task:
  - Do nothing (ignore it and leave the bag unchanged)
  - Leave bag unchanged and signal client (how do we signal the client?)
- Return a boolean to indicate success (where `T` indicates the type of `newItem` - more later):

```
boolean add(T newItem)
```

Specifying a Bag

Ulster University

- Next behaviours are all related to removing entries from a bag:
  "Remove an unspecified object from the bag"
  "Remove an occurrence of a particular object from the bag"
  "Remove all objects from the bag"
- Need to first work out what the return types would be for these operations
- ***"Remove all object from the bag"*** has a simple, well-defined function and does not return any of the bag contents, so can be

```
void clear()
```

- ***"Remove an unspecified object from the bag"*** can return an object or null if the bag is empty, so can be described as

```
T remove()
```

Specifying a Bag

- ***"Remove an occurrence of a particular object from the bag"*** won't be able to remove a particular entry if the bag does not contain that entry!

  1) We could have the method return a boolean value so it can indicate success or not:

     ```
     boolean remove(T anEntry)
     ```

  2) We could have the method return the removed object or null if it can't remove the object:

     ```
     T remove(T anEntry)
     ```

More design decisions . . . which approach should we use?

- Although both are viable choices, **option 1** is preferred in this case. Even though it doesn't return the entry removed, the client already has the entry as the argument to the method. This allows us to be consistent with Java interface Collection)

Specifying a Bag

Ulster University

- The remaining behaviours do not change the contents of the bag:
- ***"Count the number of times a certain object occurs"*** will give the frequency an entry occurs is the bag, hence:

```
int getFrequencyOf(T anEntry)
```

- ***"Test whether the bag contains a particular object"*** will return true or false if an entry occurs is the bag, hence:

```
boolean contains(T anEntry)
```

- ***"Look at all objects that are in the bag"*** will look at the contents of the bag. Rather than displaying all the entries in the bag, it is better to return an array of the entries and let the client display them (in any way they want): :

```
T[] toArray()
```

Specifying a Bag (summary)

- A complete list of our method signatures:

```
• int getCurrentSize()
• boolean isEmpty
• boolean addNewEntry(T newEntry)
• T remove()
• boolean remove(T anEntry)
• void clear()
• int getFrequencyOf(T anEntry)
• boolean contains(T anEntry)
• T[] toArray
```

- Note that we have made no decisions about how the data will be stored – but this is an implementation issue (rather than one of design)

Scenario

- Implement the specification of the Bag abstract data type by creating a Java Interface Class
  - Create a new Java project called Bag and provide the interface in a file called BagInterface.java
  - The Interface Class specifies the front-end of our ADT – specifying the methods that will be available for external entities to interact with instances of our Bag class.
  - We will provide an implementation for the class, describing how these activities are to take place, soon…

Java Sidenote - Generics

- **Generic methods** allow us to specify operations that can take place on objects of different data types
    - For example, a method that operates on an array might be appropriate on an array of either integers, floating point values or strings – how do we define the parameter?

```
void doSomething (int[] anArray) { … }
void doSomething (float[] anArray) { … }
void doSomething (char[] anArray) { … }
```

Java Sidenote - Generics

Ulster University

- The answer is to use the Generic Type `T`, which is introduced in the method signature (before the return type) and then used where required, so…

```
<T> void doSomething (T[] anArray) {
    for (T element : aArray) {
        System.out.println(element);
    }
}
```

- The method can then be called with parameters of any appropriate type, such as…

```
Integer[] intArray = { 1, 2, 3, 4 };
String[] stringArray = { "one", "two", "three", "four" };
doSomething(intArray);
doSomething(stringArray);
```

Java Sidenote - Generics

Ulster University

- A **Generic Class** is specified by providing the generic type after the class name and allows us to define ADTs that can be applied to elements of any type.

```
public class Thing<T> {
   T aThing;

   public T get() {
      return aThing
   }

   public void set(T something) {
      aThing = something
   }
}
```

Java Sidenote - Generics

- We provide the type when an instance of the generic class is created

```
public class Thing<T> {
    …

    public static void main(String[] args) {

        Thing<String> myThing = new Thing<String>();
        Thing<Integer> yourThing = new Thing<Integer>()

        myThing.set("Adrian");
        yourThing.set(10);
    }
}
```

Scenario

- Use generics to provide a single method that will print the contents of either an array of integers or an array of strings

```
public class Printer() {

  public static void main(String[] args) {
     Integer[] intArray = { 1, 2, 3 };
     String[] stringArray = { “one”, “two, “three” };
     printArray(intArray);
     printArray(stringArray)
  }

  public static void printArray() { }
}
```

Can you provide this method?