

CITS2200
Data Structures and Algorithms

Cara MacNish

School of Computer Science & Software Engineering
University of Western Australia

Topic 4

Data Abstraction and Specification of ADTs

- Example — The “Reversal Problem” and a non-ADT solution
- Data abstraction
- Specifying ADTs
- Interfaces
- javadoc documentation
- An ADT solution to the Reversal Problem

4.1 Aims


The aims of this topic are to:

1. provide a more detailed example of data type abstraction
2. introduce two example data types: the Queue and Stack
3. show how data types will be specified in this unit

4.2 The Reversal Problem and a non-ADT solution

As a more detailed example of ADTs we consider the reversal problem:

Given two character sequences A and B , is A the reverse of B ?

One solution: store in arrays, scan and compare from either end ... 

```
import java.io.*;

/*
 * Reversal program (not using ADTs).
 * Accepts two character strings from the terminal, separated by
 * whitespace, and determines whether one is the reverse of the
 * other.
 */
public class Reversal {

    // constant for maximum length of the input sequences
    public final static int MAX_SEQUENCE = 100;

    // main program
    public static void main(String[] args) throws IOException {
```

```
// arrays for storing input sequences
char[] sequence1 = new char[MAX_SEQUENCE];
char[] sequence2 = new char[MAX_SEQUENCE];

// indices for first and second sequences
int index1 = 0;
int index2 = 0;

// other local variables
boolean isReverse = true;
char c;
```

```
// Read in the first sequence and store
c = (char) System.in.read();
while (c != ' ') {
    sequence1[index1] = c;
    index1++;
    c = (char) System.in.read();
}

// Clear white space.
while (c == ' ') c = (char) System.in.read();

// Read in the second sequence and store
while (c != ' ' && c != '\n' && c != '\r') {
    sequence2[index2] = c;
    index2++;
    c = (char) System.in.read();
}
```

```
// Compare the two sequences.
isReverse = index1 == index2;
index1 = 0;
index2--;

while (isReverse && index1 <= index2) {
    isReverse = isReverse &&
        sequence1[index1] == sequence2[index2-index1];
    index1++;
}

if (isReverse) System.out.println("Yes that is the reverse.");
else System.out.println("No thats not the reverse.");
}
}
```

Notice that this program mixes

- “low-level” details of data storage (in arrays) and manipulation (using indices), with
- the “high-level” goals of inputting and comparing sequences.

⇒ difficult to modify, maintain, reuse, etc

Better solution — use ADTs!

4.3 Data abstraction

The above program integrates:

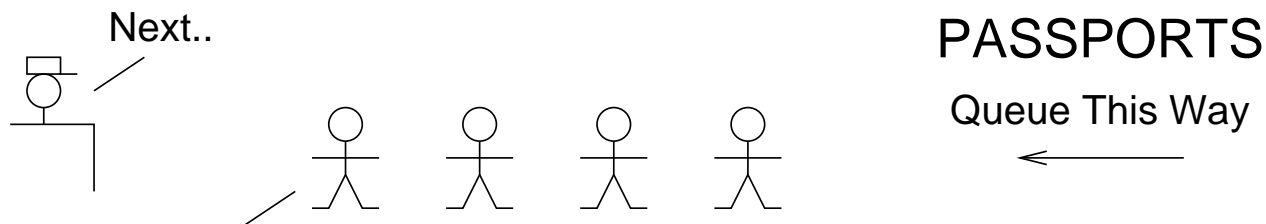
- data, and instructions to access it
- “higher-level” role of the program

We wish to take a more abstract view... can we use generic, reusable data structures?

When dealing with the first sequence we...

- “Create” an empty sequence
- Append characters to the end
- Scan from beginning to end
- Don't reuse scanned characters

But this is just what a **queue**, or **FIFO** (first-in, first-out buffer), does!



In general the operations on a queue include:

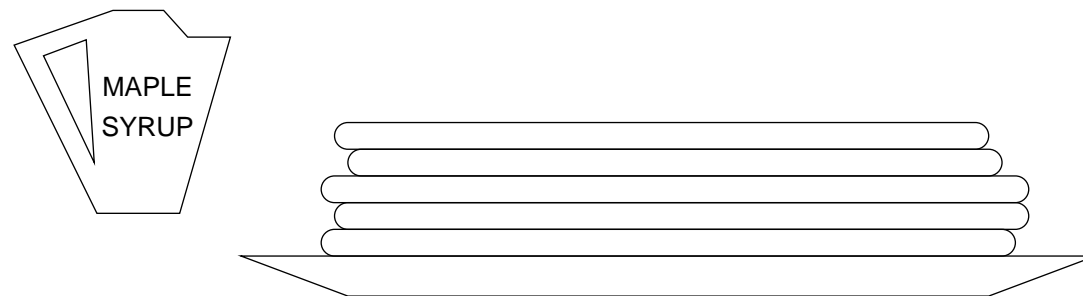
1. Create an empty queue
2. Test whether the queue is empty
3. Add a new latest element
4. Examine the earliest element
5. Delete the earliest element

From a user point-of-view, we **don't care how its implemented** — all we need in order to write our reversal program is what operations are available to us.

(Implementations will be considered later.)

Operations needed for the second sequence are the same as the first, except the elements added **last** are taken off first.

This is the operation of a **stack**, or **LIFO** (last-in first-out buffer).



Operations on a stack:

1. Create an empty stack
2. Test whether the stack is empty
3. Add (**push**) a new element on the top
4. Examine (**peek** at) the top element
5. Delete (**pop**) the top element

Implementation of a stack — see Lab Exercises!

4.4 Specifying ADTs

We saw in Topic 1 that ADTs consist of a set of operations on a set of data values. We can **specify** ADTs by listing the operations (or **methods**).

The lists of operations on the previous pages are very informal and not sufficient for writing code. For example

2. Test whether the queue is empty

doesn't tell us the name of the method, what arguments it is called with, what is returned, and whether it can throw an exception.

In these notes we will specify ADTs by providing at least:

- the **name** of each operation
- example **parameters** (the implementation may use different parameter names, but will have the same number, type and order)
- an explanation of **what the operation does** — in particular, any constraints on, or changes to, the parameters, changes to the ADT instance on which the method operates, what is returned and any exceptions thrown

Thus a Queue ADT might be specified by the following operations:

1. **Queue()**: create an empty queue
2. **isEmpty()**: return `true` if the queue is empty, `false` otherwise
3. **enqueue(e)**: e is added as the last item in the queue
4. **examine()**: return the first item in the queue, or throw an exception if the queue is empty
5. **dequeue()**: remove and return the first item in the queue, or throw an exception if the queue is empty

Note: No variable in the argument list corresponds to the object itself (the queue). This is because the methods are **instance methods** — whenever they are called they will “belong” to a particular object.

eg.

```
Queue q = new Queue();  
System.out.println(q.isEmpty());
```

In data structure texts for non-object-oriented languages such as Pascal, you will find an extra argument in the specification of operations.

Similarly, the specification of a Stack ADT:

1. **Stack()**: create an empty stack
2. **isEmpty()**: return `true` if the stack is empty, `false` otherwise
3. **push(e)**: item `e` is pushed onto the top of the stack
4. **peek()**: return the item on the top of the stack, or throw an exception if the stack is empty
5. **pop()**: remove and return the item on the top of the stack, or throw an exception if the stack is empty

Note: The use of upper and lowercase in method names should follow the rules described in the document **Java Programming Conventions**.

4.5 Interfaces

As we have seen, Java itself provides a rigorous way of specifying the methods in classes: **interfaces**.

Interfaces provide a natural way of specifying ADTs in programs and enforcing those specifications.

Example . . . 

```
// Interface for a Queue of characters.
public interface QueueChar {

    /*
     * test whether the queue is empty
     * return true if the queue is empty, false otherwise
     */
    public boolean isEmpty ();

    /*
     * insert an item at the back of the queue
     */
    public void enqueue (char a);
}
```

```
/*
 * examine and return the item at the front of the queue
 * throw an Underflow exception if the queue is empty
 */
public char examine () throws Underflow;

/*
 * remove the item at the front of the queue
 * return the removed item
 * throw an Underflow if the queue is empty
 */
public char dequeue () throws Underflow;
}
```

Note: This interface specifies a queue of characters (chars). This can be seen in the argument to enqueue and the return types of examine and dequeue.

In this course (particularly in the Labs) we will specify data structures using interfaces, and in most cases consider a number of alternative implementations.

(We'll look at different implementations of the QueueChar interface later.)

4.6 javadoc **Documentation**

Many texts will describe ADT operations in terms of **preconditions** and **postconditions**.

preconditions — constraints on variable values for the operations to work correctly

post-conditions — what the operation does, in particular changes to the input variables

In this course we will replace these, as far as possible, with the facilities provided by the documentation program javadoc.

The documentation for each method should include:

- a short general description of the method
- a `@param` statement describing each parameter
- a `@return` statement describing the value/object returned (except where the return type is void)
- an `@exception` statement describing each exception thrown

The `javadoc` program automatically generates HTML on-line documentation from these comments.

Example

```
/**
 * remove the item at the front of the queue
 * @return the removed item
 * @exception Underflow if the queue is empty
 */
public char dequeue () throws Underflow;
```

Here the “precondition” is that the queue must be non-empty, the “postcondition” is that the front element is deleted.

The final QueueChar interface ... 

```
package DAT;           // make this interface part of a package
                       // (or library) called DAT

import Exceptions.*;  // use a package of exceptions called
                       // Exceptions (contains Underflow)

/**
 * Interface for Queue of characters.
 * @author Cara MacNish           // some other javadoc fields
 */
public interface QueueChar {

    /**
     * test whether the queue is empty
     * @return true if the queue is empty, false otherwise
     */
    public boolean isEmpty ();
```

```
/**
 * insert an item at the back of the queue
 * @param a the item to insert
 */
public void enqueue (char a);

/**
 * examine the item at the front of the queue
 * @return the first item
 * @exception Underflow if the queue is empty
 */
public char examine () throws Underflow;

/**
 * remove the item at the front of the queue
 * @return the removed item
 * @exception Underflow if the queue is empty
 */
public char dequeue () throws Underflow;
}
```

Notes:

- Full javadoc documentation must be included with code that you submit on this course.
- We will sometimes omit documentation (or break formatting rules) in lectures to fit programs on slides.

4.7 An ADT solution to the reversal problem

Given specifications for Queue and Stack ADTs, which we assume for the moment are implementations of interfaces `QueueChar` and `StackChar` called `QueueCharImplementation` and `StackCharImplementation` respectively, the Reversal program can be rewritten at a more abstract level.

Program ... ▷

```
package DAT;
import java.io.*;
import Exceptions.*;

/**
 * Reversal program using ADTs.
 * Accepts two character strings from the terminal, separated by
 * whitespace and determines whether one is the reverse of the other.
 * @author Cara MacNish
 */
public class ReversalADT {

    /**
     * main program
     * @param args command line arguments
     * @exception Exception passed to interpreter
     */
    public static void main(String[] args) throws Exception {
```

```
// queue for storing first input sequence
QueueChar q = new QueueCharImplementation();

// stack for storing second input sequence
StackChar s = new StackCharImplementation();

// other local variables
boolean isReverse = true;
char c;

// Read in the first sequence and store characters in a queue.
c = (char) System.in.read();
while (c != ' ' && c != '\n' && c != '\r') {
    q.enqueue(c);
    c = (char) System.in.read();
}

// Clear white space.
while (c == ' ')    c = (char) System.in.read();
```

```
// Read in the second sequence and store characters in a stack.
while (c != ' ' && c != '\n' && c != '\r') {
    s.push(c);
    c = (char) System.in.read();
}

// Compare the two sequences.
while (isReverse && !q.isEmpty() && !s.isEmpty())
    isReverse = isReverse && q.dequeue() == s.pop();

if (isReverse && q.isEmpty() && s.isEmpty())
    System.out.println("Yes that is the reverse.");
else System.out.println("No thats not the reverse.");
}
}
```

Advantages over previous version

- Program 'reads' better
 - more 'declarative'
 - easier to follow and debug
- Modular
 - Implementation independent — easier to change/upgrade
 - Division of work-load

4.8 Summary

- When programming we should look for **abstractions** of the data — could we use a generic data structure (ADT) rather than “reimplement the wheel”?
- ADTs can be specified by listing operations and explaining how the object and arguments are affected
- More rigorous specifications can be enforced in Java using interfaces
- ADT operations (methods) should be described within the implementation using `javadoc` comments

Next we will look at implementations for the Queue...