Collections

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?
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
- **nonlinear**: elements are organized in something other than a straight line

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

Stack of cafeteria dishes
Stack of game chips
(very fancy) Stack Animation

Adding an element

Removing an element

Top of stack
“Contract” for the Stack Collection

Stack operations

- **Create** an empty 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
public interface Stack<E>
{
    /** Adds an item to the top of a stack. */
    public void push(E newItem);

    /** Removes the top of a stack. */
    public E pop();

    /** Retrieves the top of a stack. */
    public E peek();

    /** Determines whether stack is empty. */
    public boolean isEmpty();

    /** Determines whether stack is empty. */
    public int size();
}

The need for Generic Types <E>

Assume we have defined a Group class that stores and manages a group of objects

- Group could store Objects (which can hold any type – thanks to polymorphism!)
- But it’s no longer clear to the compiler what methods are available for the objects in my group...

Instead, a generic type Group<E> will be able to store, operate on, and manage objects E whose type is not specified until the class is instantiated

```java
public class Group <E>{
    //definition
}
```
Generic Types <E>

- Instantiating a Group of Product objects
  
  ```java
  Group<Product> group1 = new Group<Product>;
  ```

- Instantiating a Group of Friend objects
  
  ```java
  Group<Friend> group2 = new Group<Friend>;
  ```

- You cannot instantiate a generic type E

  ```java
  Group<E> bad_group = new Group<E>;
  ```

- You can be more specific: We want to store Comparable items

  ```java
  class Group<E extends Comparable<E>> {
      // declarations and code that manages objects of type E
  }
  ```
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?
Example: Print a Stack without destroying it

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

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

    while( !stk.isEmpty() ) {
        E 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;
}
Example: Checking for Balanced Braces
(the first thing a compiler is doing)
Example: Checking for Balanced Braces
(the first thing a compiler is doing)

Source code | What the compiler sees first
--- | ---
```
public class Motto {
    public static void main (String[] args) {
        System.out.println("Non ministrari");
        System.out.println("sed ministrare")
    }
}
```

```
a { b ( c [ ] d ) { e ( f ) g ( h ) } }
```
Example: 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

\{f(g)h\} : Mismatching braces
Checking Balanced Braces: Helper Methods

/* returns true if c is an open bracket */
private boolean open_bracket (char c) {
    return (c == '(') || (c == '{') || (c == '[') || (c == '<');
}

/* returns true if c is a close bracket */
private boolean close_bracket (char c) {
    return (c == ')') || (c == '}') || (c == ']') || (c == '>');
}

/* returns the closing bracket matching the input open bracket */
private char matching_bracket (char c) {
    if (c == '(') return ')
    else if (c == '{') return '}
    else if (c == '[') return ']
    else return ‘>’;
}
Checking Balanced Braces

Pseudocode

```plaintext
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

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 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

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** can be implemented using
  - An array
  - A reference-based (linked) list
  - A Vector (adjustable-size array with methods that sound like Linked List)

- **Stack Interface**
  - Provides a common specification for the three implementations
package javafoundations;

public interface Stack<E> {

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

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

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

    /** @return true iff 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();
}
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<E> implements Stack<E> {

    private E[] 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 = (E[]) (new Object[DEFAULT_CAPACITY]);
    }

    public boolean isEmpty() {
    }

    public void push(E newItem) {
    }

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

    public E peek() throws EmptyCollectionException {
    }
}
}
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.
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 and 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:**

```java
Vector<String> example = new Vector<String>();
example.add("bob");
example.add(0,"before");
example.get(0);
System.out.println(example.size());
```
15.5 The `java.util.Stack` Class

- 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.
Example of Stack use:
How the runtime environment keeps track of Control Flow

- Recall this discussion about control flow
public int first(int a, int b) {
    int c;
    ...
    c = second(a+b);
    return c;
}

public int second(int f){
    int g = 10;
    ...
    return (f+g);
}

public static void main(String[] args) {
    ...
    System.out.println(obj.first(1,2));
    ...
}

Consider recursive factorial:
What happens when you call factorial(500000)?
Consider recursive factorial:
What happens when you call factorial(500000)?