Collections and the Stack ADT
Objectives

• Define the notion of collection
• Define the notion of stack
• Study an array implementation of stacks
• Uses and importance of stacks
Collections

*Collection*: a group of items that we wish to treat as a conceptual unit

- The proper choice of a collection for a given problem can improve the efficiency and simplicity of a solution.
Conceptual View of a Stack

top of stack

bottom of stack
Conceptual View of a Stack

Adding an element (Push)

new top of stack

bottom of stack
Conceptual View of a Stack

Removing an element (Pop)

[Diagram showing a stack with labeled top and bottom]
Stacks

- **Stack**: a collection whose elements are added and removed from one end, called the *top* of the stack
- Stack is a *LIFO* (Last In, First Out) data structure
A stack is a LIFO structure

Order in which items were added
Stack Operations

- **push**: add an element at the top of the stack

```
data item 3
data item 2
data item 1
```

Push (data item 4)

```
data item 4
data item 3
data item 2
data item 1
```
Stack Operations

- **pop**: remove the element at the top of the stack

Diagram:

```
  data item 4
  data item 3
  data item 2
  data item 1

  pop ()

  data item 3
  data item 2
  data item 1
```
Stack Operations

- **peek**: examine the element at the top of the stack without removing it.

```
stack
| data item 4 |
| data item 3 |
| data item 2 |
| data item 1 |
```

peek → data item 4
Stack Operations

- **size**: number of elements in the stack
- **isEmpty**: true if the stack is empty
- **toString**: string representation of stack

```
data item 4
data item 3
data item 2
data item 1
```

- size: 4
- isEmpty: false
- toString: “Stack: data item 4 data item 3 data item2 data item 1”
Abstract Data Type (ADT)

It is a *collection* of data together with the *operations* on that data.

The ADT specifies *WHAT* the operations do, not *HOW* they do it.
Stack ADT

• **Stack Abstract Data Type (Stack ADT)**

  It is a *collection* of data together with the *operations* on that data:

  • push
  • pop
  • peek
  • Size
  • isEmpty
  • toString
Abstraction

• Abstraction separates the purpose of an entity from its implementation or how it works
  • *Example in real life*: a car (we do not have to know how an engine works in order to drive a car)
  • *Example in computer systems*: a computer (we do not need to know how information is stored and manipulated by the CPU to be able to execute programs)
Abstraction in Programming

**Data type**: a set of values and the operations defined on those values

Ex. integer data type (**int**):
- Values: … -2, -1, 0, 1, 2, …
- Operations: +, -, x, /, …

A data type is defined by a programming language.
Stack ADT

It is a *collection* of data together with the *operations* on that data:

- push
- pop
- peek
- Size
- isEmpty
- toString

An ADT is defined by the programmer.
Java Interfaces

Java has a *programming construct* called an *interface* that we can use to define what the operations on an ADT are.
public interface StackADT<T> {
    // Adds one element to the top of this stack
    public void push (T dataItem);
    // Removes and returns the top element of this stack
    public T pop( );
    // Returns the top element of this stack
    public T peek( );
    // Returns true if this stack is empty
    public boolean isEmpty( );
    // Returns the number of elements in this stack
    public int size( );
    // Returns a string representation of this stack
    public String toString( );
}
Java Interfaces

• A Java interface is a list of abstract methods (the signatures of the methods) and constants
  • Must be public
  • Constants must be declared as static final
public interface StackADT<T> {
    // Adds one element to the top of this stack
    public void push (T dataItem);
    // Removes and returns the top element of this stack
    public T pop( );
    // Returns the top element of this stack
    public T peek( );
    // Returns true if this stack is empty
    public boolean isEmpty( );
    // Returns the number of elements in this stack
    public int size( );
    // Returns a string representation of this stack
    public String toString( );
}
Generic Types

What is this \(<T>\) in the interface definition?

- It is called a **generic type**
  - The above interface defines a Stack for objects of type \(T\)
- The **actual type** is known only when an application program creates an object of that class
  - Example:
    - StackADT<String> s = new …
    - StackADT<Person> p = new …
    - StackADT<Rectangle> r = new …

  …
Generic Types

• Note: it is merely a convention to use the letter \( T \) to represent the generic type; any other letter or word can be used to represent the generic type.

• In a class definition, we enclose the generic type in angle brackets: \(< T >\)
public interface StackADT<GenericType> {
    // Adds one element to the top of this stack
    public void push (GenericType dataItem);
    // Removes and returns the top element of this stack
    public GenericType pop( );
    // Returns the top element of this stack
    public GenericType peek( );
    // Returns true if this stack is empty
    public boolean isEmpty( );
    // Returns the number of elements in this stack
    public int size( );
    // Returns a string representation of this stack
    public String toString( );
}
Modular Design

Interface

Module

Implementation

Module

Module
Implementing an Interface

• We cannot create an object of the class StackADT.

• To be able to create Stack objects, we first need to create a class that implements the interface by providing the implementations (code) for each of the abstract methods
Stack Implementation Issues

• What do we need to implement a stack?
  • A data structure (container) to hold the data elements
  • Something to indicate the top and bottom of the stack
Array Implementation of a Stack

bottom

0 1 2 3 4 5 6

Data Item 1
Data Item 2
Data Item 3
Data Item 4
Data Item 5

Top = 5
Array Implementation of a Stack

- Data Item 1
- Data Item 2
- Data Item 3

Push

Bottom

Top = 5
Array Implementation of a Stack

- **bottom**
- **Push ( )**

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Item 1</td>
<td>Data Item 2</td>
<td>Data Item 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Top = 5
Array Implementation of a Stack

Data Item 1
Data Item 2
Data Item 3

Push

Bottom
Top = 6
Array Implementation of a Stack

Pop ()

Top = 6

Data Item 1  Data Item 2  Data Item 3
Array Implementation of a Stack

Pop ()

Data Item 1
Data Item 2
Data Item 3

Top = 5
Array Implementation of a Stack

Pop ()

Top = 5
Array Implementation of a Stack

0 1 2 3 4 5 6
Data Item 1 Data Item 2 Data Item 3

Pop ()

Top = 5
Array Implementation of a Stack

Top = 5

Pop ()
return

Data Item 1
Data Item 2
Data Item 3

0 1 2 3 4 5 6
public interface StackADT<T> {
    // Adds one element to the top of this stack
    public void push (T dataItem);
    // Removes and returns the top element of this stack
    public T pop( );
    // Returns the top element of this stack
    public T peek( );
    // Returns true if this stack is empty
    public boolean isEmpty( );
    // Returns the number of elements in this stack
    public int size( );
    // Returns a string representation of this stack
    public String toString( );
}
public class ArrayStack<T> implements StackADT<T> {
    private T[] stack; // Array for the data
    private int top;   // Top of stack
    private final int DEFAULT_CAPACITY = 100;

    public ArrayStack() {
        top = 0;
        stack = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    public ArrayStack(int initialCapacity) {
        top = 0;
        stack = (T[]) (new Object[initialCapacity]);
    }
}
Example of using Constructor to create a Stack of Numbers

ArrayStack<String> s =
new ArrayStack<String>(5);

Code references
Example: the same **ArrayStack** object after four items have been pushed on

```
ArrayStack<String> s =
new ArrayStack<String>(5);
```

Code references

```java
ArrayStack<String> s =
new ArrayStack<String>(5);
```
public class ArrayStack<T> implements StackADT<T> {

    private T[] stack;    // Array for the data
    private int top;       // Top of stack
    private final int DEFAULT_CAPACITY = 100;

    public ArrayStack() {
        top = 0;
        stack = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    public ArrayStack(int initialCapacity) {
        top = 0;
        stack = (T[]) (new Object[initialCapacity]);
    }

    Why such complex declaration?
public class ArrayStack<T> implements StackADT<T> {

    private T[] stack; // Array for the data
    private int top; // Top of stack
    private final int DEFAULT_CAPACITY = 100;

    public ArrayStack() {
        top = 0;
        stack = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    public ArrayStack(int initialCapacity) {
        top = 0;
        stack = new T[initialCapacity];
    }
}

Why is this wrong?
public class ArrayStack<T> implements StackADT<T> {

    private T[] stack;    // Array for the data
    private int top;      // Top of stack
    private final int DEFAULT_CAPACITY=100;

    public ArrayStack( ) {
        top = 0;
        stack = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    public ArrayStack (int initialCapacity) {
        top = 0;
        stack = new Object[initialCapacity];
    }

    Why is this wrong?
public class ArrayStack<T> implements StackADT<T> {

    private T[] stack; // Array for the data
    private int top; // Top of stack
    private final int DEFAULT_CAPACITY=100;

    public ArrayStack( ) {
        top = 0;
        stack = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    public ArrayStack (int initialCapacity) {
        top = 0;
        stack = (T[]) (new Object[initialCapacity]);
    }
}
public interface StackADT<T> {
  // Adds one element to the top of this stack
  public void push (T dataItem);
  // Removes and returns the top element of this stack
  public T pop( );
  // Returns the top element of this stack
  public T peek( );
  // Returns true if this stack is empty
  public boolean isEmpty( );
  // Returns the number of elements in this stack
  public int size( );
  // Returns a string representation of this stack
  public String toString( );
}
Array Implementation of a Stack

(bottom)

Data Item 1  Data Item 2  Data Item 3

Push

Top = 5
public void push (T dataItem) {
    if (top == stack.length)
        expandCapacity( );

    stack[top] = dataItem;
    top++;
}
// Helper method to create a new array to store the
// contents of the stack, with twice the capacity
private void expandCapacity() {
    T[] larger = (T[]) (new Object[stack.length*2]);

    for (int index=0; index < stack.length; index++)
        larger[index] = stack[index];

    stack = larger;
}
Array Implementation of a Stack

Pop ()

Top = 6
// Removes the element at the top of the stack and returns a reference to it. Throws an EmptyCollectionException if the stack is empty.

public T pop() throws EmptyCollectionException {
    if (top == 0)
        throw new EmptyCollectionException("Empty stack");
    top--;
    T topItem = stack[top];
    stack[top] = null;
    return topItem;
}
Array Implementation of a Stack

Data Item 1
Data Item 2
Data Item 3

Pop ()

Top = 6
// Returns the element at the top of the stack. Throws an
// EmptyCollectionException if the stack is empty.
public T peek( ) throws EmptyCollectionException {
    if (top == 0)
        throw new EmptyCollectionException("Empty stack");
    return stack[top-1];
}
// Returns the number of elements in the stack
public int size() {
    return top;
}

// Returns true if the stack is empty and false otherwise
public boolean isEmpty() {
    return (top == 0);
}
public String toString() {
    String result = "Stack:\n";

    for (int index=0; index < top; index++)
        result = result + stack[index].toString() + "\n";

    return result;
}
}
Uses of Stacks in Computing

Stacks are fundamental structures in Computer Science

- **Execution stack (runtime or call stack)**
  - Used by runtime system when methods are invoked
  - Holds “activation records” (or “frames” or “call frames”) containing local variables, parameters, return address, etc.
public static void main (String[] args) {
    method1();
}

private void method1() {
    method2(x);
}

private void method2(int x) {
}

Execution stack
Uses of Stacks in Computing

Useful for any kind of problem involving **LIFO** data

- **Backtracking**: in solving a maze or finding a path in a map
Uses of Stacks in Computing

- **Word processors or editors**

  - To check expressions or strings of text for matching parentheses / brackets
  
  e.g. if (a == b) {

  
  \[
  c = ((d + e) - f) * (d + e);
  \]

  }


Uses of Stacks in Computing

• *Word processors or editors*

  To implement *undo* operations

  • Keeps track of the most recent operations

  if (a == b)) c =
Using a Stack: Postfix Expressions

• Normally, we write expressions using **infix notation**:
  • Operators are between operands: $3 + 4 \times 2$
  • Parentheses force precedence: $(3 + 4) \times 2$
• In a **postfix expression**, the operator comes **after** its two operands
  • Examples above would be written as:
    
    \[
    3 \quad 4 \quad 2 \quad \times \quad + \\
    3 \quad 4 \quad + \quad 2 \quad \times 
    \]

Evaluating Postfix Expressions

- **Algorithm to evaluate a postfix expression:**
  - Scan from left to right, determining if the next token is an operator or operand
  - If it is an operand, push it on the stack
  - If it is an operator, pop the stack twice to get the two operands, perform the operation, and push the result back onto the stack
- Try the algorithm on our examples …
- At the end, there will be one value in the stack – what is it?
Using a Stack to Evaluate a Postfix Expression

Evaluation of

\[ 7 \ 4 \ -3 \ * \ 1 \ 5 \ + \ / \ * \]

At end of evaluation, the result is the only item on the stack