The need for Generic Types <T>

- Describing the need by example:
  Define a Group class that stores and manages a group of objects.
  We can define Group to store references to the Object class (allowed by polymorphism).
  However, any type of object could then be stored in our Group, resulting in a loss of control (and an runtime exception).
  A better approach is to define the Group to store a generic type T:
  ```java
  class Group<T> {
      // declarations and code that manages objects of type T
  }
  ```
  Our defined class will then be able to store, operate on, and manage objects whose type is not specified until the class is instantiated.

- Instatiating a Group of Product objects
  ```java
  Group<Product> group1 = new Group<Product>;
  ```
- Instatiating a Group of Friend objects
  ```java
  Group<Friend> group2 = new Group<Friend>;
  ```
  You cannot instantiate a generic type T:
  ```java
  Group<T> bad_group = new Group<T>;
  ```
- You can be more specific: We want our stored items to be Comparable:
  ```java
  class Group<T extends Comparable<T> {
      // declarations and code that manages objects of type T
  }
  ```

Abstract Data Types

- An abstract data type (ADT) is a set of data and the particular operations that are allowed on that data.
- "Abstract" because the operations you can perform on it are separated from the underlying implementation.
- A collection as an ADT
  - For every collection we examine, we should consider:
    - How does the collection operate, conceptually?
    - What operations are included in the interface to the collection?
    - What kinds of problems does the collection help us solve?
    - How might the collection be implemented?
    - How do the implementations compare from an efficiency point of view?

Collections

- A collection is an object that serves as a repository for other objects
  - A collection provides services to add, remove, and manage the elements it contains.
  - The underlying data structure used to implement the collection is independent of the operations provided.
  - 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
  - Ordering of elements, relative to each other, is usually determined by either:
    - the order in which they were added to the collection
    - or some inherent relationship among the elements
Stacks and Queues as 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

The Contract for the Stack Collection

Stack ADT operations Description

<table>
<thead>
<tr>
<th>Description</th>
<th>push</th>
<th>pop</th>
<th>peek</th>
<th>isEmpty</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adds an element to the top of the stack</td>
<td>Removes an element from the top of the stack</td>
<td>Examines the element at the top of the stack</td>
<td>Determines if the stack is empty</td>
<td>Determines the number of elements on the stack</td>
<td></td>
</tr>
</tbody>
</table>

public interface Stack<E>

```java
public void push(E newItem);

public E pop();

public E peek();

public boolean isEmpty();

public int size();
```
Stack Use Example

```java
import java.util.*; // For Java's Stack class
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?
}
```

How do you see all the elements of a stack without destroying it?

Printing a Stack without destroying it!

```java
// returns a String representation of the contents of a Stack from top to bottom, assuming that the E on the Stack have the 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.empty()) {
        E element = stk.pop();
        S = S + element.toString() + "\n";
        tempStack.push(element);
    }
    S = S + "]\n    // restore contents of stk
    while (!tempStack.empty())
        stk.push(tempStack.pop());
    }
    return S;
```
Using CLASSPATH

- CLASSPATH is an "environment variable"
- CLASSPATH stores the list of directories where the compiler and run-time environment looks for classes not found in the API
- CLASSPATH can be set in the shell or as a parameter during program execution
- When a class is required at compile or run-time, if it is not in the API, the CLASSPATH locations are checked for the needed class.
- The -cp option sets the CLASSPATH to look for packages containing needed classes in two locations:
  - First, the package
  - Second, the current working directory (.)
- Locations are separated with semi-colons (Windows) or colons (Unix)

Example: Checking for Balanced Braces

Example:

1. Example of balanced braces: a{b{c}{d}{e}}f
2. Examples of unbalanced braces:
   - (a(b)) Too many closing braces
   - {c(d)} Too few closing braces
   - {(g(h))} Mismatching braces

Checking Balanced Braces: Helper Methods

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

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

/* returns the closing bracket matching the input open bracket */
public 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 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

What could go wrong?

Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A reference-based list
  - The ADT LinkedList
  - The ADT Vector

  Stack Interface
  - Provides a common specification for the three implementations

  StackException
  - Used by Stack Interface
  - Extends java.lang.RuntimeException

abc{defg{k}{l}{mn}}opq: true
[({<})]: true
[[{"}]: false

/* returns true if the string S has balanced open and closed brackets */
public boolean isBalanced(String S) {
    Stack<Character> stk = new Stack<Character>();
    boolean balanced = true;
    char nextC, top;
    while (balanced && (i < S.length())) {
        nextC = S.charAt(i);
        // get the next character in the string
        if (open_bracket(nextC)) {
            stk.push(new Character(nextC));
        } else if (close_bracket(nextC)) {
            // check whether the matching open bracket is on top of stack
            if (stk.empty()) balanced = false;
            else {
                top = stk.pop().charValue();
                if (nextC != matching_bracket(top)) balanced = false;
            }
        }
        i++;
    }
    return (balanced && stk.empty());
}
package javafoundations;

public interface Stack<E> {  
/** Adds the specified element to the top of the stack. */
public void push(E newItem);  
/** Removes and returns the top element from the stack. */
public E pop();  
/** Returns a reference to the top element of this stack without removing it. */
public E peek();  
/** Returns true if the stack contains no elements and false otherwise. */
public boolean isEmpty();  
/** Returns the number of elements in the stack. */
public int size();  
/** Returns a string representation of the stack. */
public String toString();
}

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;
    private final int DEFAULT_CAPACITY = 10; // Will expand as needed
    public ArrayStack() {
        count = 0;
        stack = (E[]) (new Object[DEFAULT_CAPACITY]);
    }
    public boolean isEmpty() {
        return count == 0;
    }
    public void push(E newItem) {
        if (count == stack.length) expandCapacity();
        stack[count++] = newItem;
    }
    public E pop() throws EmptyCollectionException {
        if (count == 0) throw new EmptyCollectionException();
        return stack[--count];
    }
    public E peek() throws EmptyCollectionException {
        if (count == 0) throw new EmptyCollectionException();
        return stack[count - 1];
    }
    public int size() {
        return count;
    }
    public String toString() {
        // Implementation
    }
}

An Array Based Implementation

- ArrayStack class
- Implements Stack
- Private data fields
  - An array of Objects called items
  - The index count
  - Top of stack is at count-1

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

count 4

15.5 – The java.util.Stack Class

While it contains operations similar to a classic stack, it contains other differences

- 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 is based upon 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<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.

Important Application: Execution Stack

```
public static void main(String[] args) {
    int i, j;
    ...
    System.out.println(first(i,j));
    ...
}
```

Consider factorial: What happens when you call fact(150000)?

```
public int first(int a, int b) {
    int c;
    ...
    a = second(c);
    ...
    return third(f, g);
    ...
}
```

```
public int second(int f) {
    int g;
    ...
    return third(f, g);
    ...
}
```

```
public int third(int m, int n) {
    ...
    return n;
    ...
}
```