CS 304 Lecture 3

The **`Stack`** ADT

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

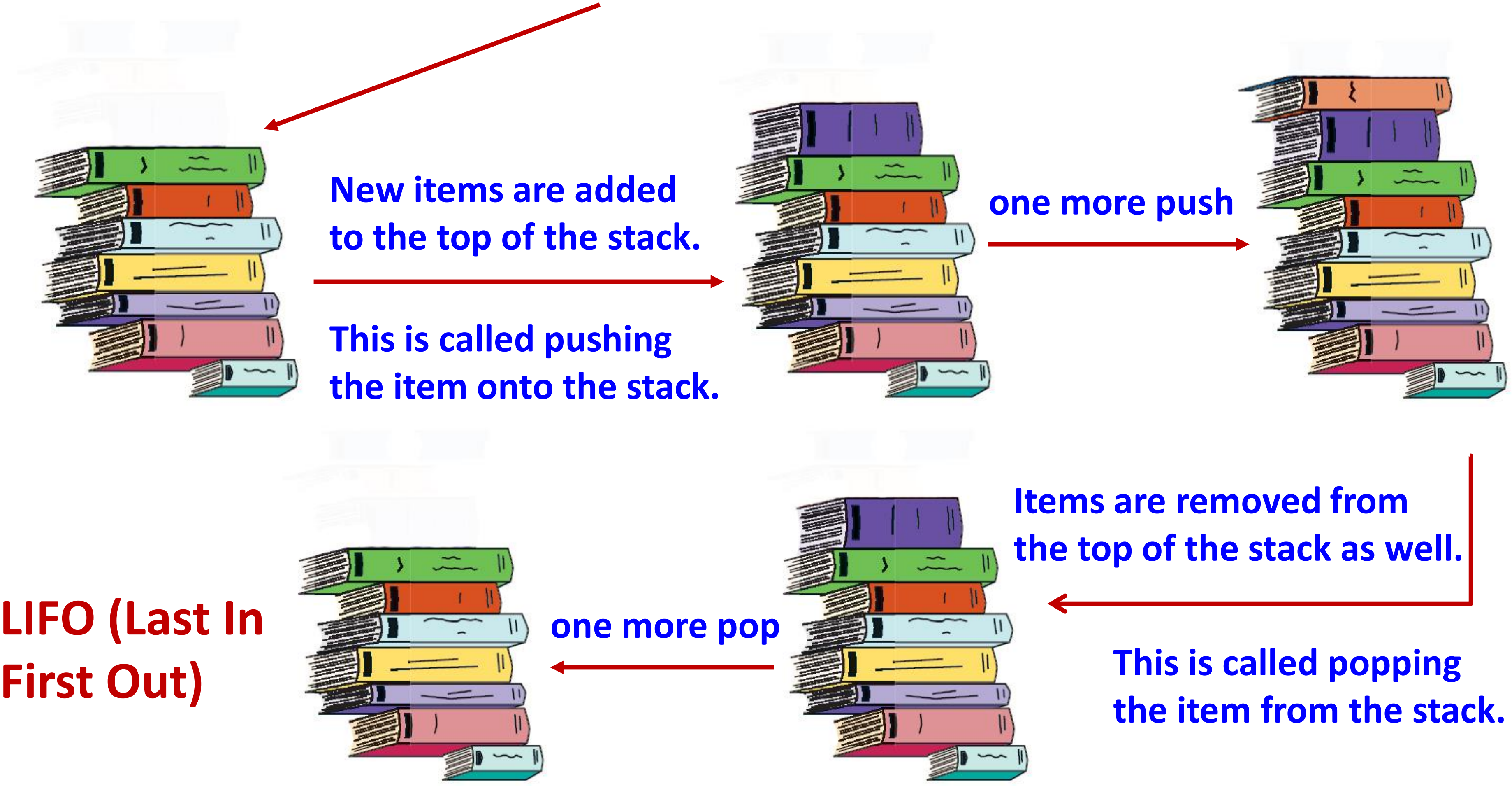
Northeastern Illinois University

Chicago, Illinois, 60625

8 September 2016

Stacks

- A stack lets you insert and remove elements at one end only, traditionally called the top of the stack.



New items are added to the top of the stack.

This is called pushing the item onto the stack.

one more push

Items are removed from the top of the stack as well.

This is called popping the item from the stack.

one more pop

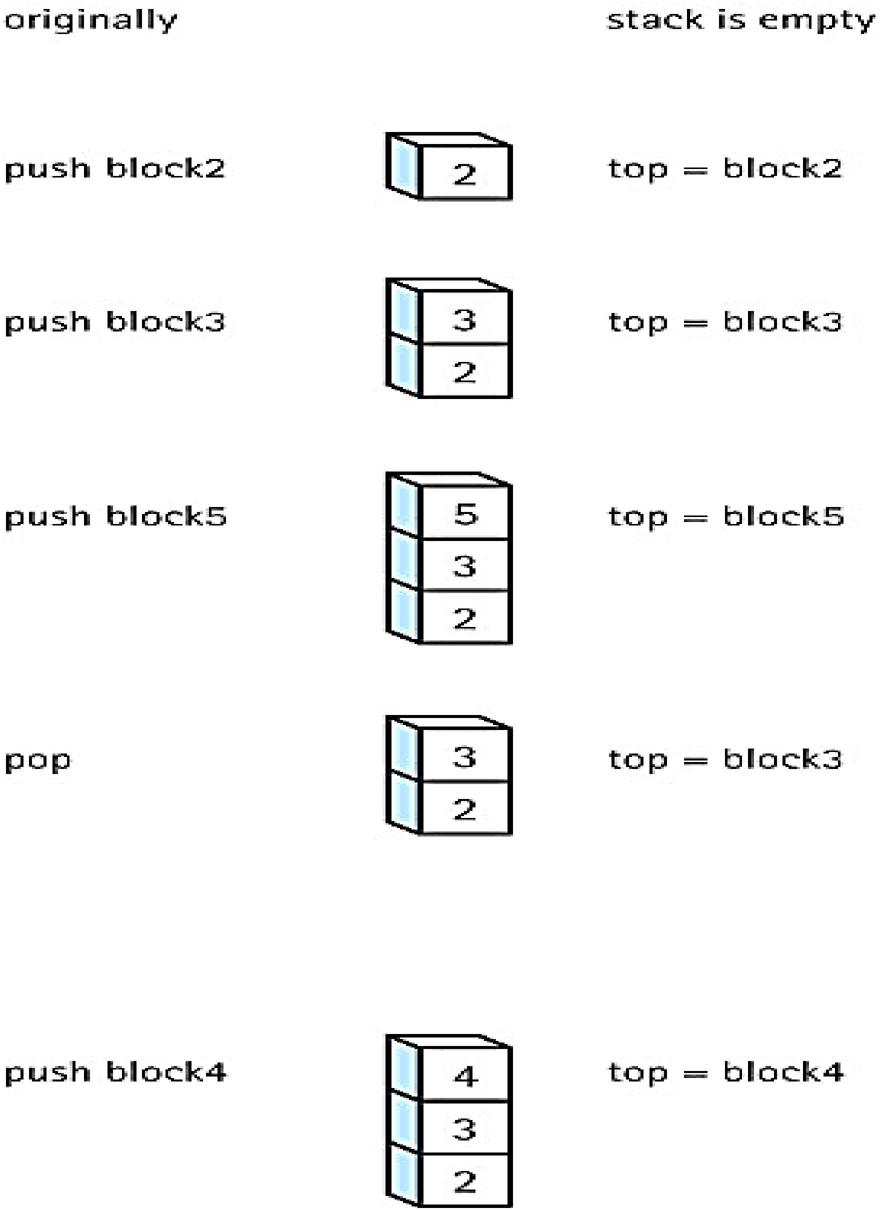
LIFO (Last In First Out)

Stack operations

- **push** - adds an element to the top of a stack

- **pop** - removes the top element off the stack

- **top** - returns the top element of a stack

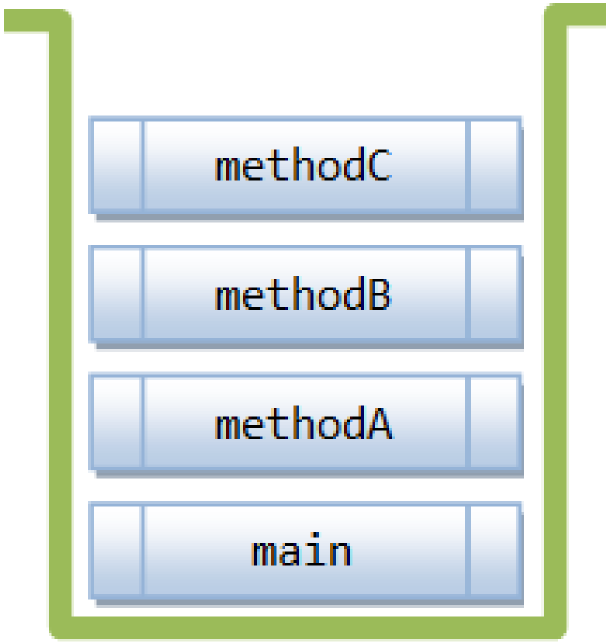


Using stacks

- Examples of stacks in real life situations
    - tennis balls in their container
    - a pile of plates in a restaurant
    - potato chips in a Pringles tube
- Stacks are often used for "system" programming:
    - Programming language systems use a stack to keep 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.

Method call stack

- A typical application involves many levels of method calls, which is managed by a so-called method call stack.
  - In the following example, the **`main`** method invokes **`methodA`**; **`methodA`** calls **`methodB`**; and **`methodB`** calls **`methodC`**.



Method Call Stack
(Last-in-First-out Queue)

Method call stack

```
public class MethodCallStackDemo
{
   public static void main(String[] args)
   {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit main()");
   }

   public static void methodA()
   {
      System.out.println("Enter methodA()");
      methodB();
      System.out.println("Exit methodA()");
   }

   ...
```

Method call stack

```
    ...

    public static void methodB() {
       System.out.println("Enter methodB()");
       methodC();
       System.out.println("Exit methodB()");
    }

    public static void methodC() {
       System.out.println("Enter methodC()");
       System.out.println("Exit methodC()");
    }
 }
```

Exceptional situations

- **Exceptional situation** - Associated with an unusual, sometimes unpredictable event, detectable by software or hardware, which 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
  - an illegal mathematical operation occurs, such as divide-by-zero
  - an impossible operation is requested of an ADT, such as an attempt to pop an empty stack

Exceptions in Java

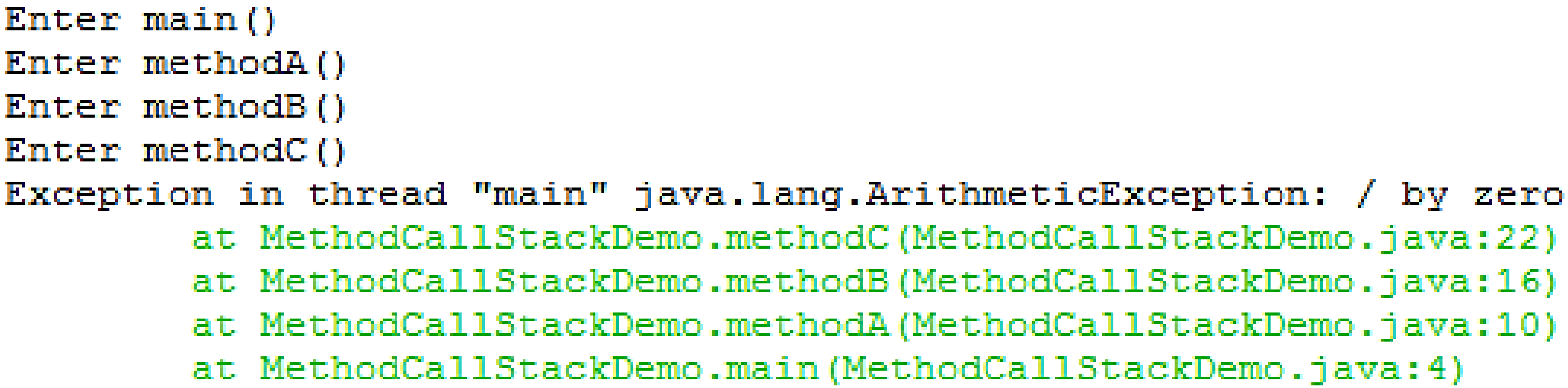
- The Java exception mechanism has three major parts:
  - **Defining the exception** – usually as a subclass of Java's Exception class.
  - **Generating (raising) the exception** – by recognizing the exceptional situation and then using Java's throw statement to "announce" that the exception has occurred.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

Method call stack and exceptions

- Suppose that we modify `methodC()` to carry out a "divide-by-0" operation, which triggers an `ArithmeticException`:

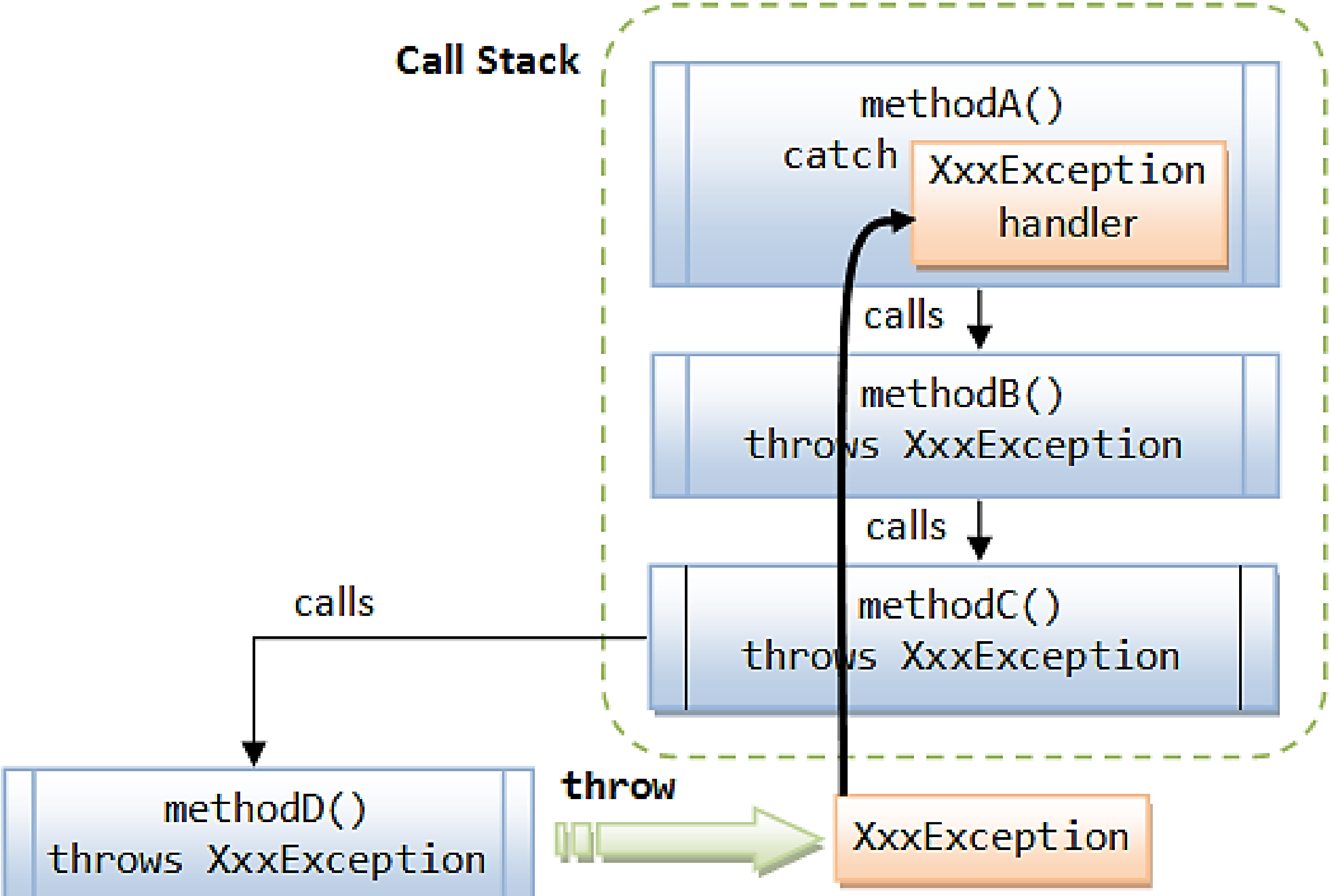
```
 public static void methodC() {
   System.out.println("Enter methodC()");
   System.out.println(1 / 0);  // divide-by-0 triggers an
                               // ArithmeticException
   System.out.println("Exit methodC()");
}
```

- The exception message will be as follows



```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)
        at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)
        at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10)
        at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

Method call stack and exceptions



JVM searches the call stack *backward* for a *matching* exception handler

Exceptions and ADTs

- We modify the constructor of our **`Date`** class to throw an exception if it is passed an illegal date.
- First, we create our own exception class:

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

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

Exceptions and ADTs

- Here is an example of 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;
 }
```

Exceptions and ADTs

- Here is an example of a program that catches and handles the exception:

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

    while (!DateOK){
      // Read and set M, D, and Y
      try{
        theDate = new Date(M, D, Y);
        DateOK = true;
      }
      catch (DateOutOfBoundsException DateOBExcept){
        output.println(DateOBExcept.getMessage());
      }
    }
    // Program continues ...
  }
}
```

General guidelines for using exceptions

- An exception may be handled any place in the software hierarchy
- Unhandled built-in exceptions carry the penalty of program termination.
- Exceptions should always be handled at a level that knows what the exception means.
- An exception need not be fatal.
- For non-fatal exceptions, the thread of execution can continue from various points in the program, but execution should continue from the lowest level that can recover from the exception.

Exceptions in the `Stack` ADT

- Recall the methods that are required by our `Stack` ADT:
  - `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 - creates an empty stack
- Our `Stack` ADT will be a generic stack – the element can be of any type.
- In addition, we need to
  - identify and address any exceptional situations;
  - determine boundedness;
  - define the `Stack` interface or interfaces.

Exceptional situations

- **`pop`** and **`top`** – what if the stack is empty?
  - throw a **`StackUnderflowException`**
  - plus define an **`isEmpty`** method for use by the application.
- **`push`** – what if the stack is full?
  - throw a **`StackOverflowException`**
  - plus define an **`isFull`** method for use by the application.

Boundedness

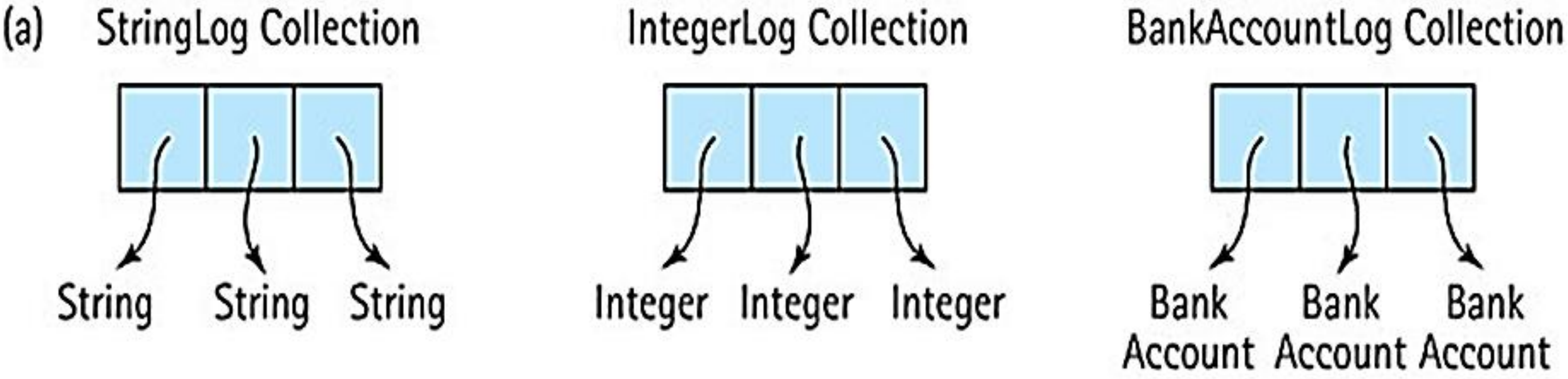
- We support two versions of the `Stack` ADT: a bounded version and an unbounded version.
- We define three interfaces
  - `StackInterface`: features of a stack not affected by boundedness
  - `BoundedStackInterface`: features specific to a bounded stack
  - `UnboundedStackInterface`: features specific to an unbounded stack
- **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.

The interfaces of the **Stack** ADT

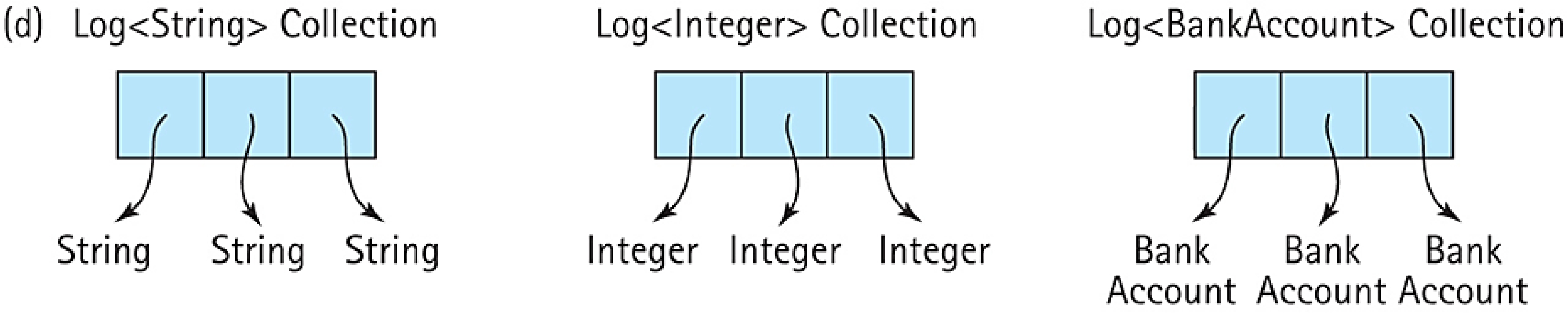
```
public interface StackInterface<T>
{
  void pop() throws StackUnderflowException;
  // Throws StackUnderflowException if this stack is empty,
  // otherwise removes top element from this stack.

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

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

Collection elements

- A stack is an example of a **`Collection`** ADT. It collects together elements for future use, while maintaining a LIFO ordering among the elements.
- Do we need separate ADTs for each type that a collection can hold?
  - This is too redundant and not useful.



Generic collections



- Parameterized types, declared as **<T>**, actual type provided upon instantiation.
- Example of collection class definition:

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

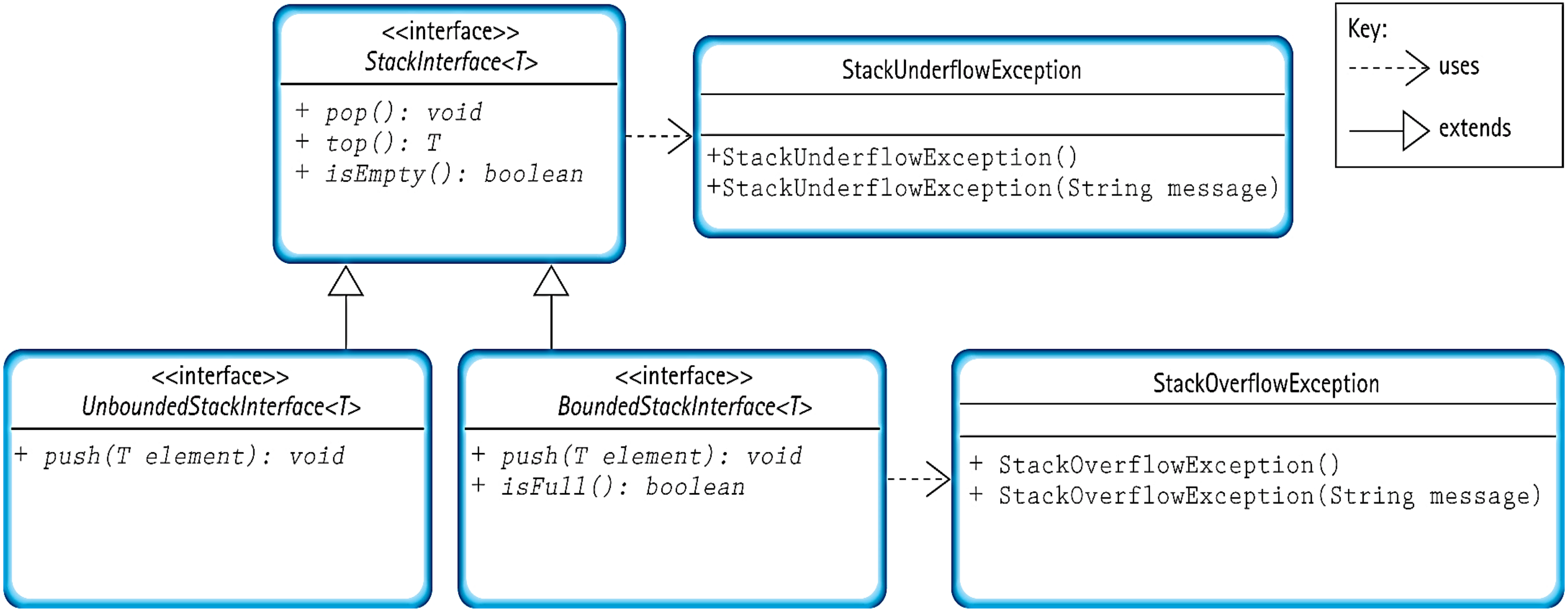
- Example of application:

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

The interfaces of the **Stack** ADT

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

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

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

The interfaces of the `Stack` ADT

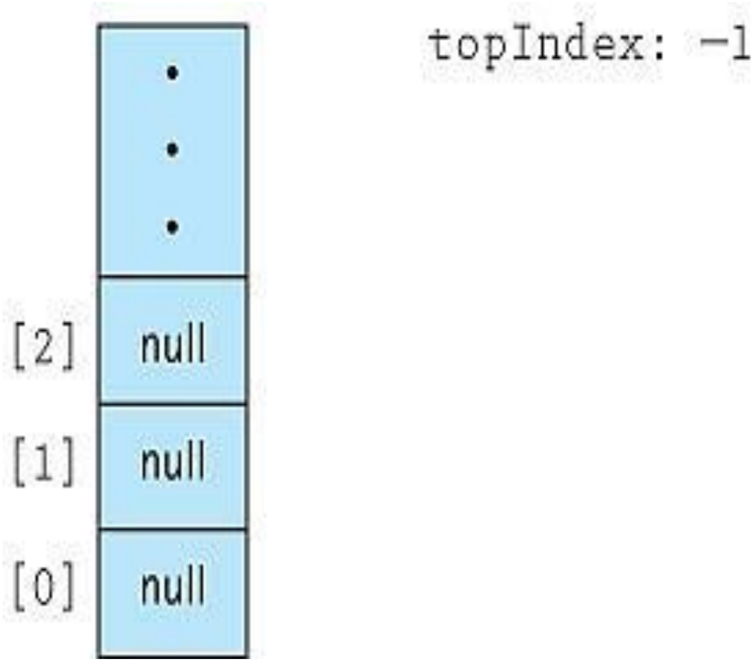


Array-based implementations

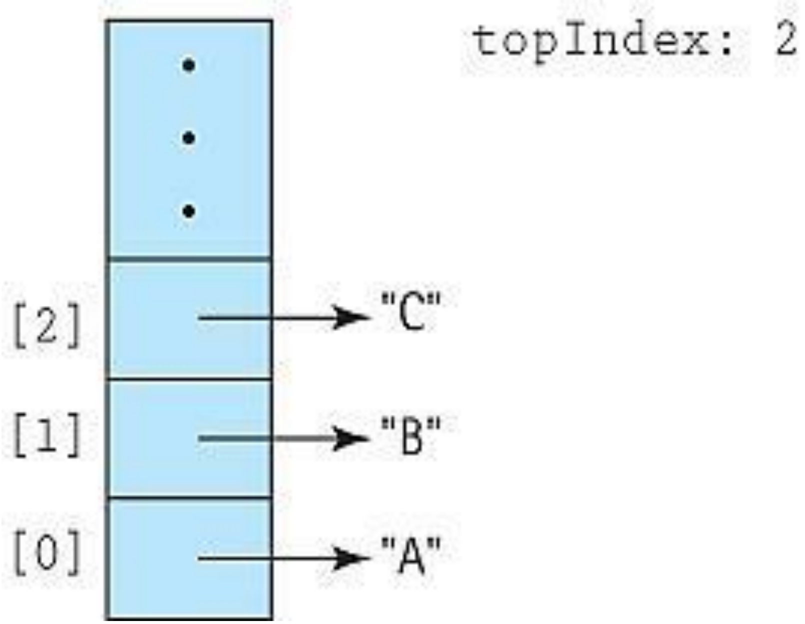
```
public class ArrayStack<T> implements BoundedStackInterface<T>
{
  protected final int defCap = 100; // default capacity
  protected T[] stack;              // holds stack elements
  protected int topIndex = -1; // index of the top element

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

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

Visualizing the stack

The empty stack:



After pushing "A", "B" and "C":



Array-based implementations

```
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;
}
```

Array-based implementations

```
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.");
}
```

Array-based implementations

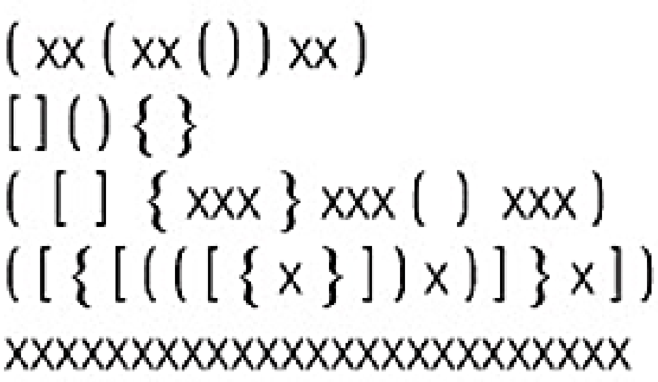
```
public T top()
// Throws StackUnderflowException if this stack is empty,
// otherwise returns top element from this stack.
{
  T topOfStack = null;
  if (!isEmpty())
    topOfStack = stack[topIndex];
  else
    throw new StackUnderflowException("Top attempted on an empty
                        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

Application: Well-formed expressions

- Given a set of grouping symbols, determine if the open and close versions of each symbol are matched correctly.
  - We'll focus on the normal pairs, `()`, `[]`, and `{}`, but in theory we could define any pair of symbols (e.g., `< >` or `/ \`) as grouping symbols.
  - Any number of other characters may appear in the input expression, before, between, or after a grouping pair, and an expression may contain nested groupings.
  - Each close symbol must match the last unmatched opening symbol and each open grouping symbol must have a matching close symbol.

Application: Well-formed expressions

Well-formed expressions



(xx (xx ( )) xx)

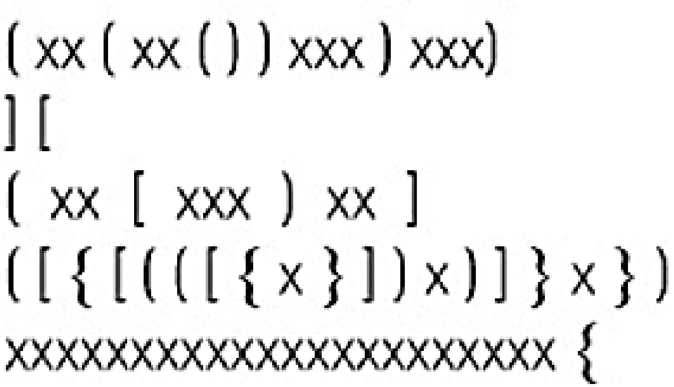
[ ] ( ) { }

([ ] { xxx } xxx ( ) xxx)

([{ [(([{ x }]) x)] } x])

xxxxxxxxxxxxxxxxxxxxxxxxx

Ill-formed expressions



(xx (xx ( )) xxx) xxx)

] [

(xx [xxx) xx]

([{ [(([{ x }]) x)] } x })

xxxxxxxxxxxxxxxxxxxxxx {

The **Balanced** class

- To help solve our problem we create a class called **Balanced**, with two instance variables of type **String** (**openSet** and **closeSet**) and a single exported method **test**.
- The constructor is:

```
public Balanced(String openSet, String closeSet)
// Preconditions: No character is contained more than once
// in the combined openSet and closeSet strings.
// The size of openSet = the size of closeSet.
{
  this.openSet = openSet;
  this.closeSet = closeSet;
}
```

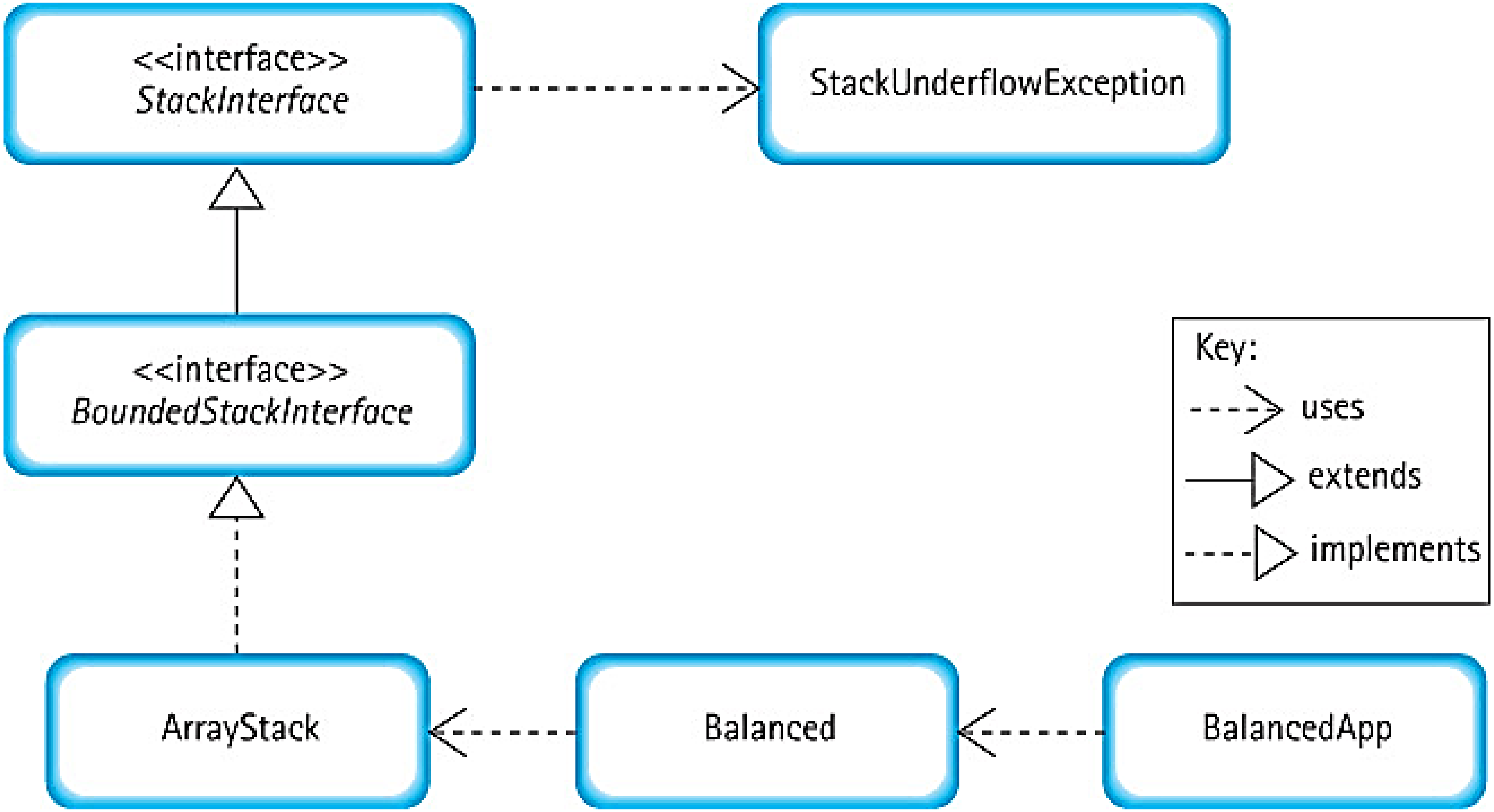
The `test` method

- Takes an expression as a string argument and checks to see if the grouping symbols in the expression are balanced.
- We use an integer to indicate the result:
    - 0 means the symbols are balanced, such as `(([xx])xx)`
    - 1 means the expression has unbalanced symbols, such as `(([xx}xx))`
    - 2 means the expression came to an end prematurely, such as `(([xxx])xx`

The `test` method

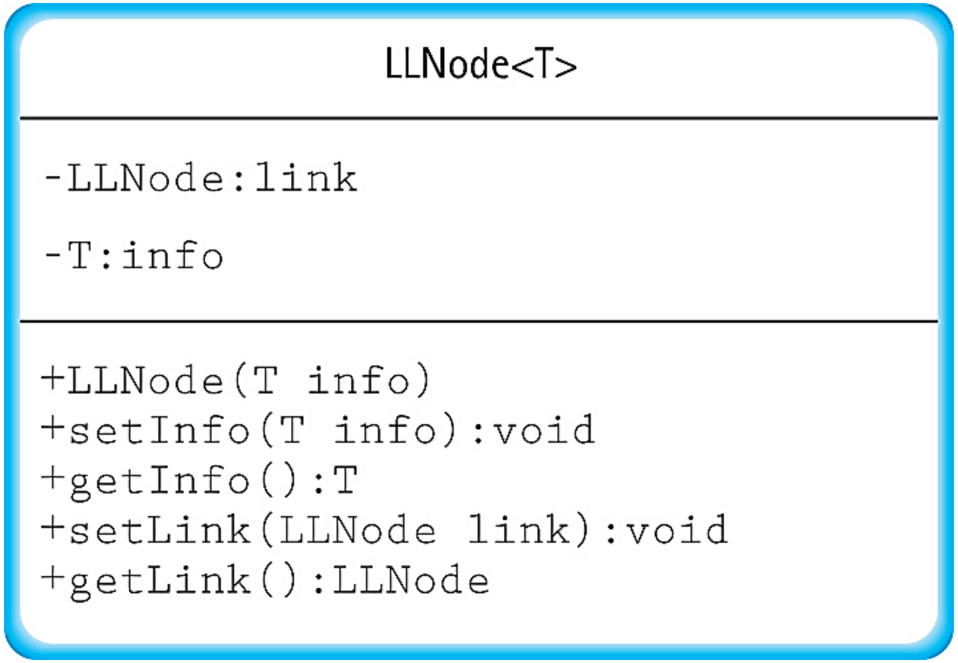
- For each input character, it does one of three tasks:
    - If the character is an open symbol, it is pushed on a stack.
    - If the character is a close symbol, it must be checked against the last open symbol, which is obtained from the top of the stack.
        - If they match, processing continues with the next character.
        - If the close symbol does not match the top of the stack, or if the stack is empty, then the expression is ill-formed.
    - If the character is not a special symbol, it is skipped.
- **See examples: Balanced.java, and BalancedApp.java**

Program Architecture



Linked-based implementations

- Like we did in the linked list **`StringLog`** implementation, we need to define a class similar to the **`LLStringNode`** class, called **`LLNode`** to act as the nodes of the list.



LLNode<T>

```
-LLNode:link
-T:info
```

```
+LLNode(T info)
+setInfo(T info):void
+getInfo():T
+setLink(LLNode link):void
+getLink():LLNode
```

The **LLNode** class

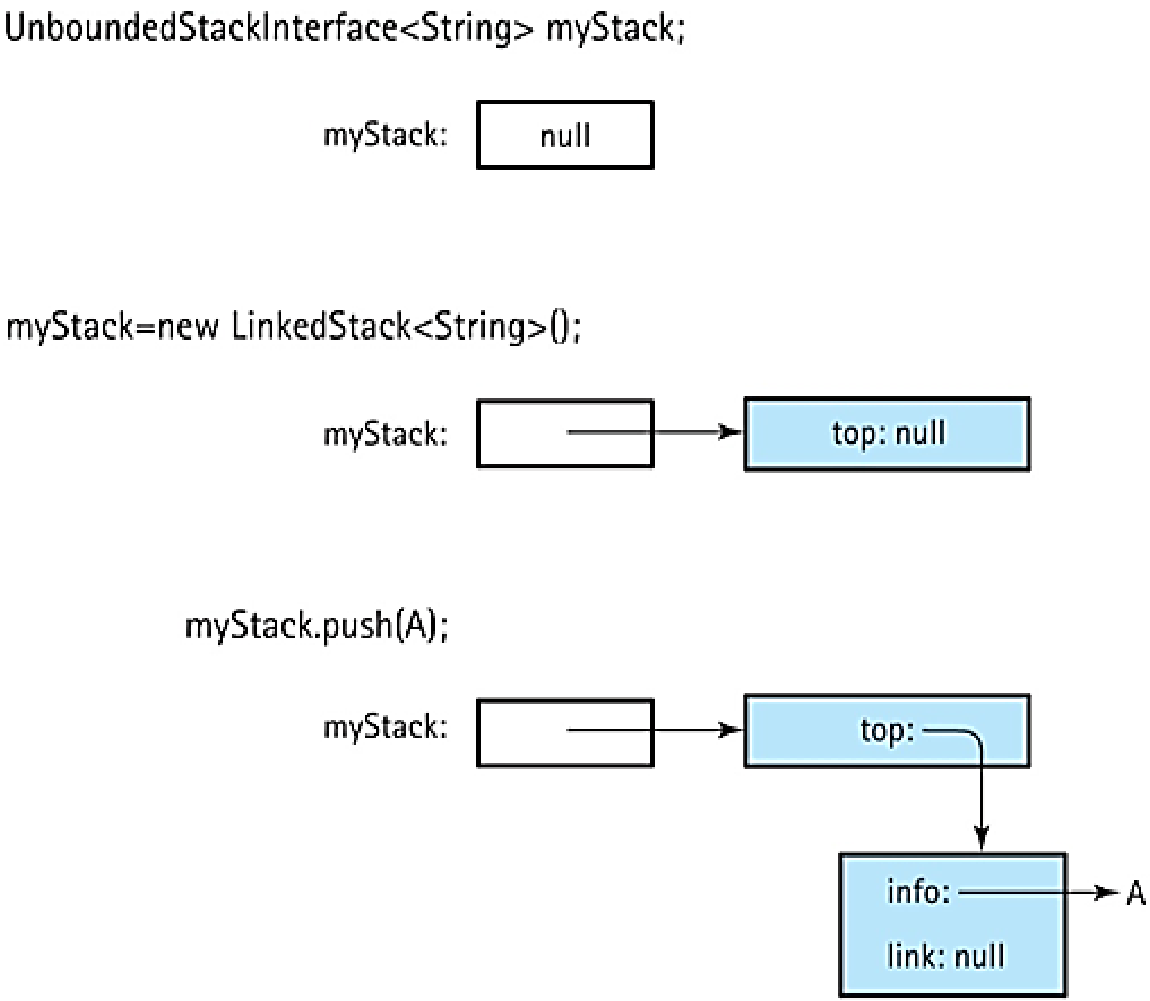
```
package support;

public class LLNode<T>
{
  private LLNode link;
  private T info;

  public LLNode(T info)
  {
    this.info = info;
    link = null;
  }

  public void setInfo(T info)
  {
    this.info = info;
  }
```

```
  public T getInfo()
  {
    return info;
  }


  public void setLink(LLNode
                      link)
  {
    this.link = link;
  }

  public LLNode<T> getLink()
  {
    return link;
  }
}
```

The **LinkedStack** class

```
package ch03.stacks;

import support.LLNode;

public class LinkedStack<T> implements
UnboundedStackInterface<T>
{
  protected LLNode top; // reference to the top of this stack
  public LinkedStack()
  {
    top = null;
  }
. . .
```

The **push** operation

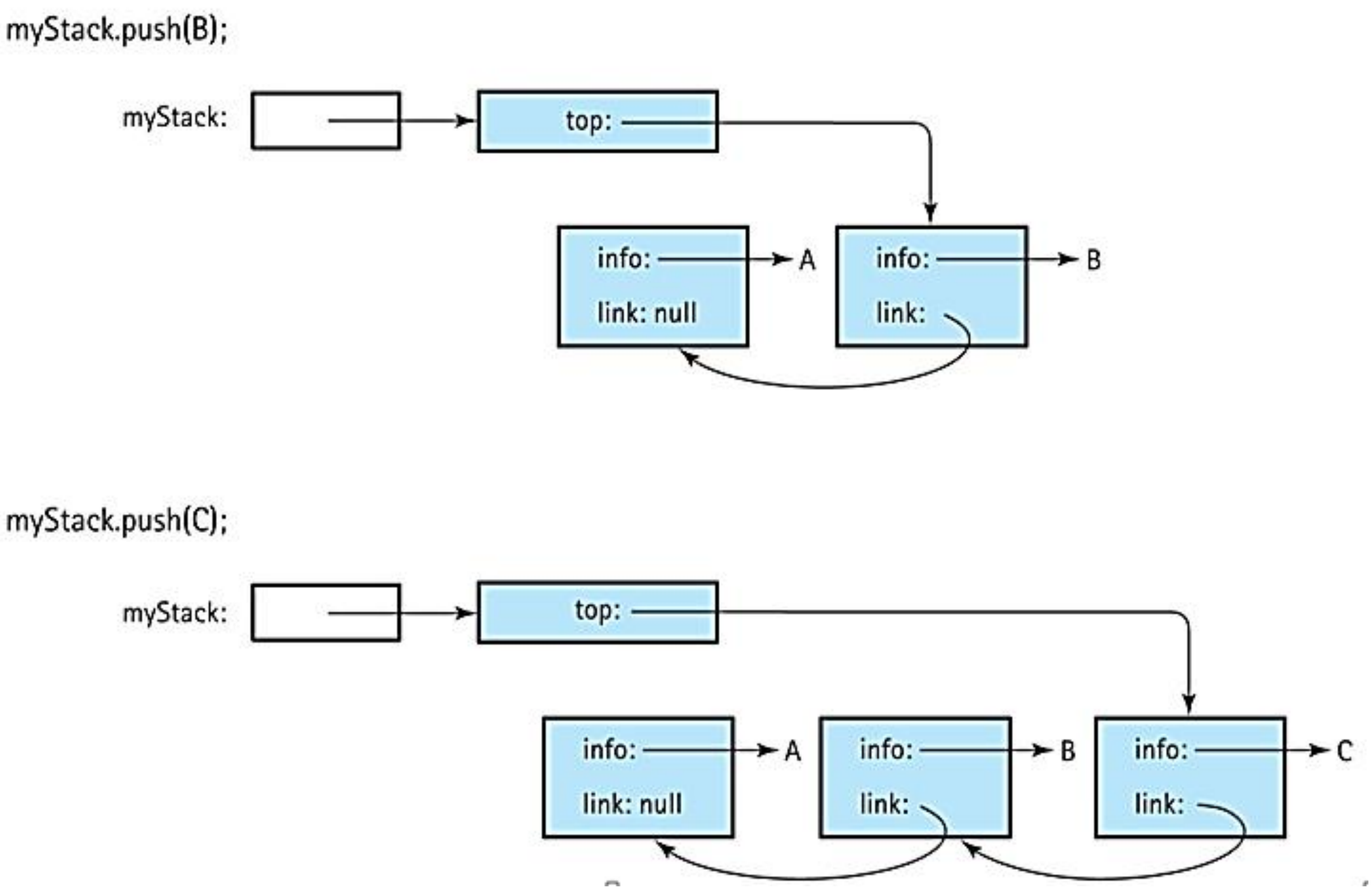


```
UnboundedStackInterface<String> myStack;
```

```
myStack=new LinkedStack<String>();
```

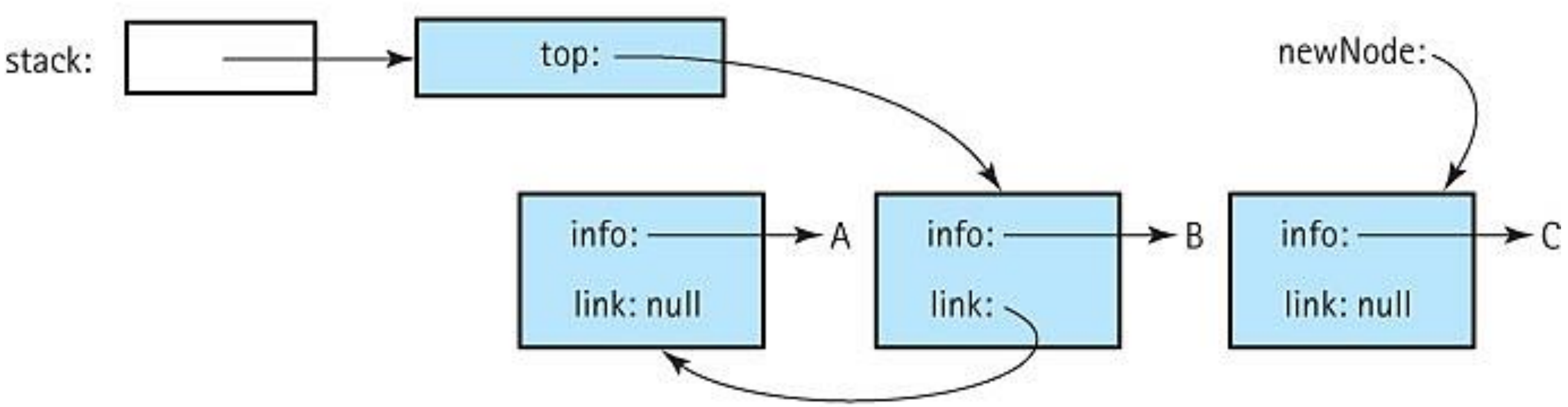
```
myStack.push(A);
```

The push operation



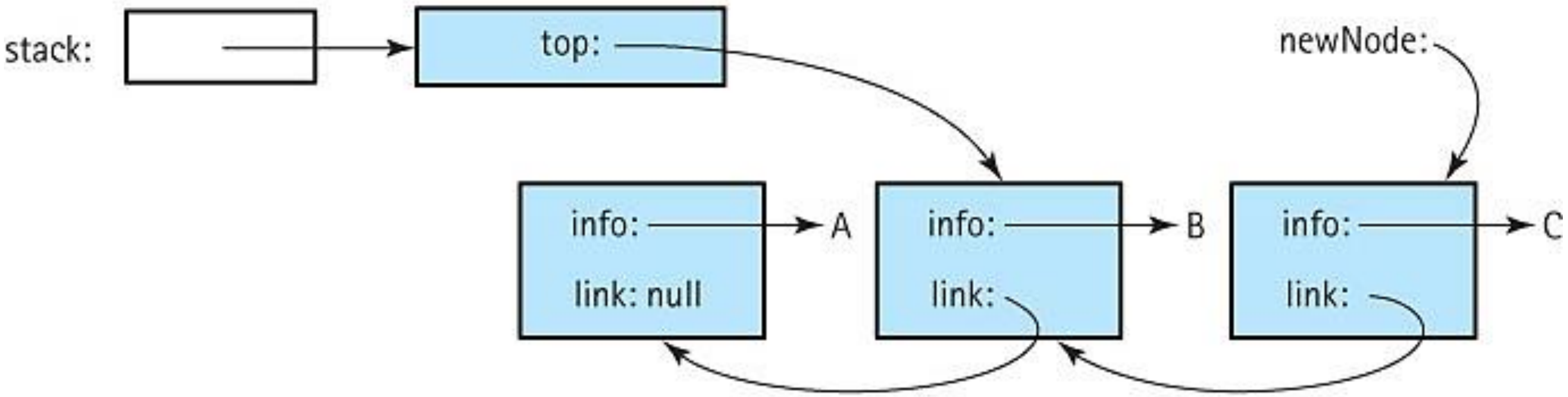
The **push** operation

- What happens when **push(C)** is called?
  - Allocate space for the next stack node and set the node info to element
  - Set the node link to the previous top of stack
  - Set the top of stack to the new stack node



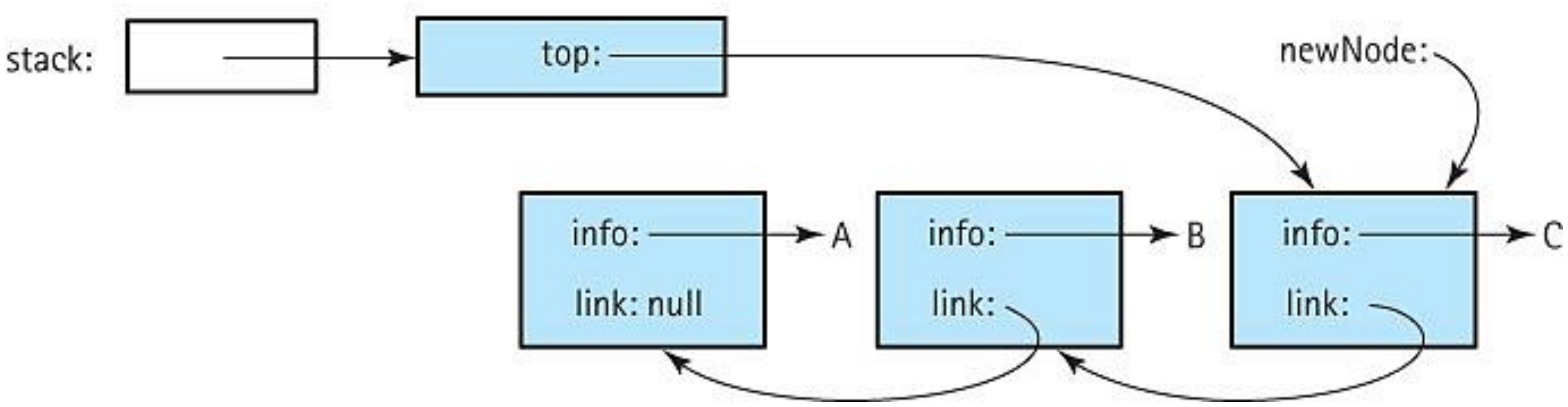
The **push** operation

- What happens when **push(C)** is called?
    - Allocate space for the next stack node and set the node info to element
    - Set the node link to the previous top of stack
    - Set the top of stack to the new stack node



The **push** operation

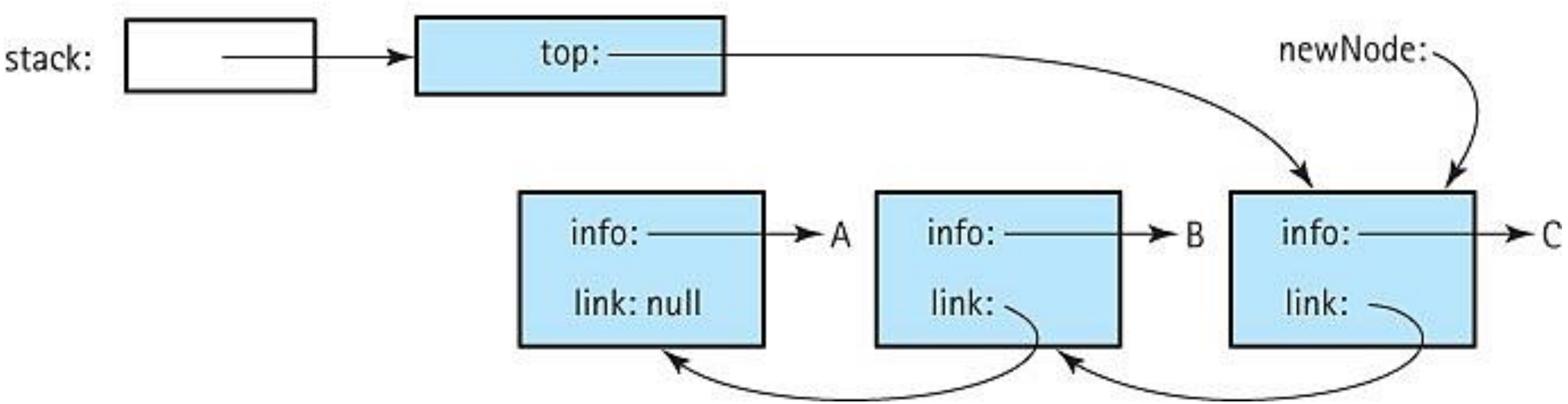
- What happens when **push(C)** is called?
  - Allocate space for the next stack node and set the node info to element
  - Set the node link to the previous top of stack
  - Set the top of stack to the new stack node



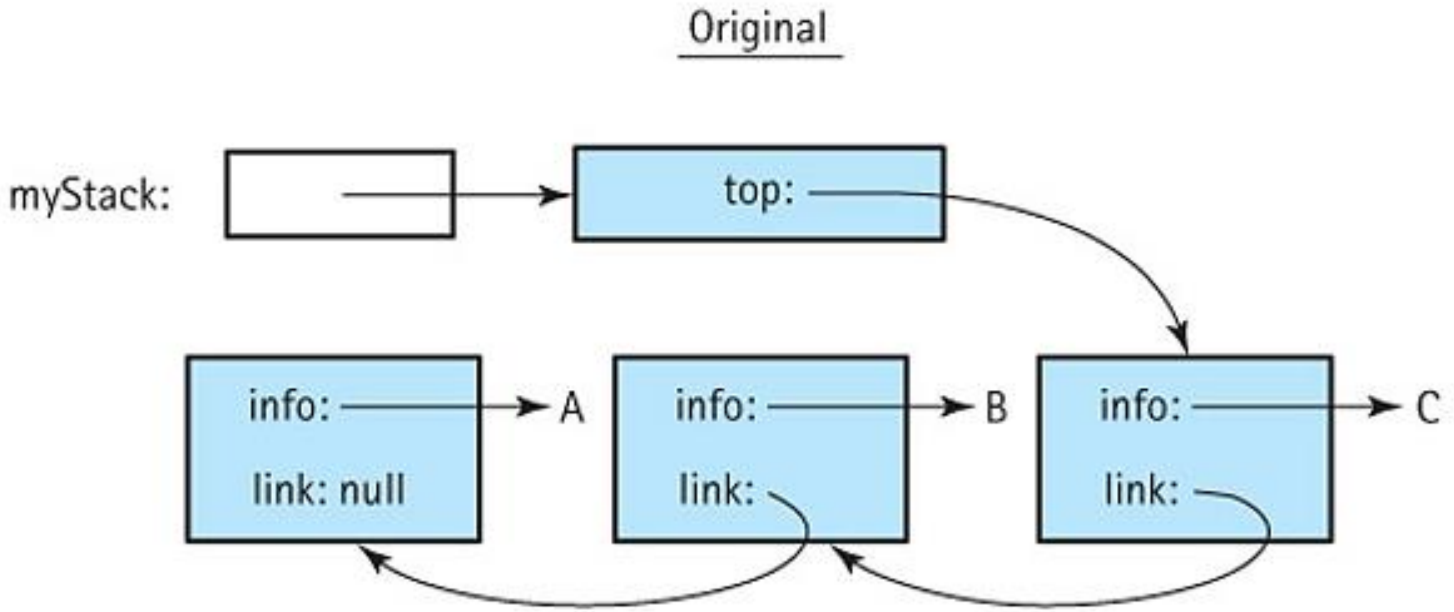
The **push** operation

- Code for the **push** method:

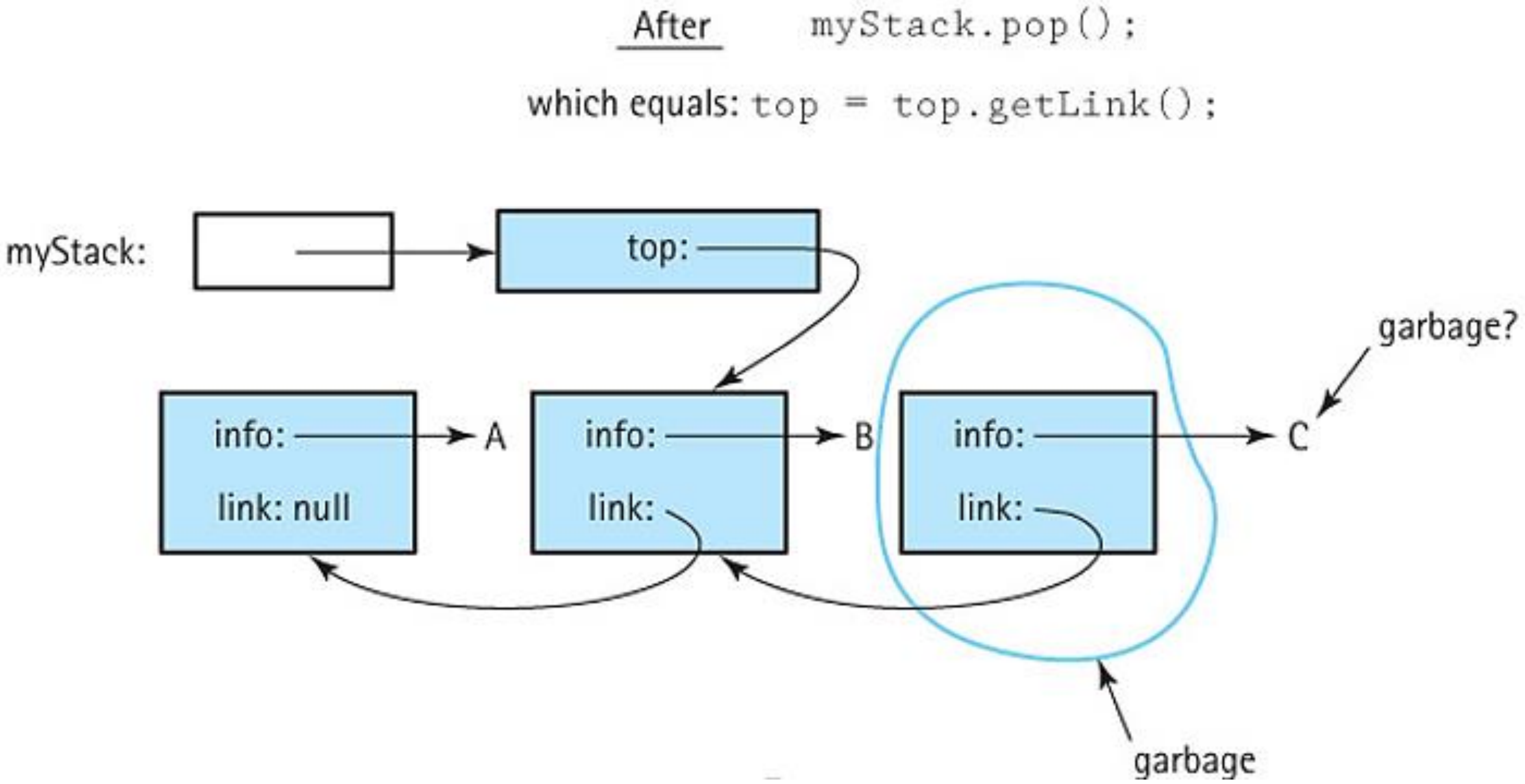
```
public void push(T element)
// Places element at the top of this stack.
{
  LLNode<T> newNode = new LLNode<T>(element);
  newNode.setLink(top);
  top = newNode;
}
```



The pop operation



The pop operation



The pop operation

- Code for the pop method:

```
public void pop()
// Throws StackUnderflowException if this stack is empty,
// otherwise removes top element from this stack.
{
  if (!isEmpty())
  {
    top = top.getLink();
  }
  else
    throw new StackUnderflowException("Pop attempted on an
                                               empty stack.");
}
```

The `top` and `isEmpty` operations

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

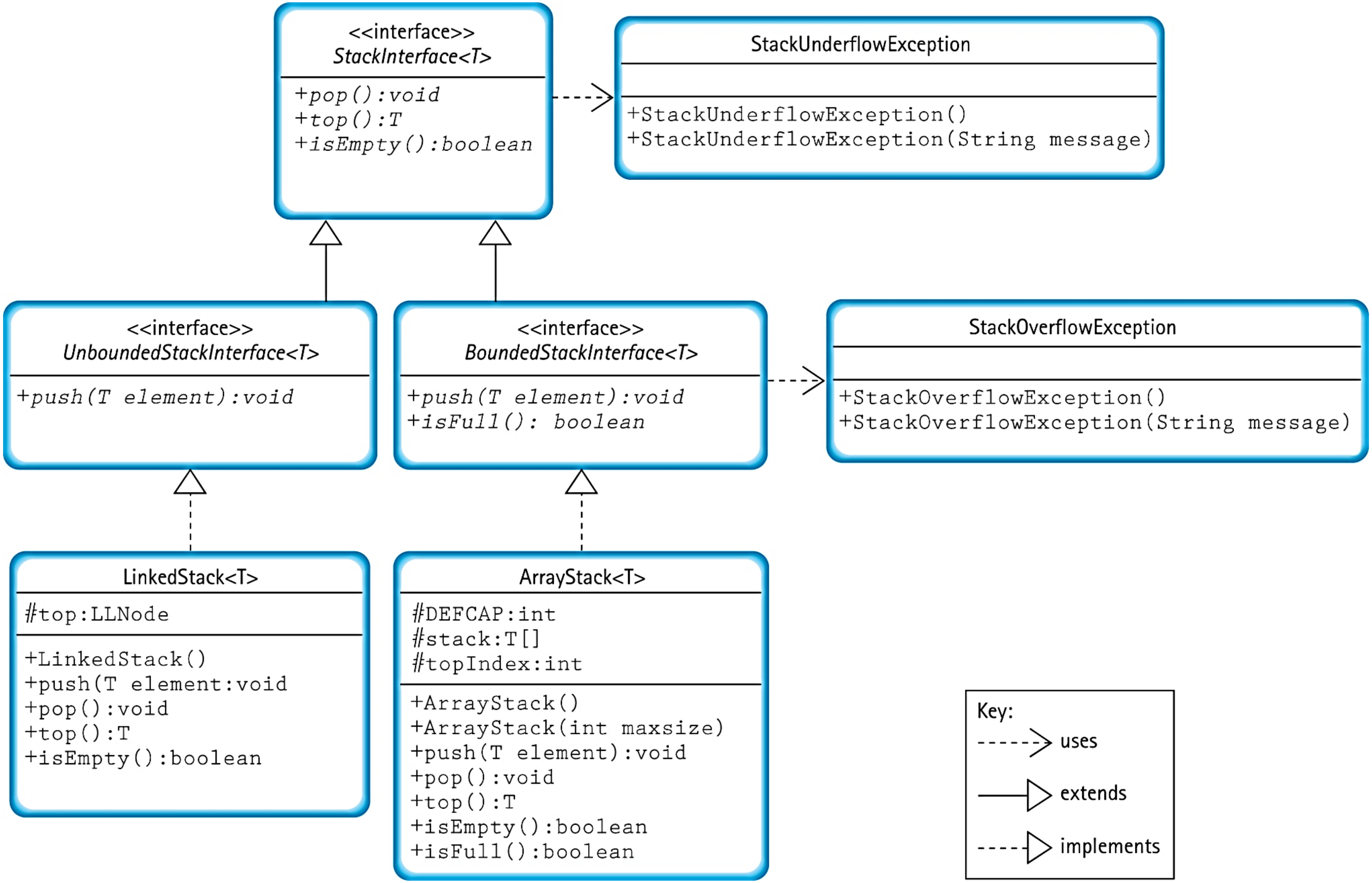
Comparing `Stack` implementations

- Storage Size
  - Array-based: takes the same amount of memory, no matter how many array slots are actually used, proportional to current capacity.
  - Link-based: takes space proportional to actual size of the queue (but each element requires more space than with array approach).
- Operation efficiency
  - All operations, for each approach, are O(1).
  - Except for the constructors:
    - Array-based: O(N)
    - Link-based: O(1)

Comparing `Stack` implementations

- So which is better?
  - The array-based implementation is short, simple, and efficient. Its operations have less overhead. When the maximum size is small and we know the maximum size with certainty, the array-based implementation is a good choice.
  - The linked implementation does not have space limitations, and in applications where the number of stack elements can vary greatly, it wastes less space when the stack is small.

Comparing `Stack` implementations



<<interface>>
StackInterface<T>

+pop():void
+top():T
+isEmpty():boolean

StackUnderflowException

+StackUnderflowException()
+StackUnderflowException(String message)

<<interface>>
UnboundedStackInterface<T>

+push(T element):void

<<interface>>
BoundedStackInterface<T>

+push(T element):void
+isFull(): boolean

StackOverflowException

+StackOverflowException()
+StackOverflowException(String message)

LinkedStack<T>

#top:LLNode

+LinkedStack()
+push(T element:void
+pop():void
+top():T
+isEmpty():boolean

ArrayStack<T>

#DEFCAP:int
#stack:T[]
#topIndex:int

+ArrayStack()
+ArrayStack(int maxsize)
+push(T element):void
+pop():void
+top():T
+isEmpty():boolean
+isFull():boolean

Key:
- - - - -> uses
——▷ extends
- - - -▷ implements

Postfix expressions

- Postfix notation is a notation for writing arithmetic expressions in which the operators appear after their operands.
  - $(2 + 14) \times 23 \;\;\rightarrow\;\; 2\;\;14 + 23 \times$
  - $9 \times 7 + 16 \div (5 - 1) \times 3 \;\;\rightarrow\;\; 9\;\;7 \times 16\;\;5\;\;1 - 3 \times \div +$
- Evaluating postfix expressions
  - Scan from the left to the right, stop at the first unprocessed operator, e.g., $+$ in expression $2\;\;14 + 23 \times$.
  - Take the two operands that are directly before the operator, e.g., 2 and 14 in expression $2\;\;14 + 23 \times$.
  - Calculate the expression, e.g., $2 + 14 = 16$.
  - Replace the just-calculated-expression with the result, e.g., $2\;\;14 + 23 \times \;\;\rightarrow\;\; 16\;\;23 \times$.
  - Repeat the process until nothing is left over in the expression.

Postfix expressions

Postfix expression evaluation algorithm using stack

```
while more item exists
  Get an item
  if item is an operand
    stack.push(item)
  else
    operand2 = stack.top()
    stack.pop()
    operand1 = stack.top()
    stack.pop()
    Set result to (apply operation corresponding to item to
                             operand1 and operand2)
    stack.push(result)
result = stack.top()
stack.pop()
return result
```

If the `pop` method returns the element at the top of the stack, then there is no need to use a separate `top` method.

Action items

- Read book chapter 3.