Cosc 241
Programming and Problem Solving
Lecture 13 (8/4/19)
Collections and ADTs

Michael Albert
michael.albert@cs.otago.ac.nz

Keywords: abstract data type, collection, generic class type, stack

Collections

- Most computing involves processing collections of data – so the ways in which those collections are stored are among the most important data types.
- A *collection* in the object oriented context is an object that gathers and organises other objects.
- It defines the way in which those *elements* can be accessed and managed.
- It specifies (and limits) the ways in which the user may interact with the collection.

Types of collections

- There are two broad types of collections *linear* and *non-linear*.
- In a linear collection, the organisation is “in a straight line”, i.e., we have natural notions of precedence and indexing.
- In a non-linear collection the organisation may be more complex (e.g., based on a tree, or network), or not present.
- The way in which elements are organised is usually determined by the sequence in which they are added to the collection as well as some inherent relationships, e.g., ordering, among the elements themselves.

Abstract data types

- An *abstraction* is a method of hiding certain details at certain times.
- As a result, the user can focus on the important issues, and not be concerned with the messy details.
- Often, abstractions focus on the *interface* to a process or structure, rather than on the process or structure itself.
- In software engineering, abstractions are often used when the detailed implementation of certain operations *should not* affect the manner in which they are used.
- For collections in particular, abstractions provide a powerful method of ensuring consistent and clean access and manipulation without having to worry about (usually) irrelevant details.

ADT versus “data structure”

- An *abstract data type* (ADT) is a data type whose intended use is specified by its interface.
- A *data structure* is the collection of programming structures (e.g., methods, data fields) used to implement an ADT (or collection).
- The data structure that implements an ADT can be changed without changing the interface, and therefore in a way that does not affect any client programs.
- Concerns of efficiency come about in evaluating which data structures to use when implementing an ADT.
- These will not generally be one-sided, i.e., there will typically be both costs and benefits associated with a particular choice of implementation.

Java collections

- Much of the Java library is organised into application programming interfaces (APIs).
- An API is (typically) a collection of ADTs (i.e., interfaces) together with some specific data structures that implement them.
- One such is the Java Collections API which represents specific types of collections.
- So, why should we bother with anything beyond learning that API?
- The collection type we want might not be there (or not exactly).
- We may have a specific implementation in mind (because of special conditions).
- We need to *understand* the issues involved.

Generics

- In a language like Java where type checking occurs at compile time, a key question about collections is: collections of *what*?
- In early versions of Java any sort of collection was fine, provided it was `Object`.
- This caused much casting, or silly wrapper classes.
- Since Java 5.0 *generic types* are specified with collections.
- From our standpoint that means we can specify the type of object that a collection will hold – access to the collection will return objects of that type, and only objects of that type can be added to the collection.

Generic syntax

- When a class contains a generic type parameter it is represented inside "pointy brackets" e.g., `<T>` or `<String>`.
- The first sort of use is when we're specifying a type variable that is, indicating in a class that "you could have objects of any type here" but where that type might be used in e.g., method parameters or returns.
- The second sort of use is when we're actually creating an instance of the class and we want to specify "what `T` is" in this specific instance, e.g., "everywhere you see `T` for this instance it means `String`".

Examples

The source code for `ArrayList` starts:

```
public class ArrayList<E> extends AbstractList<E>
```

(the Java developers like `E` for their generic type, I prefer `T`)

If we want to make an `ArrayList` to contain `String` objects we write:

```
ArrayList<String> s = new ArrayList<String>();
```

The following also works (the compiler can infer that the diamond on the right should be filled with `String`:

```
ArrayList<String> s = new ArrayList<>();
```

Engineering a Stack ADT

A *stack* is a last in, first out collection. Think of a stack of papers to be processed, or a stack of dishes to be washed. The basic stack operations are:

- *Push*: which adds an item to the stack,
- *Pop*: which removes (and returns) an item from the stack.

In practice these are usually extended by allowing *Peek*, which examines the top item of the stack, as well as convenience methods that return the size of the stack, and an emptiness test.

Stack ADT

```
public interface Stack<T> {

  public void push (T element);
  public T pop();
  public T peek();
  public boolean isEmpty();
  public int size();

}
```

Using a stack to sort

A stack can be used to (partially) sort an incoming stream of objects. The algorithm is as follows:

- Compare the next incoming item with the top of the stack (if any).
- If it is smaller, push it onto the stack.
- Otherwise, pop the stack until the top is larger than the input (or it is empty), then push it onto the stack.
- When all the input has been processed, pop each remaining item off the stack.

Stack sorting

```
public static Stack<String> stack = new Stack<String>();

public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    while (input.hasNext())
        process(input.next());
    while (!stack.isEmpty())
        System.out.println("---> " + stack.pop());
}

public static void process(String s) {
    if (stack.isEmpty() || s.compareTo(stack.peek()) < 0) {
        stack.push(s);   return;
    }
    System.out.println("---> " + stack.pop());
    process(s);
}
```

Questions

Not relevant to COSC 241:

- What characterises the ordering of input sequences that are not completely sorted by the stack sorting algorithm?
- If we consider all possible input orderings of n distinct items, how many of them are completely sorted by the stack sorting algorithm?