Data Structures:
Ways of managing Collections of objects

I had code for managing a collection of CDs, and I needed code for managing a collection of Shapes…
I know CDs are not Shapes, but they are both *collections*!

How can I *reuse* the code?
Are there better ways to *organize* my collections?
Is there a better way than using an *array*?
And what is “*better*”
What are Collections?

- A **collection** is an object that gathers and organizes other objects ("elements").

- A collection provides operations to **add**, **remove**, and **manage** its elements.

- Collections can be separated into two categories:
  - **linear**: elements are organized in a straight line, e.g.: ____________
  - **nonlinear**: elements are organized in something other than a straight line, e.g., ________________

- There are many ways ("data structures") to implement collections. Some are better than others in terms of **efficiency** of their operations and **ease of use** to solve for a particular problem.
Two Famous Collections

- **A stack**
  - Last-in, first-out (**LIFO**) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria

- **vs: A queue**
  - First in, first out (**FIFO**) property
    - The first item added is the first item to be removed
  - Analogy
    - A queue of train commuters
Let’s think...

Why is a queue of books a better choice than a stack for organizing your required reading?

**Answer:** With a stack, you would always read the latest required reading, and you might never get to the oldest readings.
Stack Animation

Adding an element

Removing an element

Top of stack
“Contract” and **interface** for the Stack Collection

**Stack operations**
- **Create**: Add a new item to the stack (aka constructor)
- **push**: Add a new item to the stack
- **pop**: Remove from the stack the item that was added most recently
- **peek**: Retrieve (but not remove) from the stack the item that was added most recently
- **isEmpty**: Determine whether a stack is empty or not
- **size**: Find out how many items are in the stack

```java
/** Adds an item to the top of a stack. */
public void push(T newItem);
/** Removes the top of a stack. */
public T pop();
/** Retrieves the top of a stack. */
public T peek();
/** Determines whether stack is empty. */
public boolean isEmpty();
/** Returns the number of items on the stack. */
public int size();
```
Using a Stack – push and pop

```java
include java.util.Stack; //uses Java’s implementation
public class StackTest {
    public static void main (String[] args) {
        Stack<String> stk = new Stack<String>();
        stk.push("one");
        stk.push("two");
        stk.pop();
        stk.push("three");
        System.out.println("Contents of Stack: "+ stk);
    }
} //What does stk contain now?
```

Q: How can we print all the elements of a stack without destroying it?
The need for Generic Types \(<T>\)

- Assume we have defined a \(\textbf{Group}\) class that stores and manages a group of objects (e.g., \textit{Stack, Queue, Bag, ...})

- \(\textbf{Group}\) stores \textit{Objects} (which can be of any type – polymorphism!)
  - But then the compiler cannot tell what methods are available for the objects in my group...
  - Wouldn’t it be better to say I want a \(\textbf{Group}<\text{Shape}>\)

- I can! creating a \textit{generic type} \(T\) I can now write:

  \[
  \textbf{Group}<T>
  \]

  that will \textit{store, operate on, and manage} objects of type \(T\) (\textit{whose type is specified only when the class is instantiated})
Generic Types <T>

```java
public class Group <T>{
    //definition of collection Group goes here
}

// Now, I can instantiate a Group of Product objects
Group<Product> group1 = new Group<Product>();

// ... or a Group of Friend objects
Group<Friend> group2 = new Group<Friend>();

// But I cannot instantiate a generic type T
Group<T> bad_group = new Group<T>();

// And I can be more specific: If I want to store Comparable items:
class Group<T extends Comparable<T>> {
    // declarations and code that manages objects of type T
}
```
Example: Print the [ elements ] of a Stack without destroying the Stack

```java
/** @return String with the contents of stk from top to bottom,
 * assuming that the T on the stack have own toString() method */

public String toString (Stack<T> stk) {
    // Create a temporary stack to hold contents of stk
    Stack<T> tempStack = new Stack<T>();
    String s = "["

    while( !stk.isEmpty() ) {
        T element = stk.pop();
        s = s + element.toString() + " ";
        tempStack.push(element);
    }
    s = s + "]";
    // restore contents of stk
    while( !tempStack.isEmpty() )
    stk.push(tempStack.pop());
    return s;
}
```
Case Study: Check for balanced braces in a program

This is really the first things a compiler is checking during compilation.
Checking for Balanced Braces
(the first thing a compiler is doing)

Source code

```java
public class Motto {
    public static void main (String[] args) {
        System.out.println("Non ministrari");
        System.out.println("sed ministrare");
    }
}
```

What the compiler sees first

```
a { b ( c [ ] d ) { e ( f ) g ( h ) } }
```
Checking for Balanced Braces (the first thing a compiler is doing)

An example of balanced braces
\{a\{b\}c\}

Examples of unbalanced braces
\{a\{bc\} \:
  Too few closing braces
\{ab\}\} : 
  Too many closing braces
\{a(b)c\} : 
  Mismatching braces
Matching Balanced Braces: (Helper Methods)

```java
private boolean open_bracket (char c) {
    return (c == '(') || (c == '{') || (c == '[') || (c == '<');
}

private boolean close_bracket (char c) {
    return (c == ')') || (c == '}') || (c == ']') || (c == '>');
}

private char matching_bracket (char c) {
    if (c == '(') return ')';
    else if (c == '{') return '}
    else if (c == '[') return ']
    else return '>'; }```

Checking Balanced Braces
Pseudocode

while (still more chars to read {

  get next char in the string

  if it is an open_bracket
    then push it on top of the stack

  if it is a close_bracket
    pop char off stack
    check to see if it matches bracket

}
Checking Balanced Braces
Pseudocode

Start by declaring input string balanced
while (still more chars to read && string balanced) {

   get next char in the string

   if it is an open_bracket
      then push it on top of the stack

   if it is a close_bracket
      if stack empty => not balanced
      pop char off stack
      check to see if it matches bracket
      if not matched => not balanced

}  
if stack not empty => not balanced
/** @return true if string S has balanced open and closed brackets */
public boolean isBalanced(String s) {
    Stack<Character> stk = new Stack<Character>();
    int i = 0; char nextChar, top;

    boolean balanced = true;
    while ((i < s.length()) && balanced) {
        nextChar = s.charAt(i); // get the next character in string
        if (open_bracket(nextChar)) // push open brackets onto stack
            stk.push(new Character(nextChar));
        else if (close_bracket(nextChar)) {
            // check whether the matching open bracket is on top of stack
            if (stk.isEmpty()) balanced = false;
            else {
                top = stk.pop().charValue();
                if (nextChar != matching_bracket(top)) balanced = false;
            } // else
        } // else if
        i++;
    } // while
    return (balanced && stk.empty());
} // isBalanced

Checking Balanced Braces

abc{defg{ijk}{l}{mn}}{op}qr : true
[(<>)]: true
[<>()]: false
[{}]: false
Implementing a Stack ADT

Note: Up to now we have been just *using* a stack. For example, the bracket-balancing program is a client of a Stack.

Let’s see how we can implement one!
We can implement our own Library of Data Structures!!
Implementations of a Stack

• A Stack ADT can be implemented using
  – An array: **ArrayStack**
  – A reference-based (linked) list: **LinkedStack**
  – A vector (adjustable-size array with methods that sound like Linked List): **java.util.Stack**

• Stack<T> Interface
  – Provides a common specification for the three implementations

But, why so many different implementations? Why not just one?
ArrayStack:

Stack Implementation using an array (part of javafoundations)
package javafoundations;

public interface Stack<T> {

    /** @param newItem is added to the top of the stack. */
    public void push(T newItem) {

    }

    /** @return the removed top element from the stack. */
    public T pop();

    /** @return the top element without removing it. */
    public T peek();

    /** @return true if the stack contains no elements. */
    public boolean isEmpty();

    /** @return the number of elements in the stack. */
    public int size();

    /** @return a string representation of the stack. */
    public String toString();
}

Let's say you want to use an array to do the following Stack operations:
push(A)
push(F)
pop()
push(B)
push(C)
push(D)
push(F)
pop()

What does the array look like now?

0 1 2 3 4 5 6 7
Array-Based Implementation

- **ArrayStack class**
  - Implements Stack
  - Private data fields (instance variables)
    - stack[]: An array of objects <E>
    - count: the number of items on the stack
    - top: the top of the stack should be at count-1

```
0 1 2 3 4 5 6 7 ... ...
A B C D
```

```
count 4
```
package javafoundations;

package javafoundations.exceptions.*;

public class ArrayStack<T> implements Stack<T> {

    private T[] stack; // Assumes top of stack is at stack[count-1]
    private int count; // Number of items in stack
    private final int DEFAULT_CAPACITY = 10; // Will expand as needed

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

    public boolean isEmpty() {
        return (count == 0);
    }

    public void push(T newItem) {
    }

    public T pop() throws EmptyCollectionException {
        if(count == 0) throw new EmptyCollectionException("Pop failed.");
    }

    public T peek() throws EmptyCollectionException {
        if(count == 0) throw new EmptyCollectionException("Peek failed");
    }
}

}}
Chapter 3 introduced the use of packages and the `import` statement to access package contents.

Packages are used to organize classes by related functionality:
- `java.io` – classes related to input/output
- `java.text` – classes related to text processing
- `java.util` – utility classes

The book organizes the collection classes into a package called `javafoundations`.
Data Structures in javafoundations
java.util.Stack<T>

Java’s API is Using a Vector and many more classes...
public class Stack<E>
extends Vector<E>

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example:

```java
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Since:
JDK1.0

See Also:
Serialized Form
The `java.util.Stack` class was developed mainly as a convenience.

Much of the added functionality comes through inheritance and interface implementation.

A stack is not everything a `Vector` is, so it is not a proper is-a relationship.

It also violates the premise of a well-designed collection class.
The java.util.Stack Class

Java provides its own Stack implementation, but while it contains operations similar to a classic stack, it also contains other, non-Stack methods.

java.util.Stack does not implement any Stack interface.

java.util.Stack provides a search operation that attempts to locate a target element returns its distance from the top of the stack.

java.util.Stack extends the Vector class, which supports direct access to elements at specific indices.

Vector is an adjustable-size array with methods that sound like Linked List:

Vector example = new Vector();
example.add("bob");
example.add(0,"before");
example.get(0);
System.out.println(example.size());
Example of Stack use: How the runtime environment keeps track of Control Flow

We saw that the compiler uses a stack to match parentheses. Cool, eh? Wait until you see how the runtime environment keeps track of your program.
Example of Stack use:
How the runtime environment keeps track of Control Flow

- Recall this discussion about control flow
Consider recursive factorial: What happens when you call \texttt{factorial(500000)}?