Chapter 3
The Stack ADT

Homework

- Solutions are on-line.
- Exam talk.

Lab

- Creating project files
  - We all need to be on the same page
  - Create a new JAVA project called CIS113
  - Within CIS113 create packages for all the folders found in bookFiles
    - bookFiles is contained within CIS113_SourceCode.zip
  - Now we will look at the remaining subfolders and fix errors
- Expected result: a functional project with all of the book files.

Chapter 3: The Stack ADT

3.1 – Stacks

3.2 – Collection Elements

3.3 – Exceptional Situations

3.4 – Formal Specification

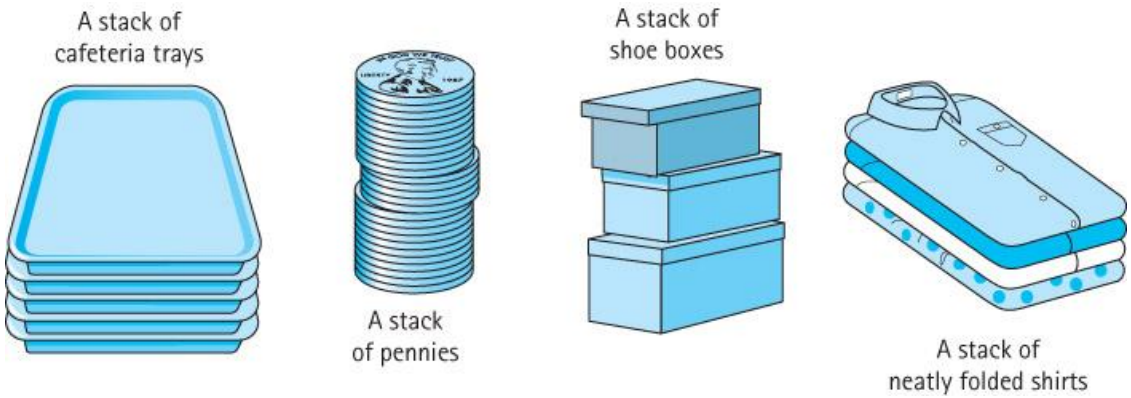
3.5 – Array-Based Implementation

3.6 – Application: Well-Formed Expressions

3.7 – Link-Based Implementation

3.8 – Case Study: Postfix Expression Evaluator

3.1 Stacks

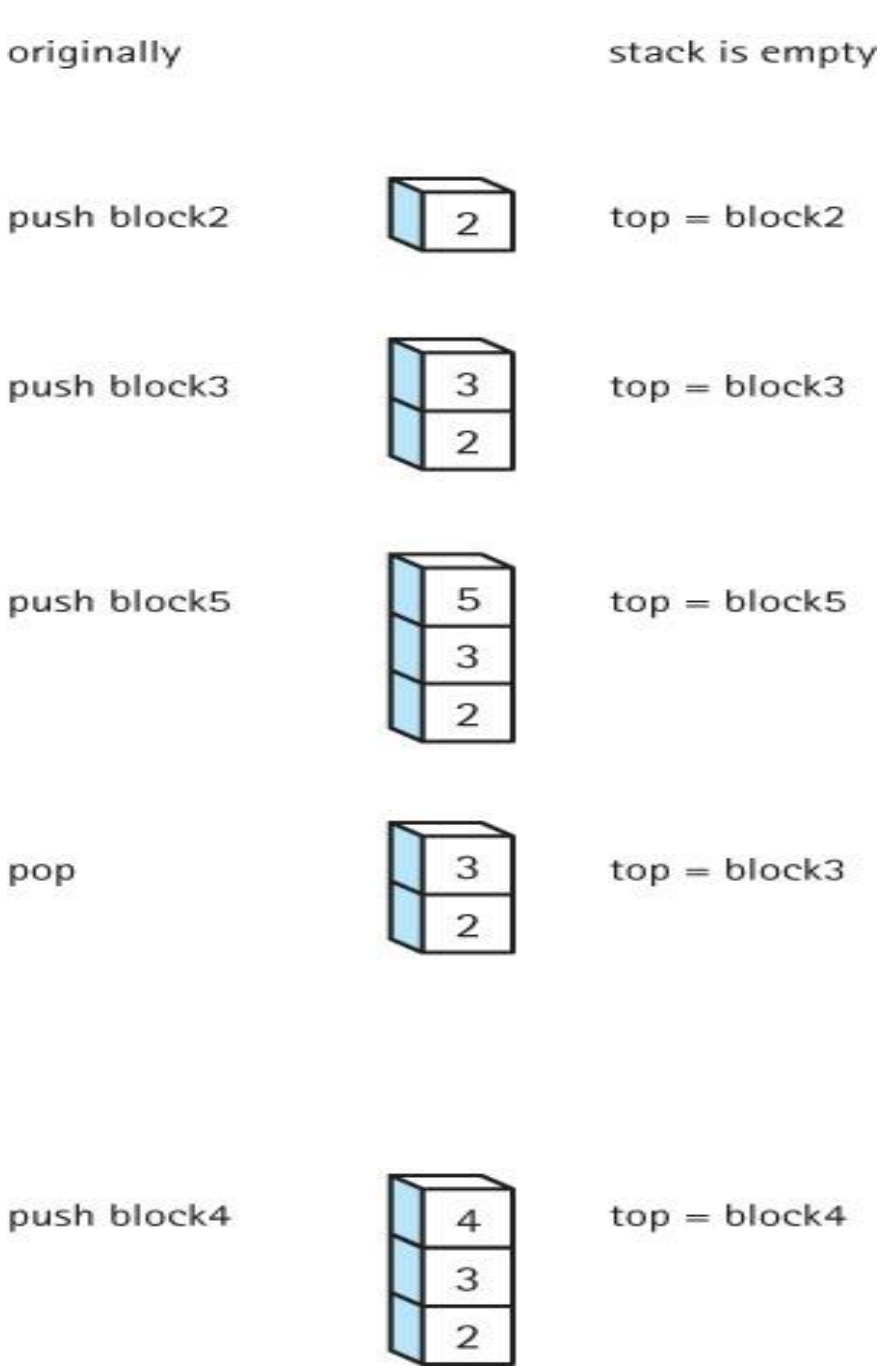


- **Stack** A structure in which elements are added and removed from only one end
- (LIFO) structure: “Last In First Out”
- Demo stack-based bubble sort using playing cards.
- The betting trick…. and more extra credit

Operations on Stacks

- Constructor
  - new - creates an empty stack
- Transformers
  - push - adds an element to the top of a stack
  - pop - removes the top element off the stack
- Observer
  - top - returns the top element of a stack

Effects of Stack Operations



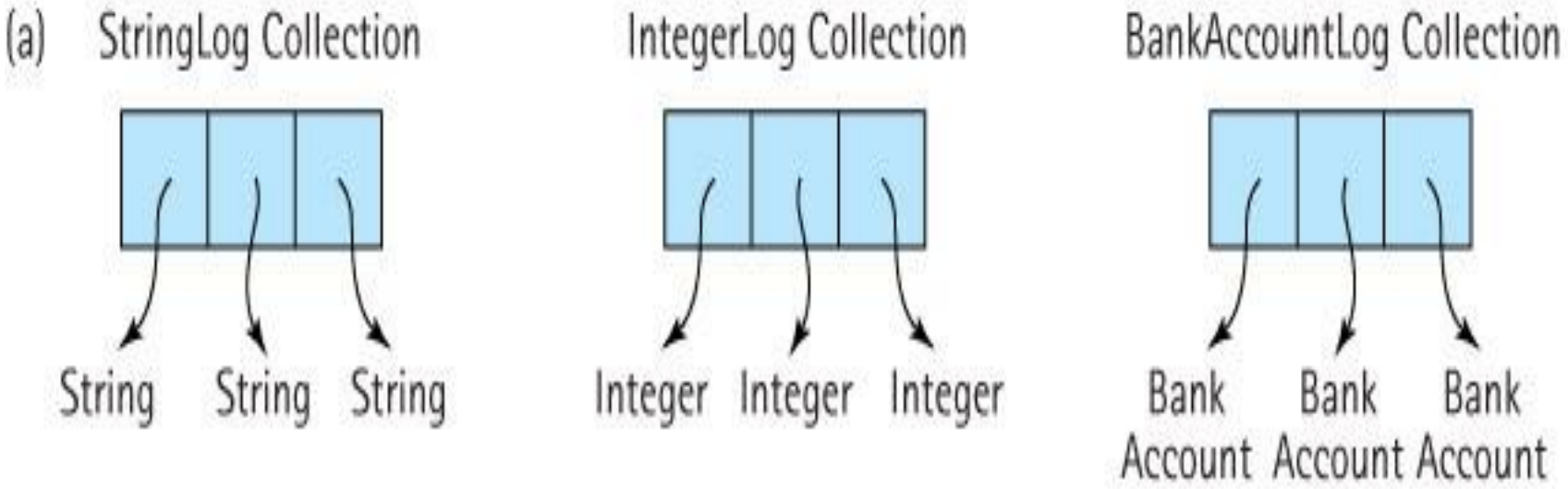
Using Stacks

- What is the most visible use of the stack?
  - The undo command ctrl-z
- Also, commonly used for “system” programming:
  - Programming language systems
    - Stack to keeps track of sequences of operation calls
  - Compilers
    - Use stacks to analyze nested language statements
  - Operating systems
    - save information about the current executing process on a stack, so that it can work on a higher-priority, interrupting process

3.2 Collection Elements

- **Collection**
  - An object that holds other objects.
  - Interested in inserting, removing, and iterating through the contents of a collection.
- A stack is an example of a Collection ADT.
  - It collects together elements for future use
  - Maintains LIFO ordering among the elements

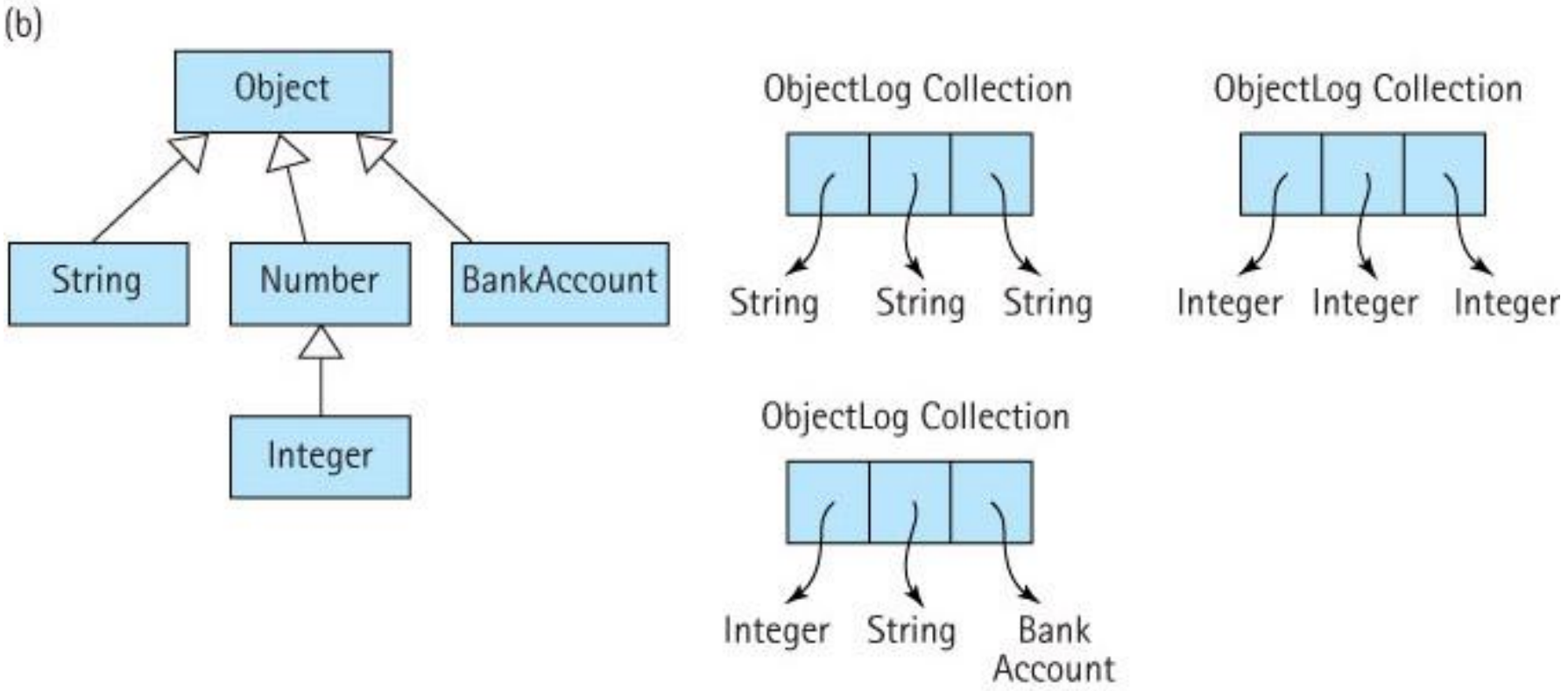
Separate ADTs for each type that a collection can hold?



This approach is not useful… Why?

It is too rigid, requires more work (more code -> more debugging)!

Collections of Class Object



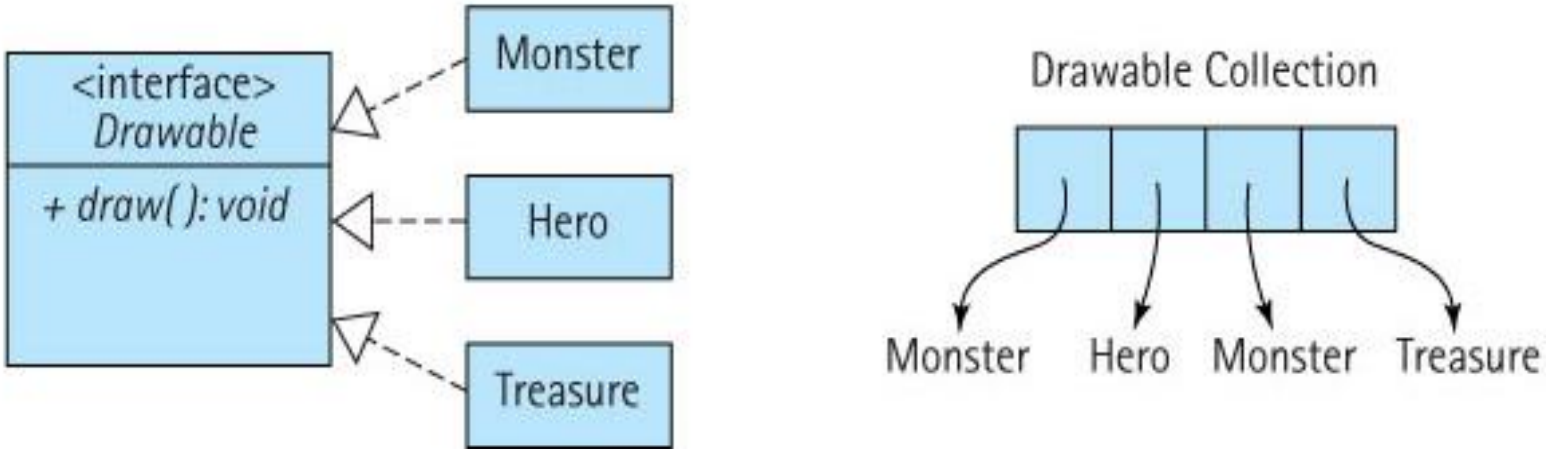
Collections of Class Object

- When an element is removed from a collection it can only be referenced as an Object.
  - If it is to be used as something else, it must be cast into the type that is intended to be use.
- For example:

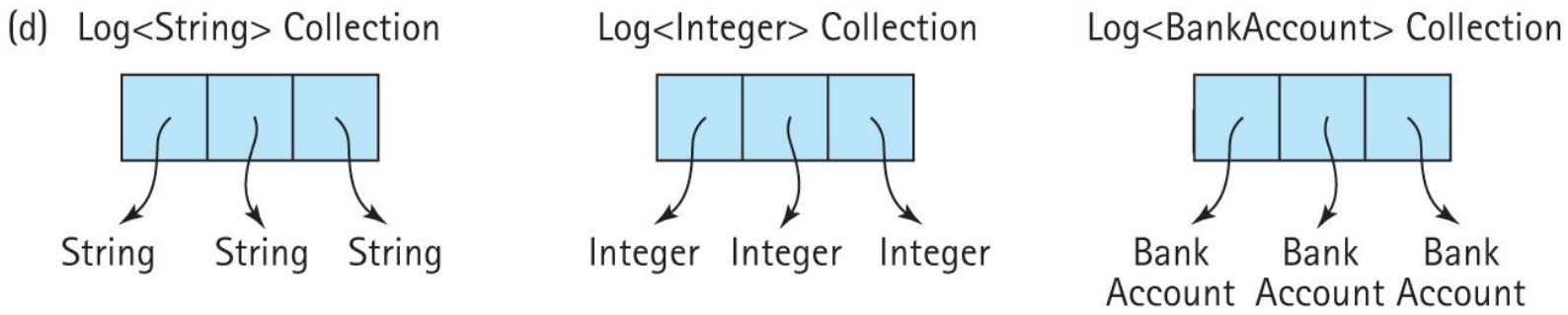
```
collection.push("E. E. Cummings");          // push string on a stack
String poet = (String) collection.top();    // cast top to String
System.out.println(poet.toLowerCase());     // use the string
```

Collections of a Class that implements a particular Interface

(c)



Generic Collections



- Parameterized types
- Declared as <T>
- Actual type provided upon declaration / instantiation

Generic Collections

- Example of collection class definition:

```
public class log<T>
{
    private T[ ] log; // array that holds objects of class T
    . . .
}
```

- Example of application level declarations:

```
log<Integer> numbers;
log<BankAccount> investments;
log<String> answers;
```

Agile - Test Driven Development

- Mark Dexter lesson 6

3.3 Exceptional Situations

- **Exceptional situation**
  - Associated with an unusual, sometimes unpredictable event
  - Detectable by software or hardware
  - Requires special processing
  - The event may or may not be erroneous.
- For example:
  - A user enters an input value of the wrong type
  - While reading information from a file, the end of the file is reached
  - A user presses a control key combination
  - An illegal mathematical operation occurs, such as divide-by-zero
  - An impossible operation is requested of an ADT
    - An attempt to *pop* an empty stack

Exceptions with Java

- The Java exception mechanism has three major parts:
  - Defining the exception
    - usually as a subclass of Java's Exception class
  - Generating (raising) the exception
    - by recognizing the exceptional situation and then using Java's throw statement to "announce" that the exception has occurred
  - Handling the exception
    - using Java's try – catch statement to discover that an exception has been thrown and then take the appropriate action

Exceptions and ADTs An Example

- Modify the constructor of the Date class to throw an exception if it is passed an illegal date
- First, we create our own exception class
  - Customary to define the constructors so that they call (mirror) the corresponding constructors of the superclass

```
public class DateOutOfBoundsException extends Exception
{
  public DateOutOfBoundsException()
  {
    super();
  }

  public DateOutOfBoundsException(String message)
  {
    super(message);
  }
}
```

Example: a constructor that throws the exception:

```
public Date(int newMonth, int newDay, int newYear)
           throws DateOutOfBoundsException
{
  if ((newMonth <= 0) || (newMonth > 12))
    throw new DateOutOfBoundsException("month " + newMonth + "out of range");
  else
    month = newMonth;

  day = newDay;

  if (newYear < MINYEAR)
    throw new DateOutOfBoundsException("year " + newYear + " is too early");
  else
    year = newYear;
 }
```

Example: a program that throws the exception out to interpreter to handle:

```
public class UseDates
{
  public static void main(String[] args)
                          throws DateOutOfBoundsException
  {
    Date theDate;
    // Program prompts user for a date
    // M is set equal to user's month
    // D is set equal to user's day
    // Y is set equal to user's year
    theDate = new Date(M, D, Y);

    // Program continues ...
  }
}
```

The interpreter will stop the program and print an "exception" message, for example

Exception in thread "main" DateOutOfBoundsException: year 1051 is too early
at Date.<init>(Date.java:18)
at UseDates.main(UseDates.java:57)

Example: a program that catches and handles the exception:

```
public class UseDates
{
  public static void main(String[] args)
  {
    Date theDate;
    boolean DateOK = false;

    while (!DateOK)
    {
      // Program prompts user for a date
      // M is set equal to user's month
      // D is set equal to user's day
      // Y is set equal to user's year
      try
      {
        theDate = new Date(M, D, Y);
        DateOK = true;
      }
      catch(DateOutOfBoundsException DateOBExcept)
      {
        output.println(DateOBExcept.getMessage());
      }
    }

    // Program continues ...
  }
}
```

General guidelines for using exceptions

- Exceptions may be handled any place in the software hierarchy
  - in the module where it is first detected through the top level of the program.
- Unhandled built-in exceptions carry a big penalty
  - PROGRAM TERMINATION
- In applications exceptions should be handled as design decisions
  - It is best to handle exceptions at what program level?
    - **At the level of the program that knows what the exception means.**
- An exception need not be fatal
  - That is the program may not have to terminate
- Non-fatal exceptions
  - the execution thread can continue from various points in the program, but execution should continue from the lowest level that can recover from the exception.

Java RunTimeException class

- Exceptions of this class are thrown when a standard run-time program error occurs. (Name 2…)
  - division-by-zero
  - array-index-out-of-bounds
- Exceptions can happen in virtually any method or segment of code…
  - We are not required to explicitly handle them.
  - Classified as **unchecked exceptions**.

Error Situations and ADTs

- Options for dealing with errors within our ADT methods
  - Detect and handle the error within the method
    - **Best approach**
    - If the error can be handled internally and if it does not greatly complicate design.
  - Detect the error within the method
    - Force the calling method to deal with the exception by throwing an exception related to the error
    - If it is not clear how to handle a particular error situation, throw the exception at a level where it can be handled
  - Ignore the error situation
    - Recall the “programming by contract”
    - If the preconditions of a method are not met, the method is not responsible for the consequences.

More Mark Dexter

- Lessons 7

Completing the Formal Specification of the StackADT

- We need to
  - Identify and address any exceptional situations
  - Determine boundedness
  - Define the Stack interface or interfaces

3.4 Formal Specification (of our Stack ADT)

- Recall Stacks are LIFO
  - push: adds an element to the top of the stack
  - pop: removes the top element off the stack
  - top: returns the top element of a stack
- A constructor is needed to create an empty stack
- Our Stack ADT will be a generic stack
  - The class of elements that a stack stores will be specified by the client code at the time the stack is instantiated
    - Here, use <T> to represent the class of objects stored in our stack

Exceptional Situations

- pop and top – what if the stack is empty?
  - Detect the situation and throw an exception
    - StackUnderflowException
  - Handle within the method
    - define an observer isEmpty method
- push – what if the stack is full?
  - Detect the situation and throw an exception
    - throw a StackOverflowException
  - Handle within the method
    - define an isFull method for use by the application

StackInterface

```
package ch03.stacks;

public interface StackInterface<T>
{
  // Throws StackUnderflowException if this stack is empty,
  // otherwise removes top element from this stack.
  void pop() throws StackUnderflowException;

  // Throws StackUnderflowException if this stack is empty,
  // otherwise returns top element from this stack.
  T top() throws StackUnderflowException;

 // Returns true if this stack is empty, otherwise returns false.
 boolean isEmpty();

}
```

Boundedness

- We support two versions of the Stack ADT
  - a bounded version
  - an unbounded version
- We define three interfaces
  - StackInterface: (defined on the previous slide)
    - features of a stack not affected by boundedness
  - BoundedStackInterface:
    - features specific to a bounded stack
  - UnboundedStackInterface:
    - features specific to an unbounded stack

The Remaining Stack Interfaces

The BoundedStackInterface

```
package ch03.stacks;

public interface BoundedStackInterface<T> extends StackInterface<T>
{
  // Throws StackOverflowException if this stack is full,
  // otherwise places element at the top of this stack.
  public void push(T element) throws StackOverflowException;

  // Returns true if this stack is full, otherwise returns false.
  public boolean isFull();
}
```

The UnboundedStackInterface

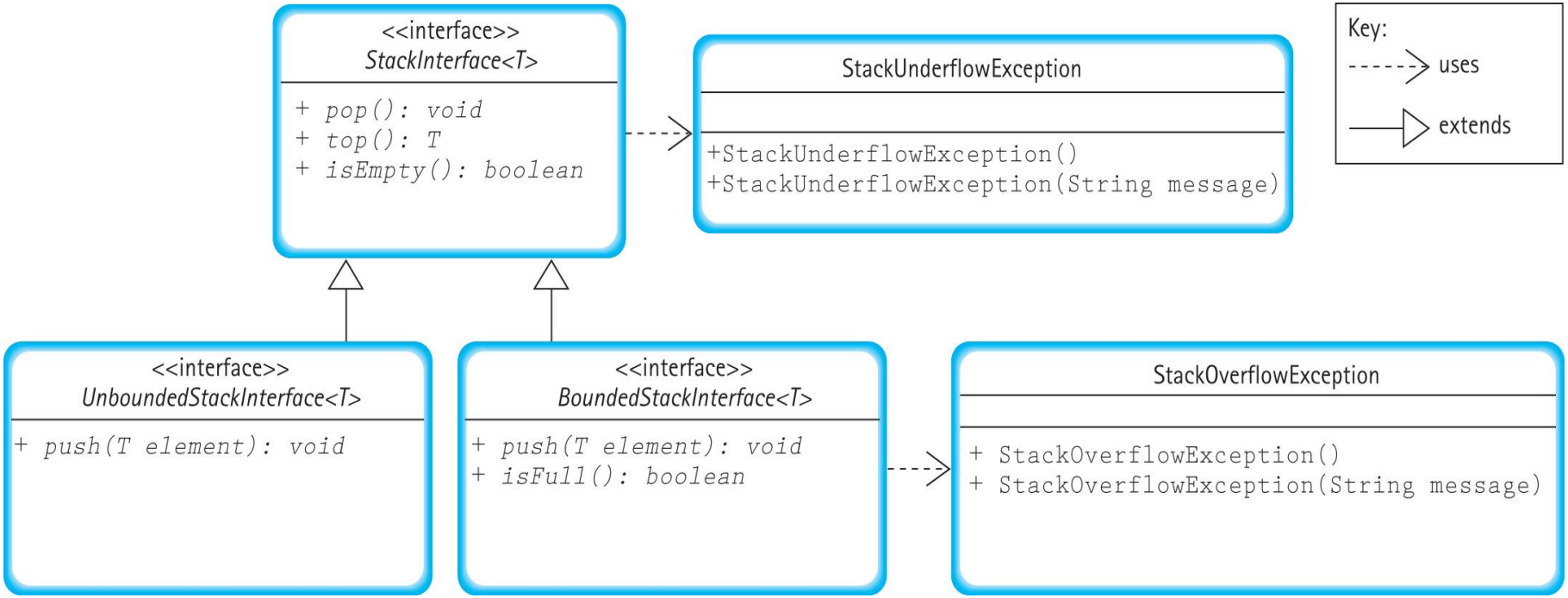
```
package ch03.stacks;

public interface UnboundedStackInterface<T> extends StackInterface<T>
{
  // Places element at the top of this stack.
  public void push(T element);
}
```

Inheritance of Interfaces

- **Inheritance of interfaces**
  - A Java interface can extend another Java interface, inheriting its requirements.
  - If interface B extends interface A, then classes that implement interface B must also implement interface A.
  - Usually, interface B adds abstract methods to those required by interface A.

Relationships among Stack Interfaces and Exception Classes



Is this an example of multiple inheritance of interfaces?

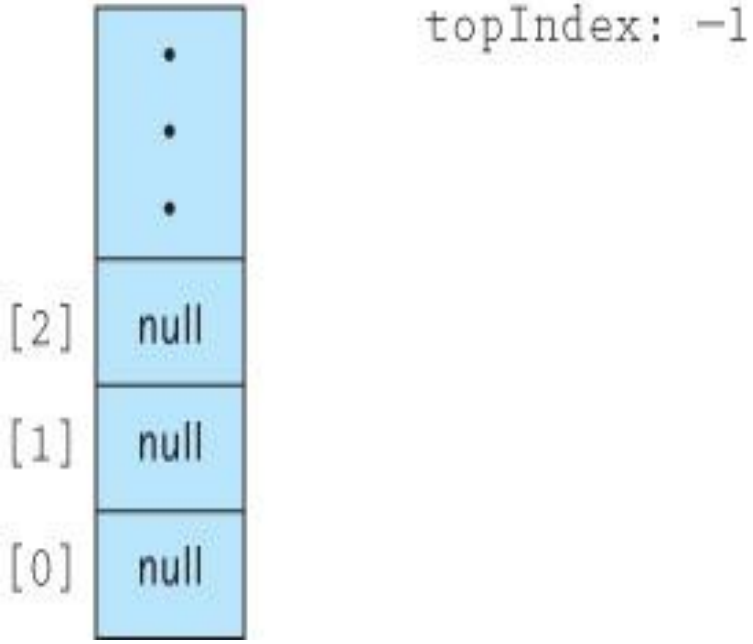
No, here two interfaces inherit the properties of a single interface
Multiple inheritance: if a single interface inherited from more than one interface

3.5 Array-Based Implementations

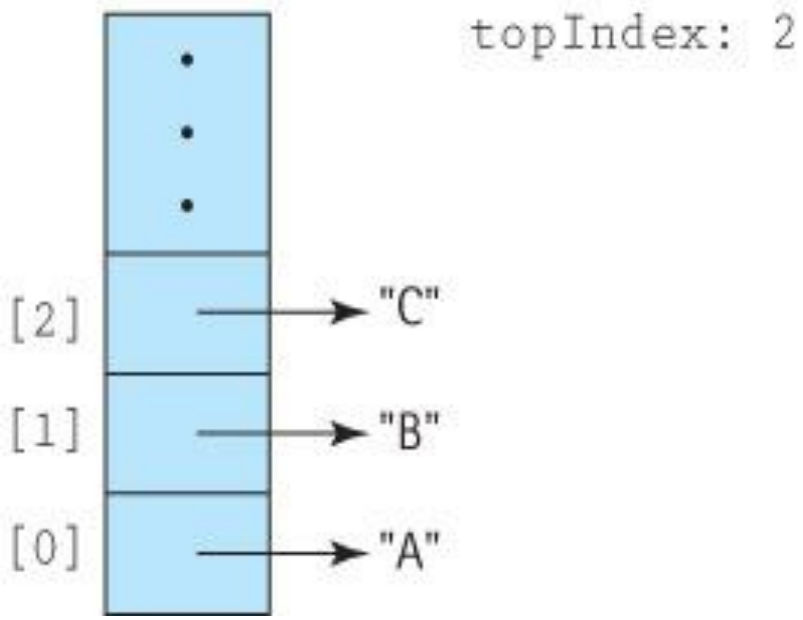
1. Our array-based Stack ADT.
2. The built in Java Library ArrayList class.

Visualizing the stack

- The empty stack:



- After pushing “A”, “B” and “C”:



Brief Lab – Overview/Demo

- Look at ReverseStrings
  - Array
  - LinkedList
- ReverseStrings2
  - Java Stack Class and the Collections Framework
  - P184-185

The ArrayStack Class

Constructors

T is specified by the client class using the bounded stack

```
package ch03.stacks;

public class ArrayStack<T> implements BoundedStackInterface<T>
{
  protected final int defCap = 100; // default capacity
  protected T[] stack;              // holds stack elements
  protected int topIndex = -1;      // index of top element in stack
```

Must instantiate the array of class object and cast array elements into class T

```
  public ArrayStack()
  {
    stack = (T[]) new Object[defCap];
  }

  public ArrayStack(int maxSize)
  {
    stack = (T[]) new Object[maxSize];
  }
…
```

Definitions of Stack Operations

```
public boolean isEmpty()
// Returns true if this stack is empty, otherwise returns false.
{
  if (topIndex == -1)
    return true;
  else
    return false;
}
```

```
public boolean isFull()
// Returns true if this stack is full, otherwise returns false.
{
  if (topIndex == (stack.length - 1))
    return true;
  else
    return false;
}
```

Definitions of Stack Operations

```
public void push(T element)
{
  if (!isFull())
  {
    topIndex++;
    stack[topIndex] = element;
  }
  else
    throw new StackOverflowException("Push attempted on a full stack.");
}
```

```
public void pop()
{
  if (!isEmpty())
  {
    stack[topIndex] = null;
    topIndex--;
  }
  else
    throw new StackUnderflowException("Pop attempted on an empty stack.");
}
```

Definitions of Stack Operations

```
public T top()
// Throws StackUnderflowException if this stack is empty,
// otherwise returns top element from this stack.
{
  T topOfStack = null;
  if (!isEmpty())
    topOfStack = stack[topIndex];
  else
    throw new StackUnderflowException("Top attempted on an empty stack.");
  return topOfStack;
}
```

Lab – Implement a test plan

- Set the stack size on ArrayStack.java to 5
- Implement tests on page 191

- Exam 1
- Chapters 1-3
- Three part
  - 1. Multiple guess / True False
  - 2. Short Answer
  - 3. Short Programming

- A study guides for CH1-3 are available on myAVC and Blackboard at the same level as the syllabus. Most of the questions will come from the study guides!

Homework – Due March 10th

- **Create a class called Card**
  - Card 'has a' value (type integer)
    - 2,3,4,5,6,7,8,9,10,11,12,13,14
    - Note: 11 (Jack), 12 (Queen), 13 (King), 14 (Ace)
  - Card 'has a' suit (type string)
    - Heart, Diamond, Spade, Clover
  - Create an overriding toString method which prints
    - Card value = 14, suit = "Hearts" : 'Ace of Hearts'

- **Create a class called Deck**
  - Deck will inherit Card
  - Deck 'has a' instance variable named 'deckStack'
    - Use the ArrayListStack
    - The deck should hold 52 cards
  - Default constructor should create a deck of 52 unique cards in order
    - Hearts: 2 – 14
    - Diamonds: 2-14
    - Spades: 2-14
    - Clovers: 2-14
  - Create a method DisplayCards
    - Prints out the entire deck with one card per line
    - Example: Ace of Hearts
2 of Clovers
King of Spades
…
  - Create a method ShuffleCards
    - Randomizes the order of cards
    - See next page for algorithm…
- **Create a test driver and display the unshuffled and shuffled decks**

Algorithm for Shuffle Cards

1. Create two ArrayListStack objects of type Card: fStack, rStack

2. Pop/Push all the data from cards(instance variable) to rStack.
   rStack should now be loaded with all the data from cards, but in reverse order

3. Loop: while (rStack is not empty)
   - Let j = a random number within the range [1 to i]
   - while (rStack is not empty)
     - if jth iteration then Pop from rStack to deckStack
     - else Pop rStack and Push to fStack
   - while (fStack is not empty)
     - Pop from rStack and Push to fStack