An abstract data type (ADT) is a set of data and the particular operations that are allowed on that data.

- Abstract means the operations you can perform on it are separated from the underlying implementation.

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?

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

Stacks and Queues as Collections
The Contract for the Stack Collection

Stack operations
- **Create** an empty stack
- **Add** a new item to the stack
- **Remove** from the stack the item that was added most recently
- **Retrieve** (but not remove) from the stack the item that was added most recently
- Determine whether a stack is **empty**

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

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();
}

Assume we have defined a **Group** class that stores and manages a group of objects
- **Group** could store **Objects** (which can hold any type – polymorphism!)
- But it’s no longer clear what methods are available for the objects in my group...

(Die) (myGroup.getMember()).roll(); ← Loss of control, and awkward. What if it’s not a Die object?

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

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

```
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 can we print all the elements of a stack without destroying it?

```java
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;
}
```

An example of balanced braces

```
a(b|c|d|e|f)g
```

Examples of unbalanced braces

```
a(b): Too many closing braces
c|d|e: Too few closing braces
[f|g|h]: Mismatching braces
```

<table>
<thead>
<tr>
<th>Input string</th>
<th>Stack as algorithm executes</th>
</tr>
</thead>
<tbody>
<tr>
<td>`{4}{b}</td>
<td>e</td>
</tr>
<tr>
<td>`{4}{b}</td>
<td>g</td>
</tr>
<tr>
<td>`{4}{b}</td>
<td>c</td>
</tr>
</tbody>
</table>
/* 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 '>

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

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
}

@returns true if string S has balanced open and closed brackets */
ic boolean isBalanced (String s) {
    Stack <Character> stk = new Stack <Character>();
    int i = 0; char nextChar, top; boolean balanced = true;
    while (balanced && (i < s.length())) {
        nextChar = s.charAt(i); // get the next character in the :
        if (open_bracket(nextChar)) // push open brackets onto the st:
            stk.push(new Character(nextChar));
        else if (close_bracket(nextChar)) {
            // check whether the matching open bracket is on top of :
            if (stk.isEmpty())
                balanced = false;
            else {
                top = stk.pop().charValue();
                if (nextChar != matching_bracket(top)) balanced = false;
            }
    }
    i++;
}
return (balanced && stk.empty());
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

```java
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 top element of this stack without removing */
    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();
}
```

**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

![Array-Based Implementation Diagram](image)

Note: Up to now we have been *using* a stack. Let's see how it works inside.
package javafoundations;

give javafoundations.exceptions.*;

c 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) {
    if (count == stack.length) expandCapacity();
}

public E pop() throws EmptyCollectionException {
}

public E peek() throws EmptyCollectionException {
}

* While it contains operations similar to a classic stack, it 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<String> example = new Vector<String>();
example.add("bob");
example.add(0,"before");
example.get(0);
System.out.println(example.size());

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

• Recall this discussion about control flow
Consider recursive factorial: What happens when you call `factorial(500000)`?

**WARNING**

- THERE ARE 2 TYPES OF PEOPLE.
  THOSE WHO WILL PAY ATTENTION TO THE NEXT 2 SLIDES AND THOSE WHO WILL NOT

- THE FIRST KIND WILL LIVE A HAPPY PRODUCTIVE LIFE AS POWERFUL PROGRAMMERS

- THE SECOND TYPE MAY DROWN IN THEIR OWN TEARS BEFORE DROPPING OUT OF CS

- YOU HAVE BEEN WARNED!
Because we import javafoundations, we need to tell Java where to find this package:

```bash
$ echo $CLASSPATH
.
$ CLASSPATH=/Users/tm/:.
$ export CLASSPATH
$ echo $CLASSPATH
/Users/tm/:
```

Or we can compile and run with the full path:

```bash
$ javac -cp /Users/tm/: Test.java
$ java -cp /Users/tm/: Test
```

But make sure you have implemented all the missing methods!!