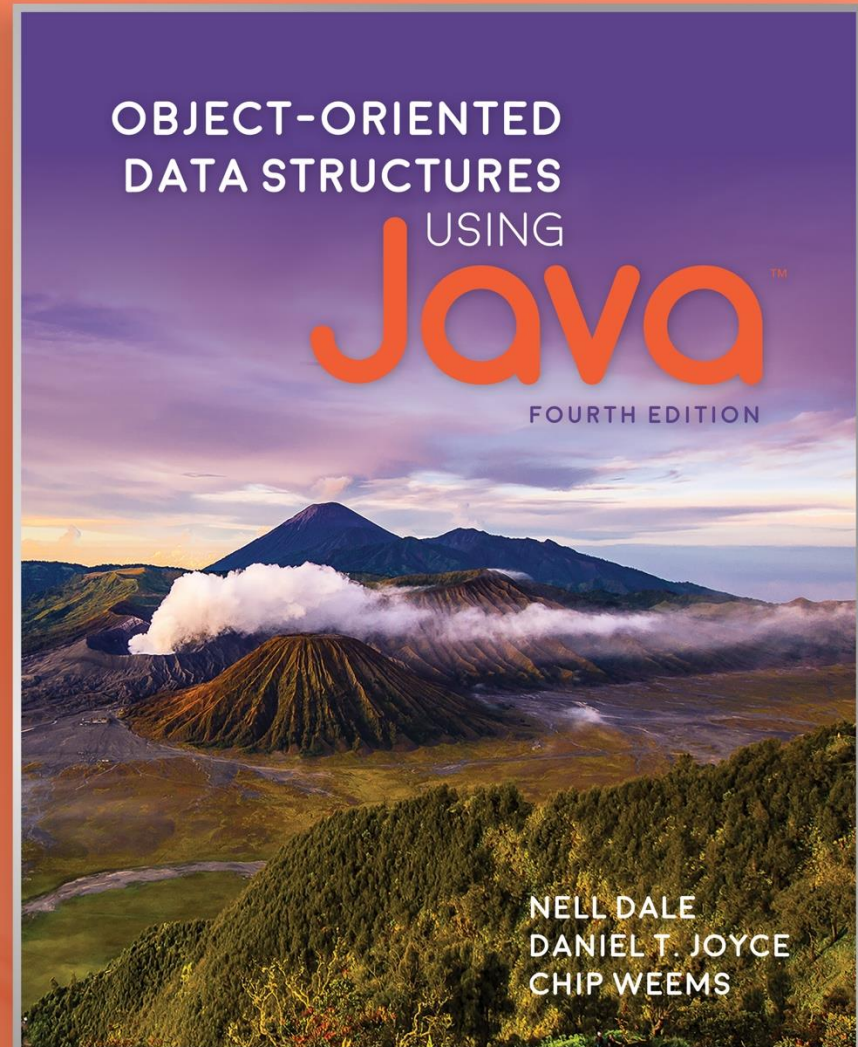# Chapter 2

The Stack ADT

Chapter 2: The Stack ADT

2.1 – Abstraction

2.2 – The Stack

2.3 – Collection Elements

2.4 – The Stack Interface

2.5 – Array-Based Stack Implementations

2.6 – Application: Balanced Expressions

2.7 – Introduction to Linked Lists

2.8 – A Link-Based Stack

2.9 – Application: Postfix Expression Evaluator

2.10 – Stack Variations

2.1 Abstraction

- **Abstraction** A model of a system that includes only the details essential to the perspective of the viewer of the system
- **Information hiding** The practice of hiding details within a module with the goal of controlling access to the details from the rest of the system
- **Data abstraction** The separation of a data type's logical properties from its implementation
- **Abstract data type (ADT)** A data type whose properties (domain and operations) are specified independently of any particular implementation

ADT Perspectives or Levels

- *Application (or user or client or external) level*: We use the ADT to solve a problem. When working at this level we only need to know how to create instances of the ADT and invoke its operations.

- *Abstract (or logical) level*: Provides an abstract view of the data values (the domain) and the set of operations to manipulate them. At this level, we deal with the "*what*" questions. What is the ADT? What does it model? What are its responsibilities? What is its interface?

- *Implementation (or concrete or internal) level*: Provides a specific representation of the structure to hold the data and the implementation of the operations. Here we deal with the "*how*" questions.

Preconditions and Postconditions

- **Preconditions** Assumptions that must be true on entry into a method for it to work correctly
- **Postconditions or Effects** The results expected at the exit of a method, assuming that the preconditions are true
- We specify pre- and postconditions for a method in a comment at the beginning of the method

Java: Abstract Method

- Only includes a description of its parameters
- No method bodies or implementations are allowed.
- In other words, only the *interface* of the method is included.

Java Interfaces

- Similar to a Java class
  - can include variable declarations
  - can include methods
- However
  - Variables must be constants
  - Methods must be abstract.
  - A Java interface cannot be instantiated.
- We can use an interface to formally specify the logical level of an ADT:
  - It provides a template for classes to fill.
  - A separate class then "implements" it.
- For example, see the `FigureInterface` interface and the `Circle` class (next slide) and `Rectangle` class (following slide) that implement it

FigureInterface

```
package ch02.figures;

public interface FigureInterface
{
    final double PI = 3.14;

    double perimeter();
    // Returns perimeter of this figure.

    double area();
    // Returns area of this figure.
}
```

Circle Class

```
package ch02.figures;

public class Circle implements
                    FigureInterface
{
  protected double radius;

  public Circle(double radius)
  {
    this.radius = radius;
  }

  public double perimeter()
  {
    return(2 * PI * radius);
  }

  public double area()
  {
    return(PI * radius * radius);
  }
}
```

FigureInterface

```
package ch02.figures;

public interface FigureInterface
{
    final double PI = 3.14;

    double perimeter();
    // Returns perimeter of this figure.

    double area();
    // Returns area of this figure.
}
```

Rectangle Class

```
package ch02.figures;

public class Rectangle implements
                        FigureInterface
{
  protected double length, width;

  public Rectangle(double length,
                double width)
  {
    this.length = length;
    this.width = width;
  }

  public double perimeter()
  {
    return(2 * (length + width));
  }

  public double area()
  {
    return(length * width);
  }
```
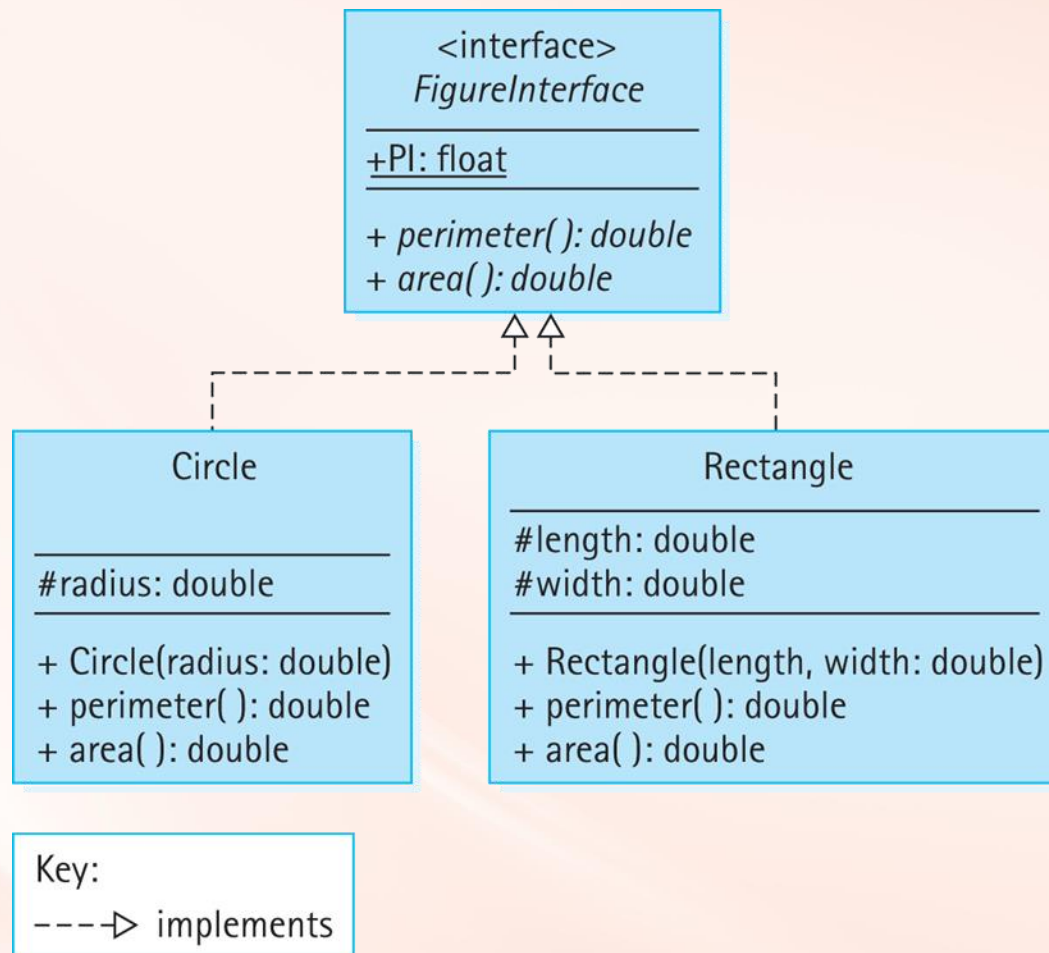
UML Class Diagram

<interface>
FigureInterface
___
+PI: float
___
+ *perimeter( ): double*
+ *area( ): double*

Circle
___
#radius: double
___
+ Circle(radius: double)
+ perimeter( ): double
+ area( ): double

Rectangle
___
#length: double
#width: double
___
+ Rectangle(length, width: double)
+ perimeter( ): double
+ area( ): double

Key:
---▷ implements

Benefits from Using Interfaces

- We can formally check the syntax of our specification. When we compile the interface, the compiler uncovers any syntactical errors in the method interface definitions.

- We can formally verify that the interface "contract" is met by the implementation. When we compile the implementation, the compiler ensures that the method names, parameters, and return types match what was defined in the interface.

- We can provide a consistent interface to applications from among alternate implementations of the ADT.

Interface-Based Polymorphism

- Polymorphism – an object variable can reference objects of *different* classes at *different* times.

- A variable declared to be of an `interface` "type" `T` can hold a reference to any object of class that implements that interface.

- See example, next slide.

```
FigureInterface[] figures = new FigureInterface[COUNT];
final int COUNT = 5;

// generate figures
for (int i = 0; i < COUNT; i++)
{
  switch (rand.nextInt(2))
  {
    case 0: figures[i] = new Circle(1.0);
    break;

    case 1: figures[i] = new Rectangle(1.0, 2.0);
    break;
  }
}
```

What is held in `figures[3]`?

We cannot predict. The binding of `figures[3]` to a class (`Circle` or `Rectangle`) occurs dynamically, at run time.

`figures[3]` is a polymorphic object.

2.2 The Stack
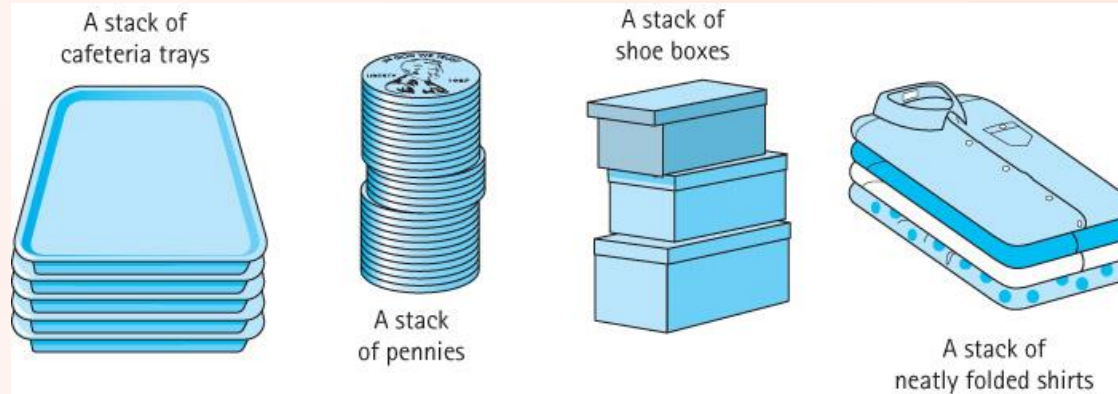
- **Stack** A structure in which elements are added and removed from only one end; a "last in, first out" (LIFO) structure

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
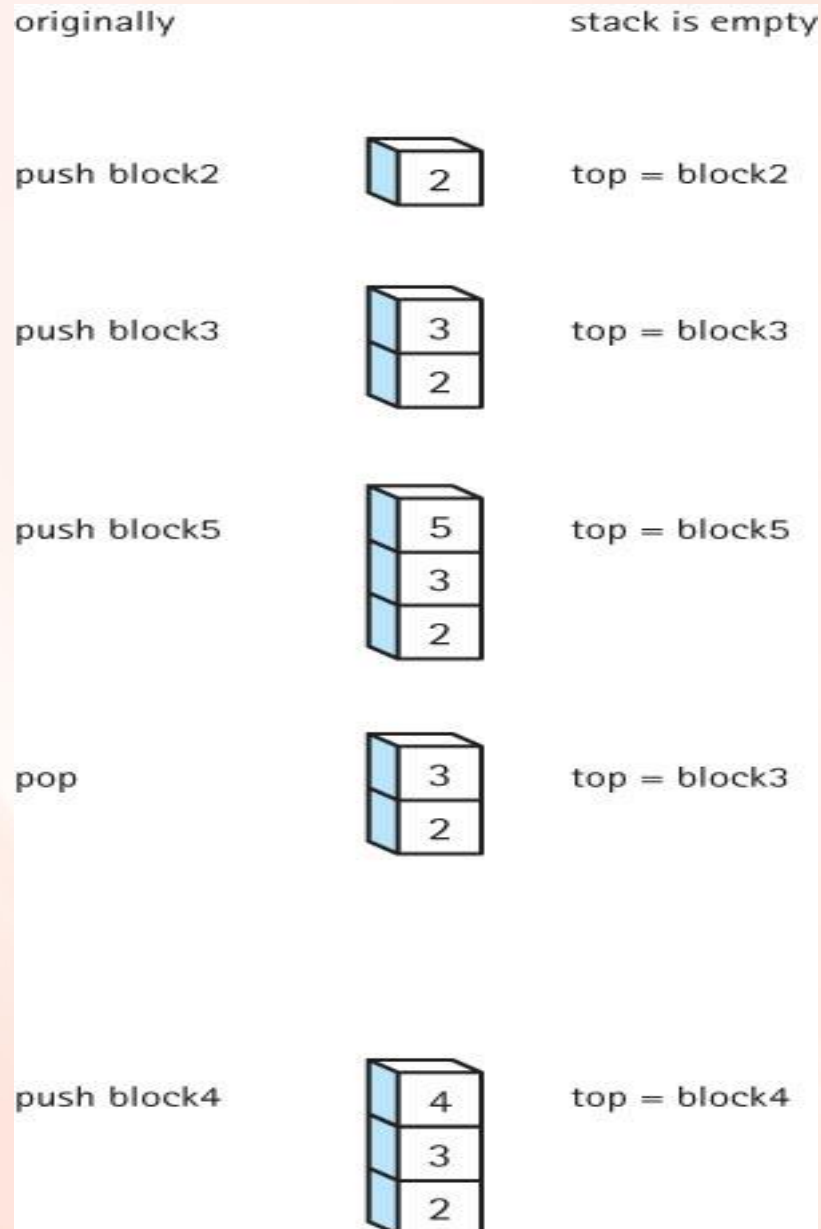
Using Stacks

- 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 process on a stack, so that it can work on a higher-priority, interrupting process
- Stacks are good for storing "postponed obligations"

2.3 Collection Elements

- **Collection** An object that holds other objects. Typically we are interested in inserting, removing, and obtaining the elements of a collection.

- A stack is an example of a Collection ADT. It collects together elements for future use, while maintaining a first in – last out ordering among the elements.

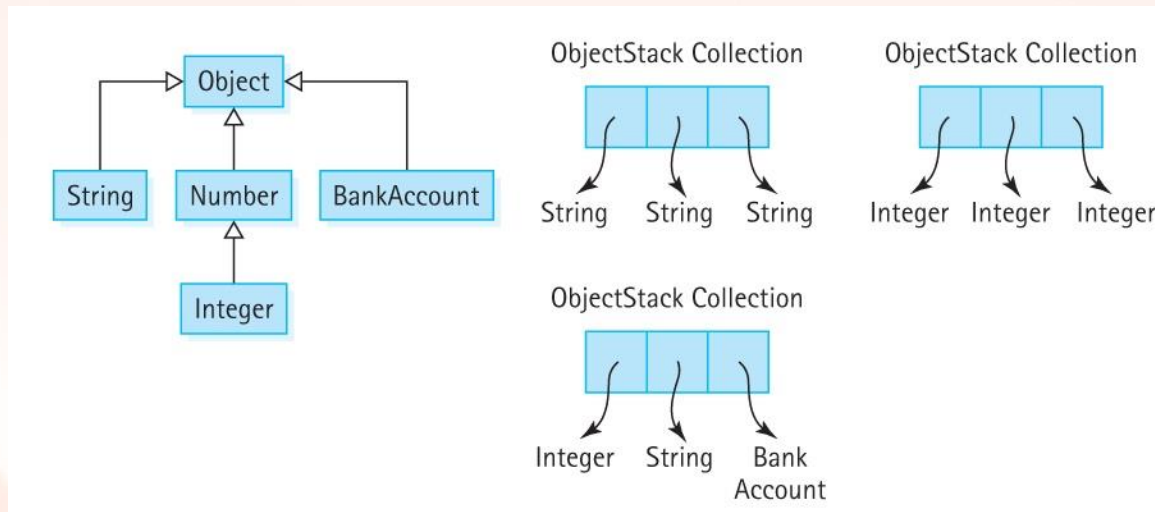# How can we implement stacks that hold different types of elements?

One approach is to implement a separate stack class for each element type:
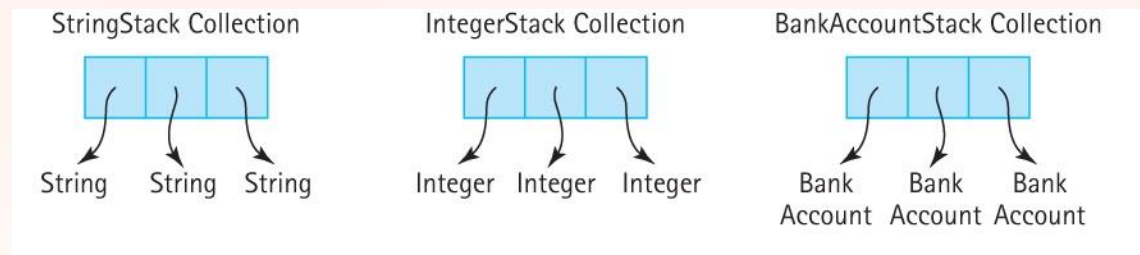
This approach is too redundant and not generally useful.

How can we implement stacks that hold different types of elements?

Another approach is to implement a stack class that can hold elements of class `Object`:

This approach requires extra effort to keep track of, and indicate, exactly what is in the stack.

Collections of Class `Object`

- Note: whenever an element is removed from the collection it can only be referenced as an `Object`. If you intend to use it as something else you must cast it into the type that you intend to 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
```

# How can we implement stacks that hold different types of elements?

Another approach is to implement a stack class that can hold elements of a particular interface:
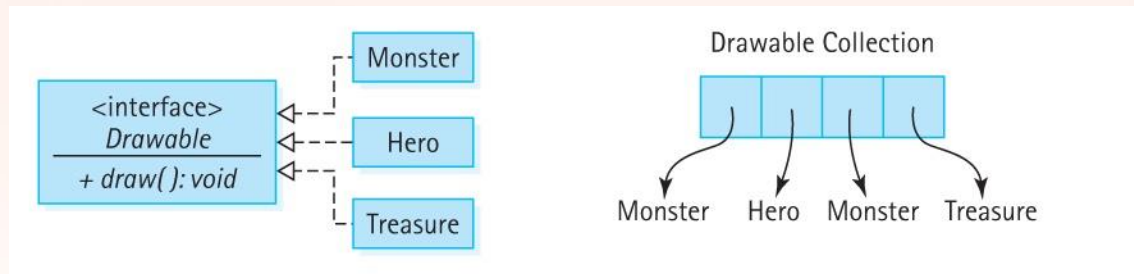
This approach could work in certain specific situations but it is not a general solution.

How can we implement stacks that hold different types of elements?

Generic stack – implement once and indicate the type of element to be held on instantiation
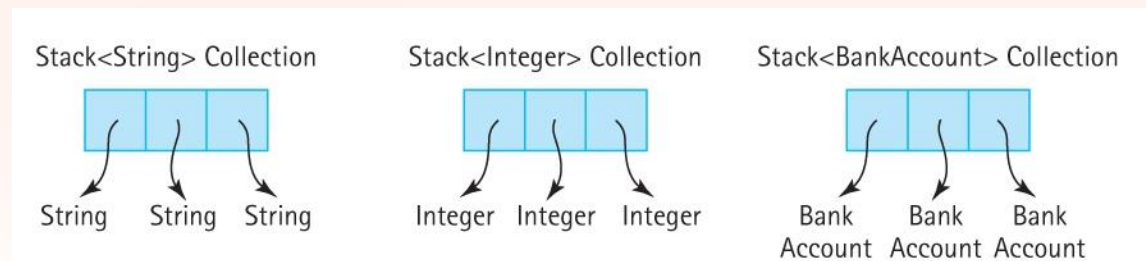
This is the approach we will use:

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

Generic Collections

- Example of beginning of Stack class definition:

```
public class Stack<T>
{
    protected T[ ] elements ;
    protected int topIndex = -1;
    ...
```

- Example of application declaring various stacks:

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

2.4 The Stack Interface

- Recall from Section 2.1 that a stack is a "last-in first-out" structure, with primary operations
  - 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
- In addition we need a constructor that creates 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.

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

StackInterface

```java
package ch02.stacks;

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

  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 of this stack.

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

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

Example Use of a Stack

- Instructors can now review and demonstrate the `ReverseStrings` application found in package `ch02.apps.`
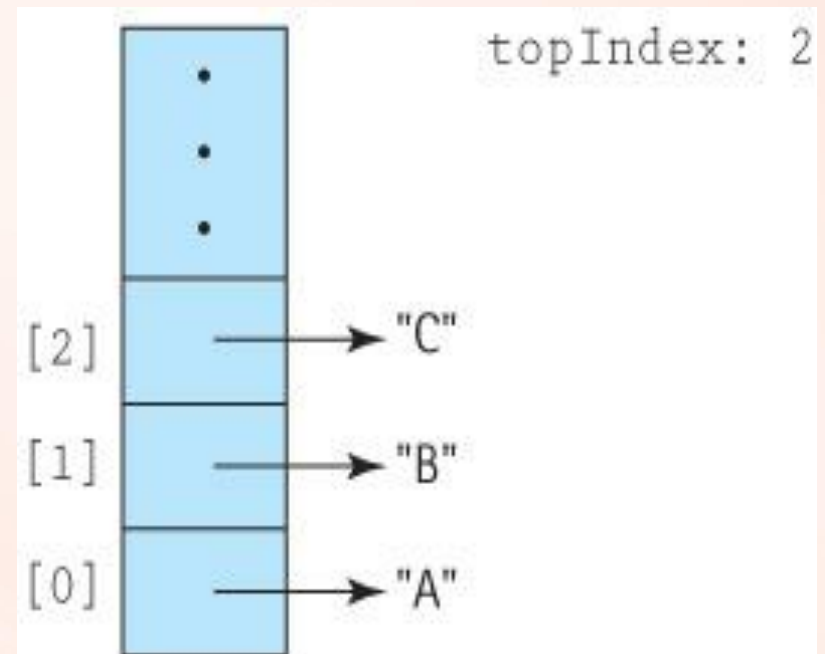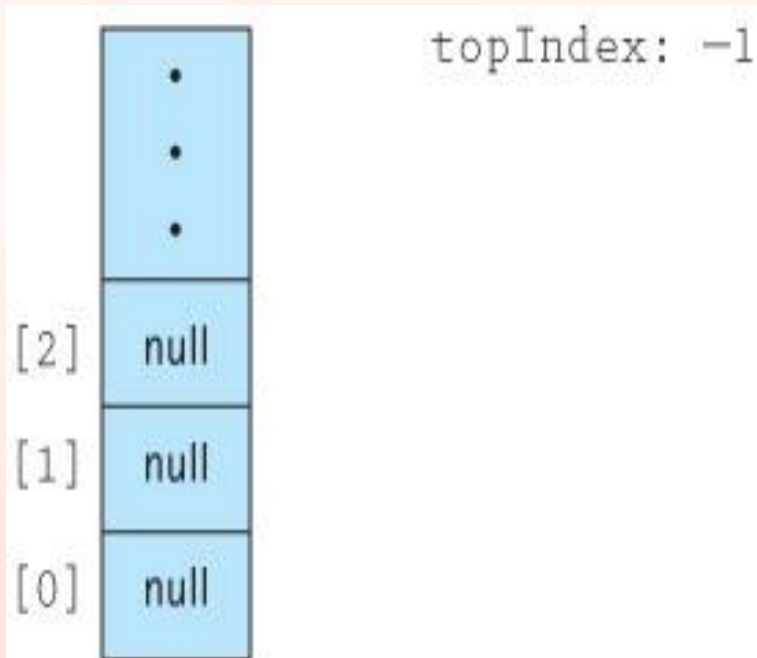
2.5 Array-Based Implementations

- In this section we study an array-based implementation of a bounded Stack ADT.

- Additionally, we look at an alternate unbounded implementation that uses the Java Library `ArrayList` class.

The beginning of the
`ArrayBoundedStack` Class

```java
//------------------------------------------------------------------
// ArrayBoundedStack.java      Dale/Joyce/Weems                  Chapter 2
//
// Implements StackInterface using an array to hold stack elements.
//
// Two constructors are provided: one that creates an array of a default
// size and one that allows the calling program to specify the size.
//------------------------------------------------------------------
package ch02.stacks;

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

  public ArrayBoundedStack()
  {
    elements = (T[]) new Object[DEFCAP];
  }

  public ArrayBoundedStack(int maxSize)
  {
    elements = (T[]) new Object[maxSize];
  }
```

Visualizing the stack

- The empty stack:

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

Definitions of Stack Operations

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


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

Definitions of Stack Operations

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

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

Definitions of Stack Operations

```java
public T top()
{
  T topOfStack = null;
  if (isEmpty())
    throw new StackUnderflowException("Top attempted on empty stack.");
  else
    topOfStack = elements[topIndex];
  return topOfStack;
}
```

Figure depicts Application, Implementation and Abstract views of the code sequence:

StackInterface<String> myStack;

myStack = new ArrayBoundedStack<String>(3);

myStack.push(A);

myStack.push(B);
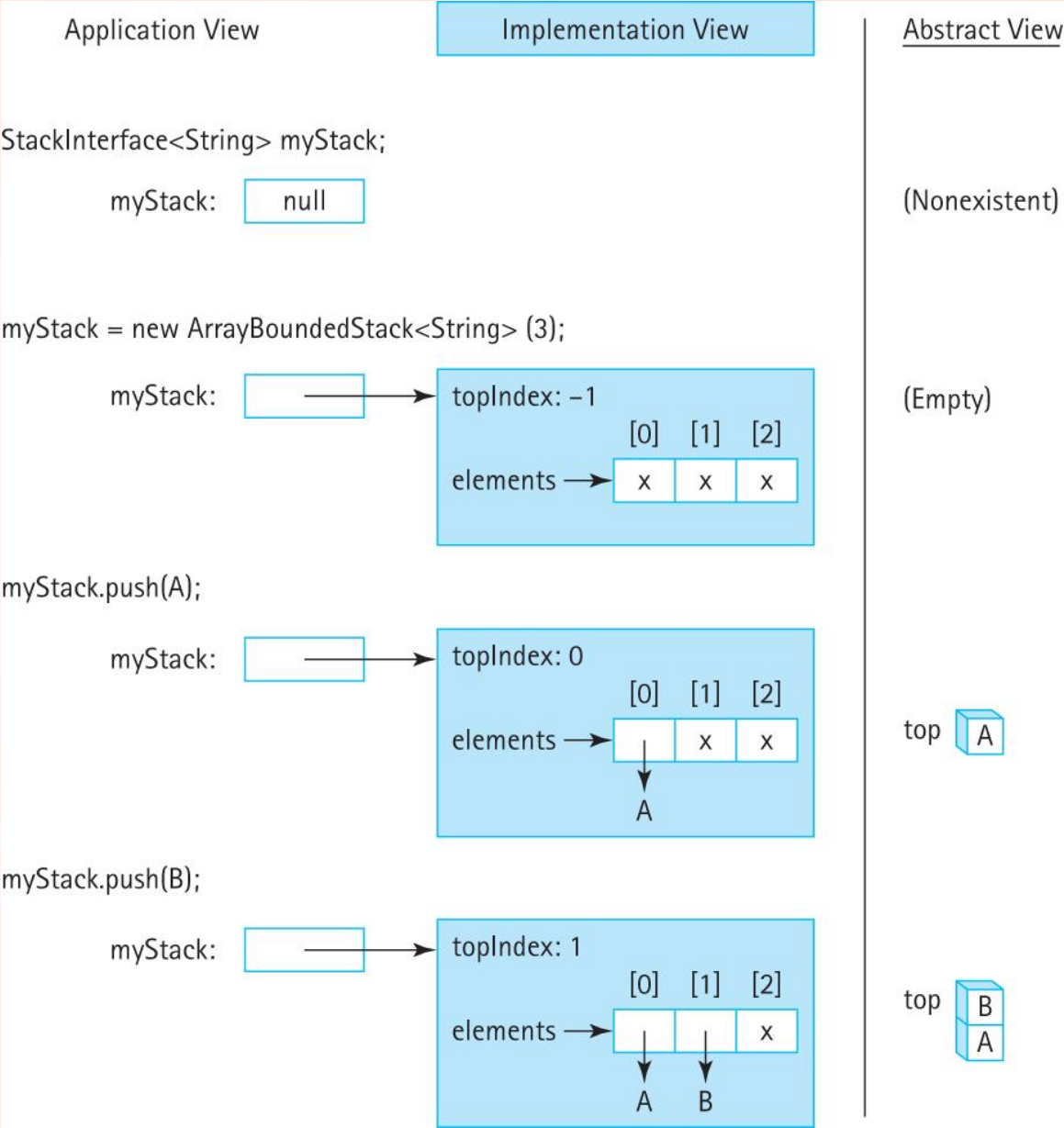
The beginning of the
`ArrayListStack` Class

```java
//---------------------------------------------------------------------
// ArrayListStack.java by Dale/Joyce/Weems Chapter 2
//
// Implements an unbounded stack using an ArrayList.
//---------------------------------------------------------------------
package ch02.stacks;

import java.util.*;

public class ArrayListStack<T> implements StackInterface<T>
{
  protected ArrayList<T> elements; // ArrayList that holds stack elements

  public ArrayListStack()
  {
    elements = new ArrayList<T>();
  }
```

Definitions of Stack Operations

```
public boolean isEmpty()
// Returns true if this stack is empty, otherwise returns false.
{
  return (elements.size() == 0);
}


public boolean isFull()
// Returns false - an ArrayListStack is never full.
{
  return false;
}
```

Definitions of Stack Operations

```java
public void push(T element)
{
  elements.add(element);
}

public void pop()
{
  if (isEmpty())
    throw new StackUnderflowException("Pop attempted on empty stack.");
  else
    elements.remove(elements.size() - 1);
}

public T top()
{
  T topOfStack = null;
  if (isEmpty())
    throw new StackUnderflowException("Top attempted on empty stack.");
  else
    topOfStack = elements.get(elements.size() - 1);
  return topOfStack;
}
```

2.6 Application: Balanced 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.

Examples

Well-Formed Expressions	Ill-Formed Expres
(xx (xx ( )) xx)	(xx (xx ( )) xxx) xxx)
[ ] ( ) { }	] [
([ ] { xxx } xxx ( ) xxx)	(xx [xxx) xx]
([{ [(([{ x }]) x)] } x])	([{ [(([{ x }]) x)] } x })
xxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxx {

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.

Test for Well-Formed Expression Algorithm (String expression)

```
Create a new stack of size equal to the length of subject

Set stillBalanced to true

Get the first character from expression

while (the expression is still balanced
        AND
        there are still more characters to process)
    Process the current character
    Get the next character from expression

if (!stillBalanced)
    return 1
else if (stack is not empty)
    return 2
else
    return 0
```

Expansion of
"Process the current character"

```
if (the character is an open symbol)
   Push the open symbol character onto the stack
else if (the character is a close symbol)
   if (the stack is empty)
     Set stillBalanced to false
   else
     Set open symbol character to the value at the top of the stack
     Pop the stack
     if the close symbol character does not "match" the open symbol character
       Set stillBalanced to false
else
   Skip the character
```

Code and Demo

- Instructors can now walk through the code contained in `Balanced.java` found in the `ch02.balanced` package and `BalancedCLI.java` and/or `BalancedGUI.java` found in the `ch02.apps` package, review the notes on pages 104 and 105, and demonstrate the running program.

Program Architecture
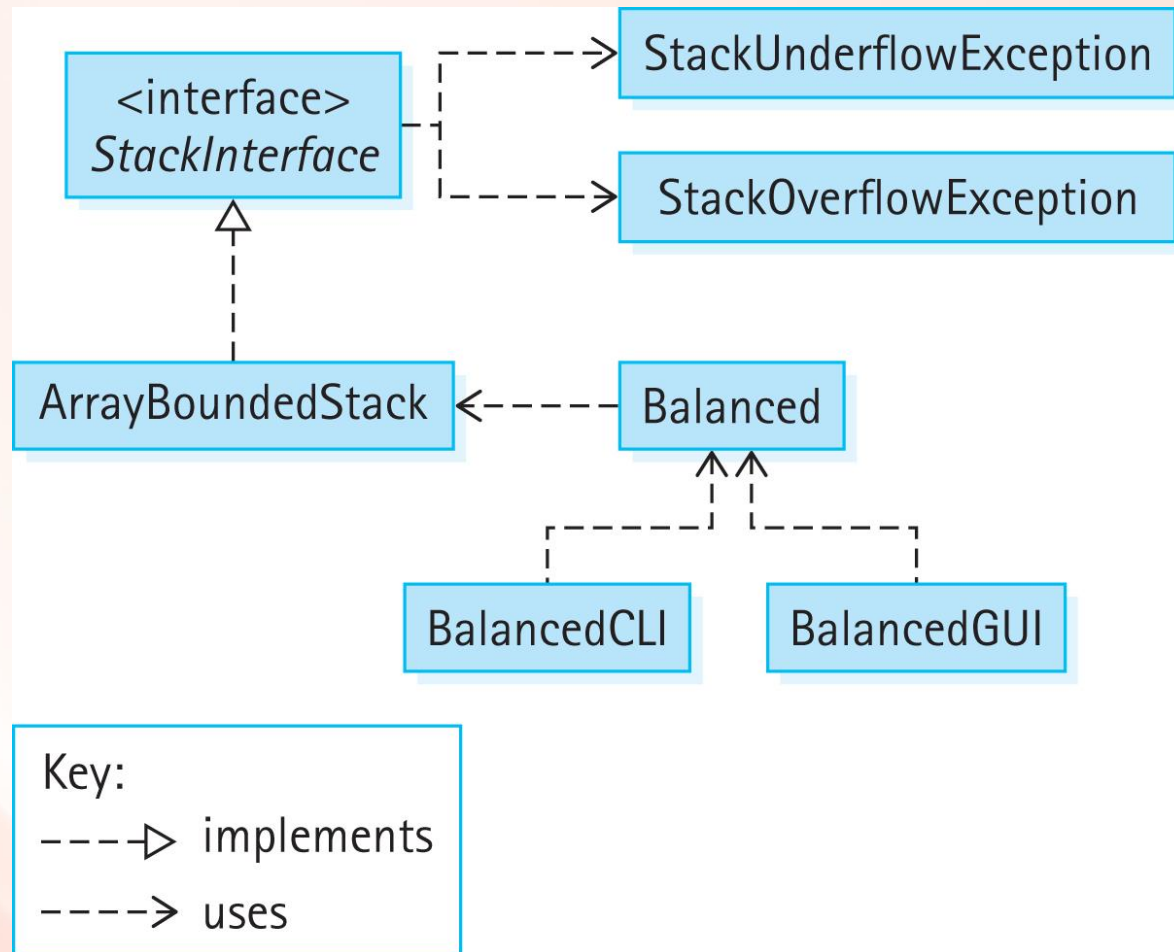
2.7 Introduction to Linked Lists
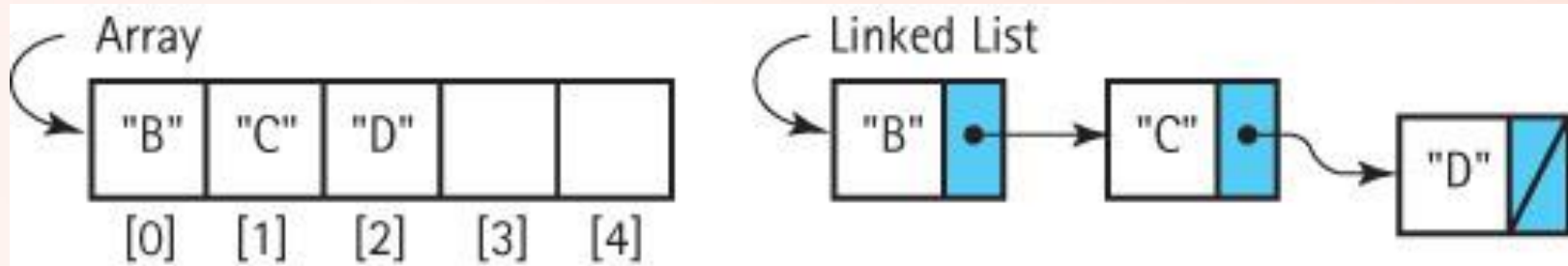
- Arrays and Linked Lists are both basic building blocks for implementing data structures.
- They differ in terms of
  - use of memory
  - manner of access
  - efficiency for various operations
  - language support

Nodes of a Linked-List

- A node in a linked list is an object that holds some important information, such as a string, plus a link to the exact same type of object, i.e. to an object of the same class.

- **Self-referential class** A class that includes an instance variable or variables that can hold a reference to an object of the same class.

- To support our linked implementations of our ADTs we create the self-referential `LLNode` class …

LLNode Class

```
//------------------------------------------------------
// LLNode.java     by Dale/Joyce/Weems        Chapter 2
//
// Implements <T>  nodes for a Linked List.
//------------------------------------------------------

package support;

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

   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<T> link){ this.link = link;}

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

Using the LLNode class

1: `LLNode<String> sNode1 = new LLNode<String>("basketball");`
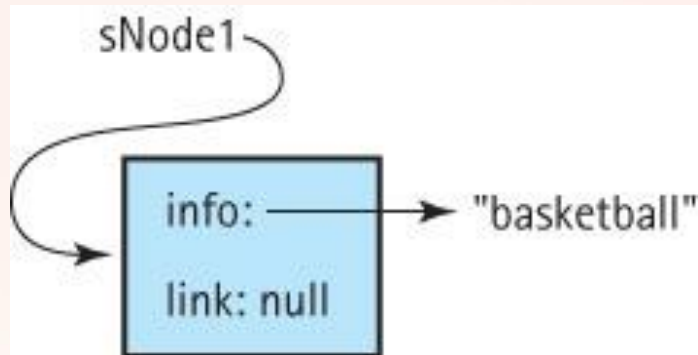
2: suppose that in addition to `sNode1` we have `sNode2` with info "baseball" and perform

`sNode1.setLink(sNode2);`
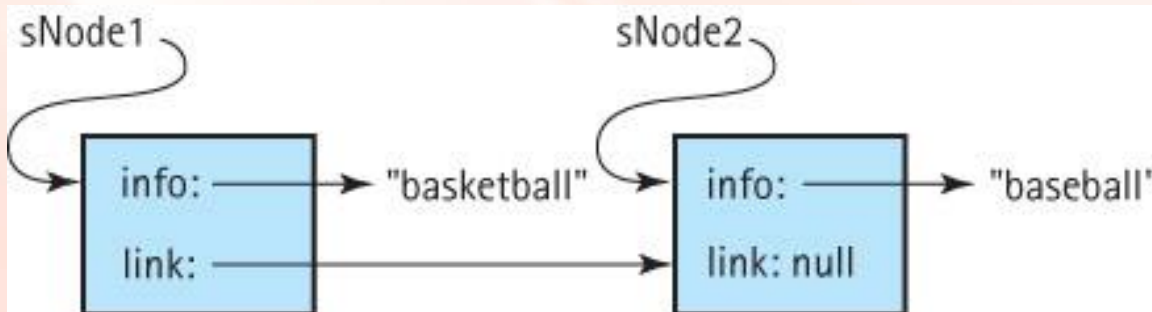
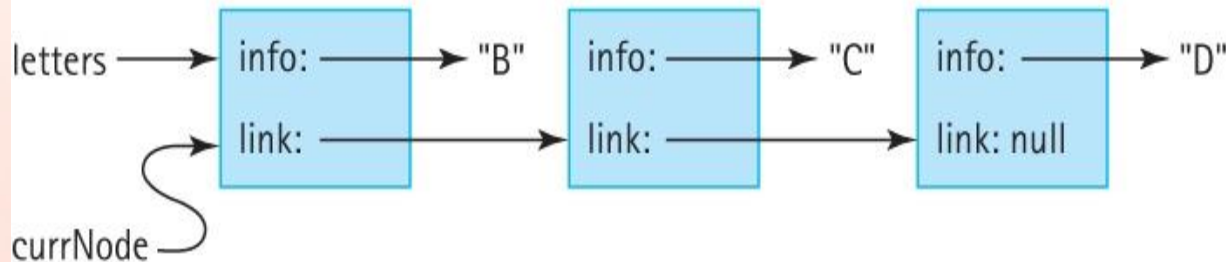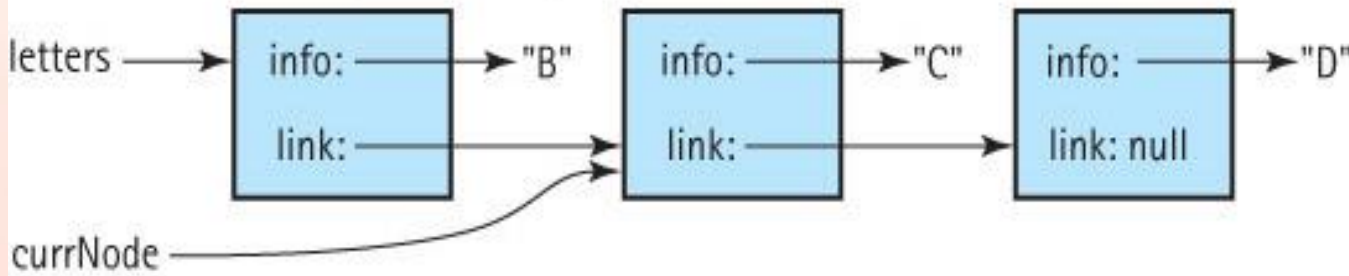# Traversal of a Linked List
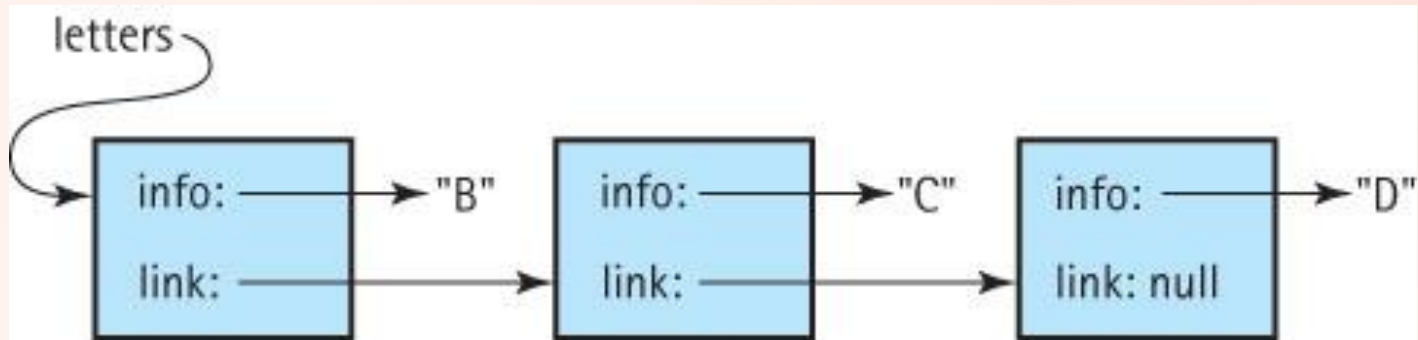

```
LLNode<String> currNode = letters;
while (currNode != null)
{
   System.out.println(currNode.getInfo());
   currNode = currNode.getLink();
}
```
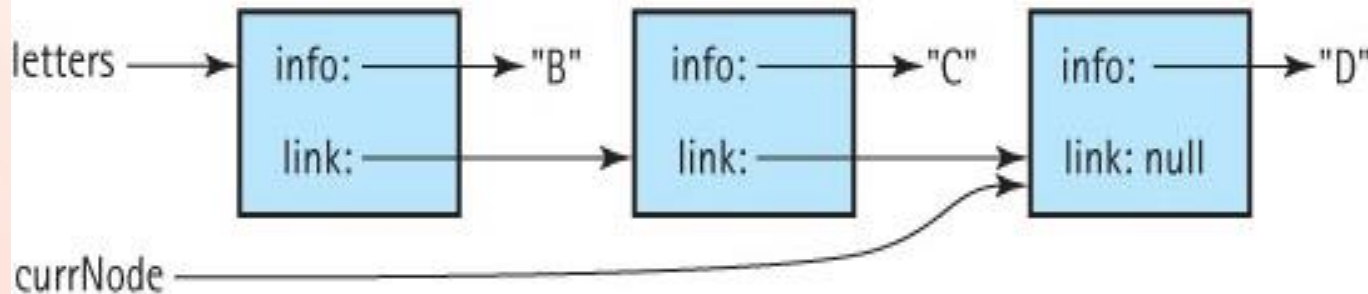
Tracing a Traversal (part 1)

```
LLNode<String> currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Internal View

Output


After "LLNode<String> currNode = letters;":

Tracing a Traversal (part 2)

```
LLNode<String> currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Internal View

Output

B

After first time through *while* loop:

letters → info: → "B" link: → info: → "C" link: → info: → "D" link: null

currNode →

Tracing a Traversal (part 3)

```
LLNode<String> currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Internal View

Output

B
C

After second time through *while* loop:

letters → info: → "B"   info: → "C"   info: → "D"
          link: →   link: →   link: null

currNode →

Tracing a Traversal (part 4)

```
LLNode<String> currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Internal View

Output

B
C
D

After third time through *while* loop:

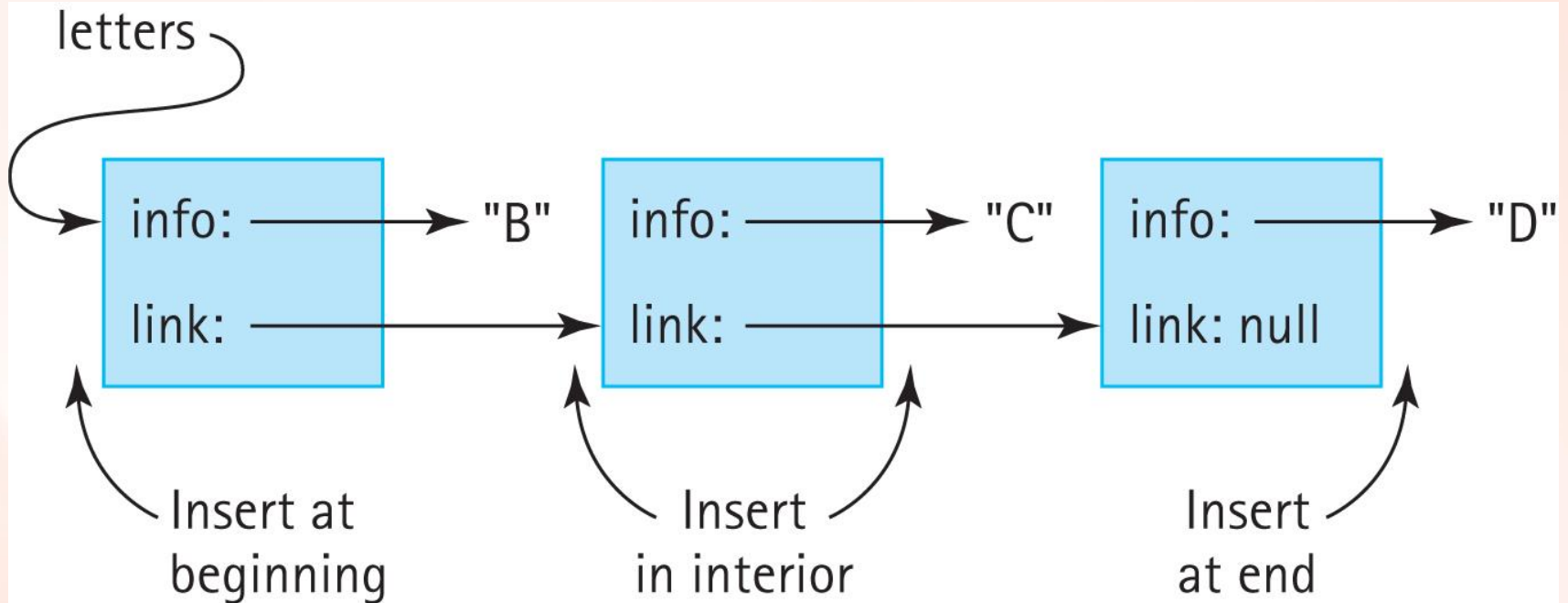letters → info: → "B" link: → info: → "C" link: → info: → "D" link: null
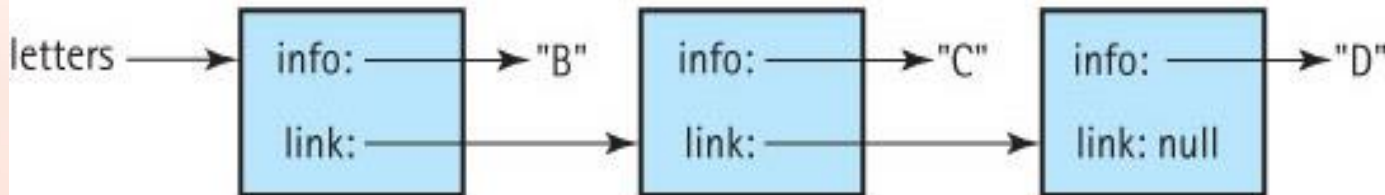
currNode: null

The *while* condition is now false

Three general cases of insertion
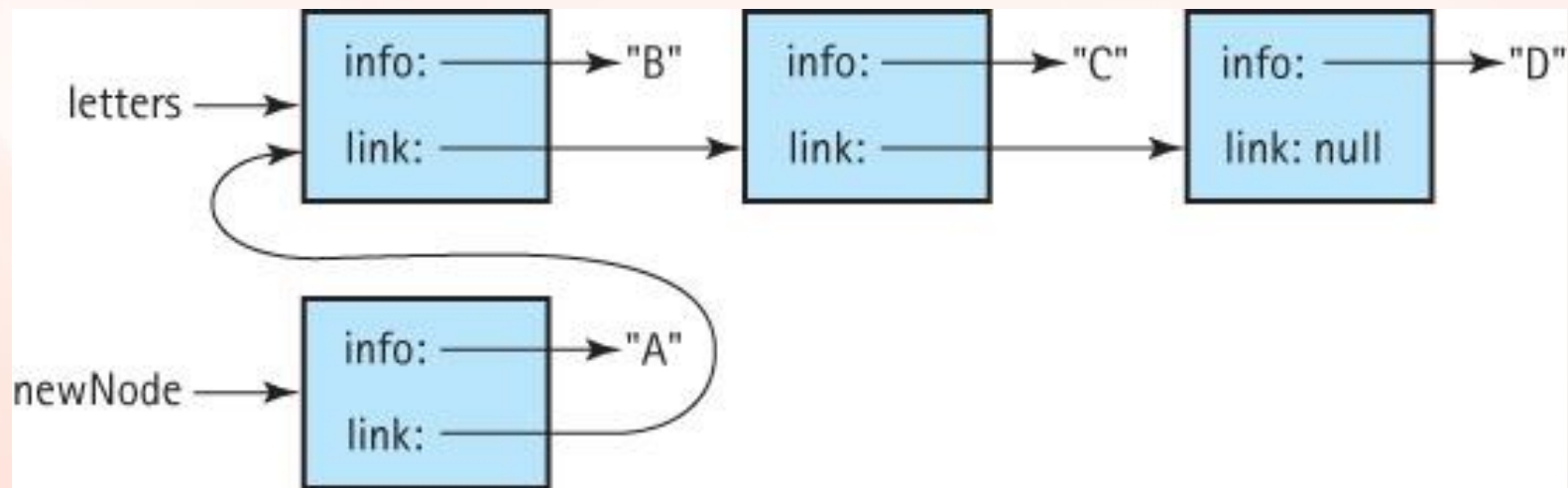
Insertion at the front (part 1)

Suppose we have the node newNode to insert into the beginning of the letters linked list:
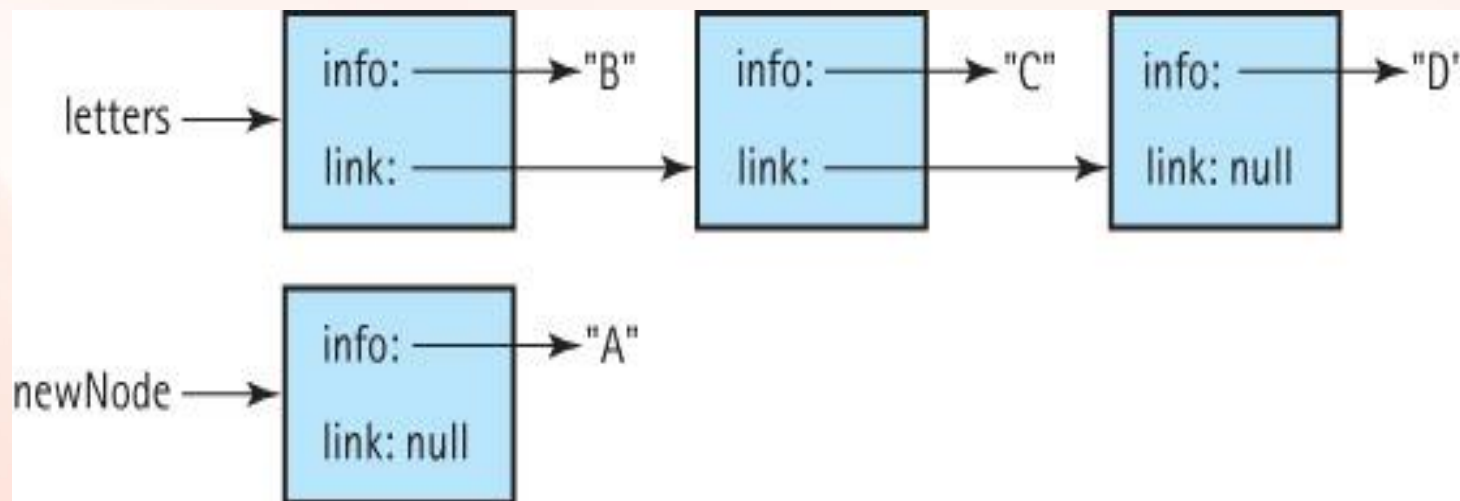
Insertion at the front (part 2)

Our first step is to set the link variable of the newNode node to point to the beginning of the list :
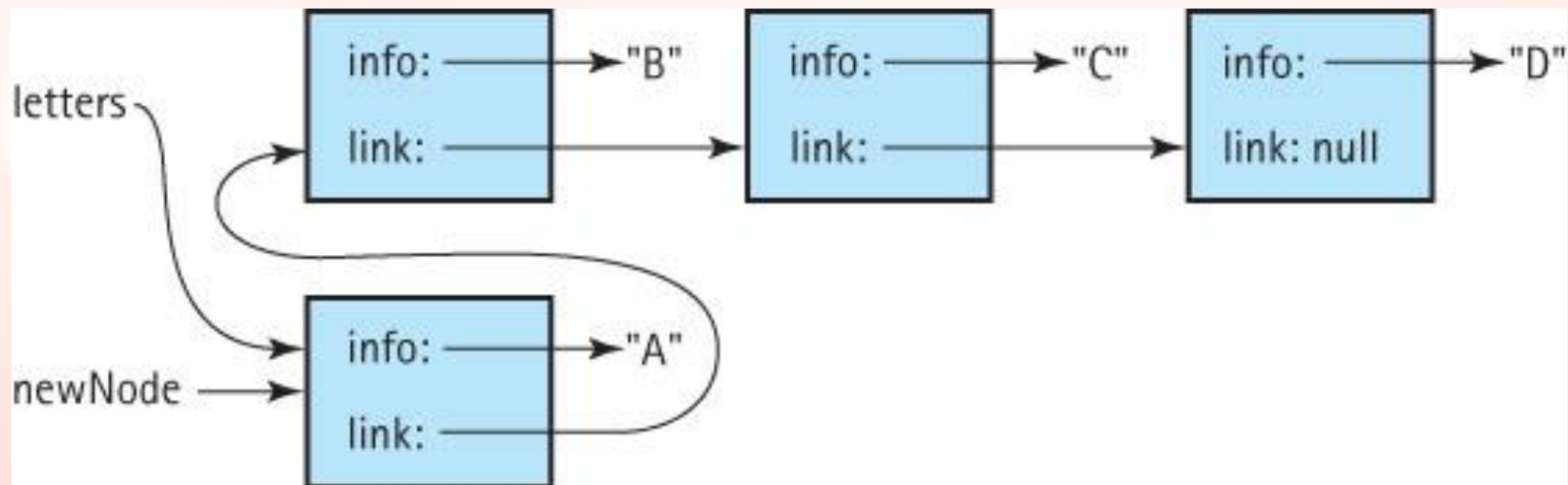
`newNode.setLink(letters);`

Insertion at the front (part 3)

To finish the insertion we set the letters variable to point to the newNode, making it the new beginning of the list:
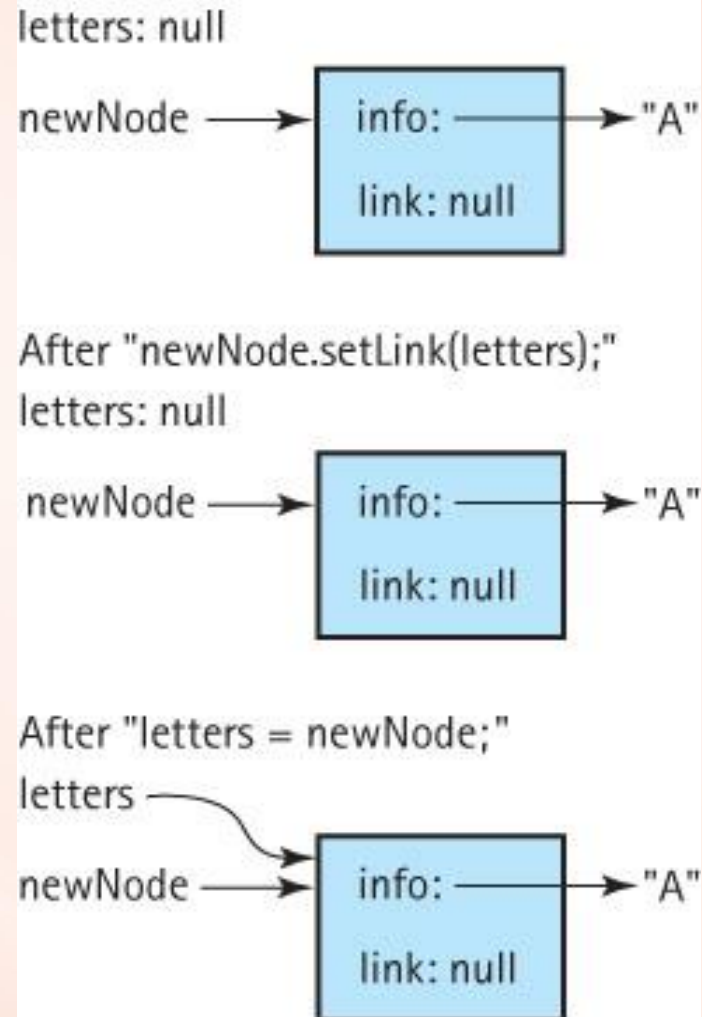
```
letters = newNode;
```

Insertion at front of an empty list

The insertion at the front code is

```
newNode.setLink(letters);
letters = newNode;
```

What happens if our insertion code is called when the linked list is empty?

As can be seen at the right the code still works, with the new node becoming the first and only node on the linked list.

letters: null

newNode → info: ─── → "A"
link: null

After "newNode.setLink(letters);"
letters: null

newNode ─── info: ─── → "A"
link: null

After "letters = newNode;"
letters ─┐
newNode ─── info: ─── → "A"
link: null

2.8 A Link-Based Stack

- In this section we study a link-based implementation of the Stack ADT.

- After discussing the link-based approach we compare our stack implementation approaches.

The LinkedStack Class

We only need a single instance variable, the "pointer" to the 'top' of the stack:

```
package ch02.stacks;
import support.LLNode;

public class LinkedStack<T> implements StackInterface<T>
{
    protected LLNode<T> top; // reference to the top of this stack

    public LinkedStack()
    {
        top = null;
    }

. . .
```
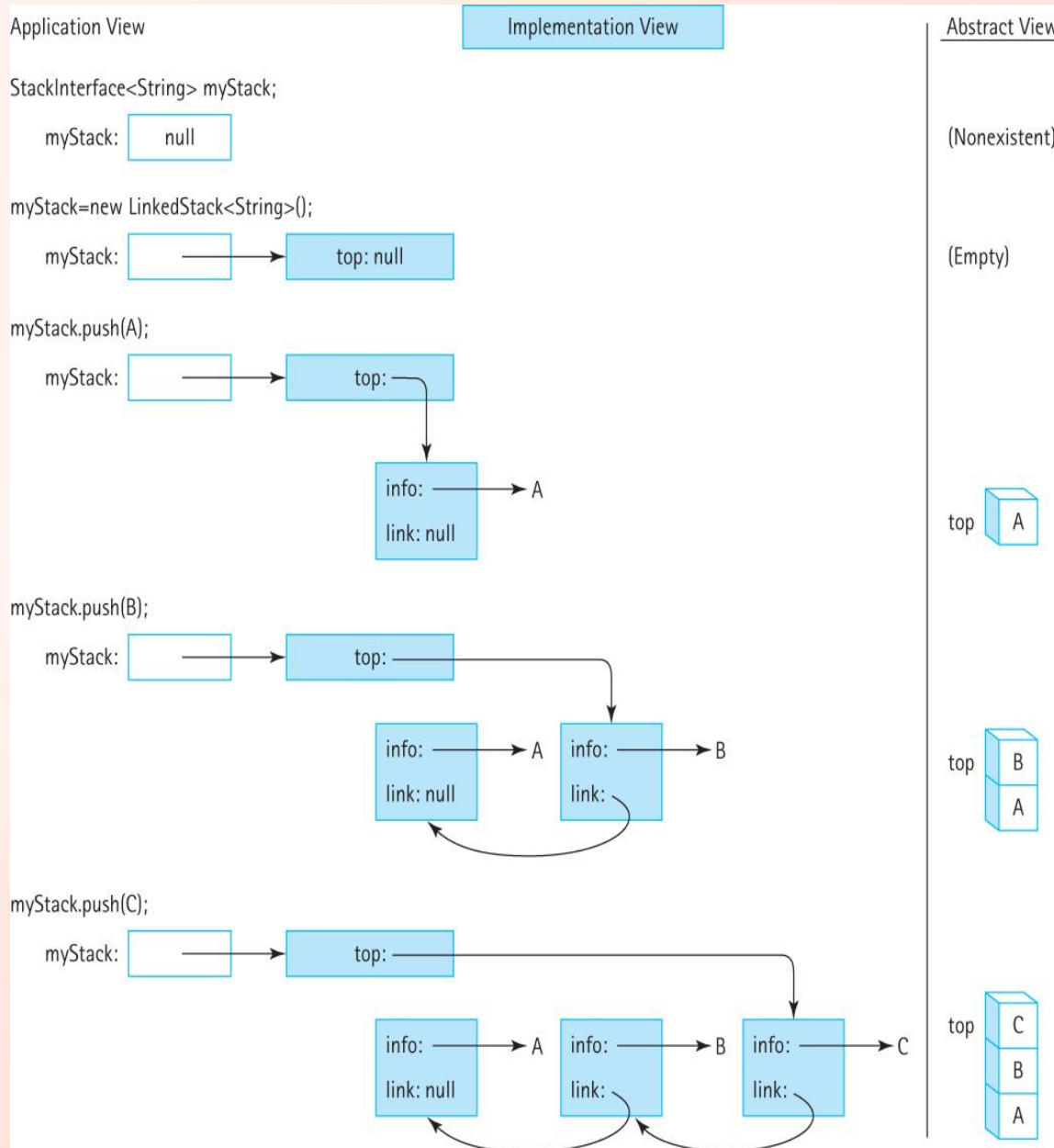
Visualizing the push operation

(insertion at front of a linked-list)

StackInterface<String> myStack;
myStack = new LinkedStack<String>();
myStack.push(A);
myStack.push(B);
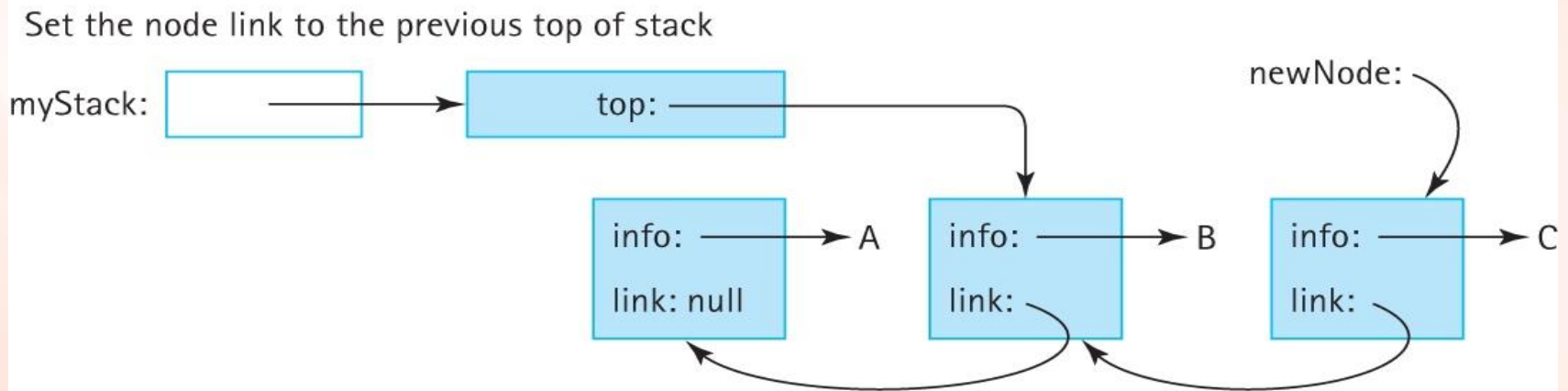**myStack.push(C);**

The push(C) operation (step 1)

- **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

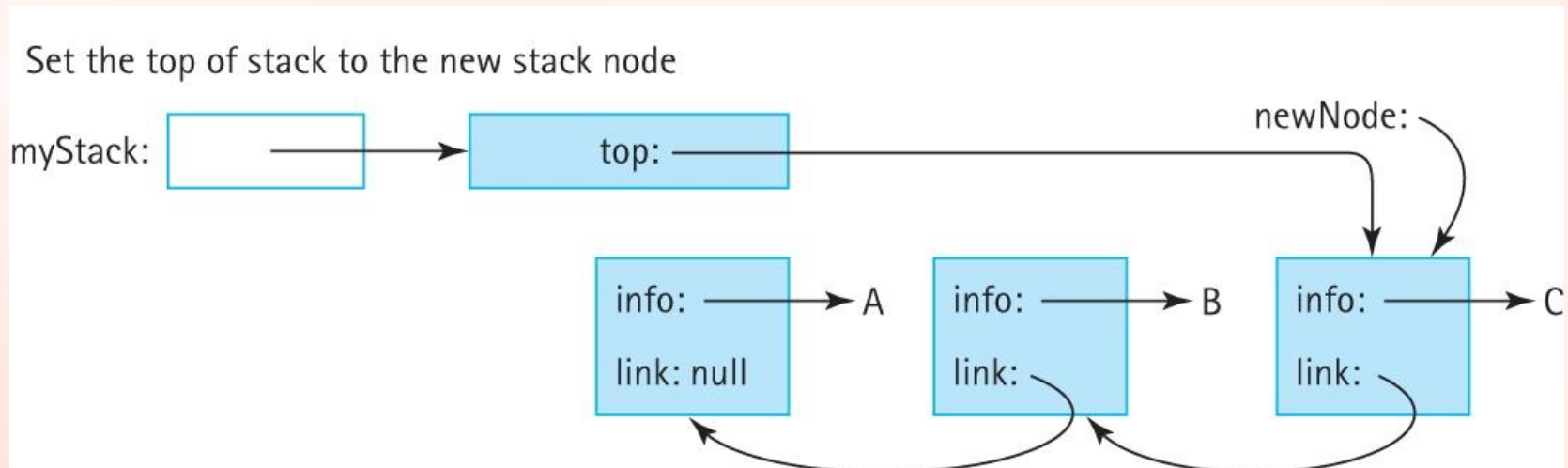Allocate space for the next stack node and set the node info to element
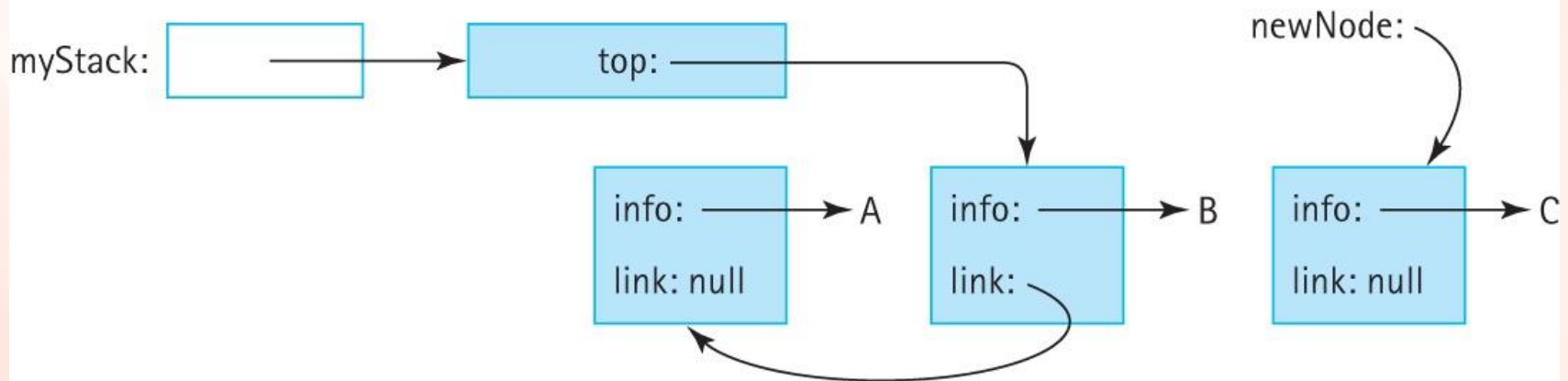
The push(C) operation (step 2)

- 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

Set the node link to the previous top of stack

The push(C) operation (step 3)

- 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**

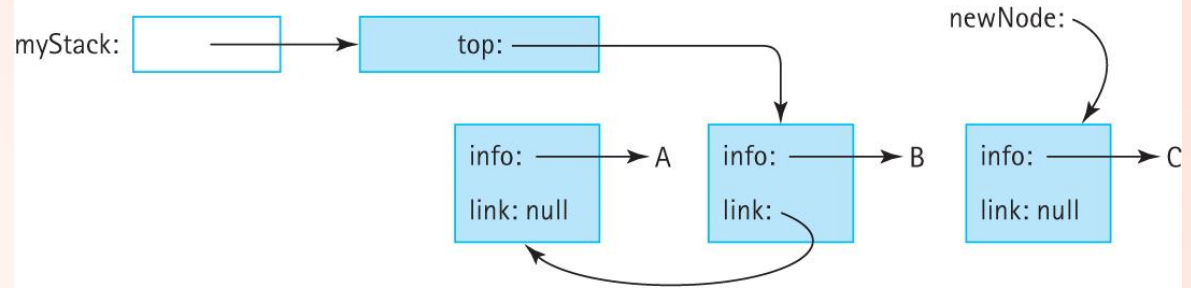Set the top of stack to the new stack node
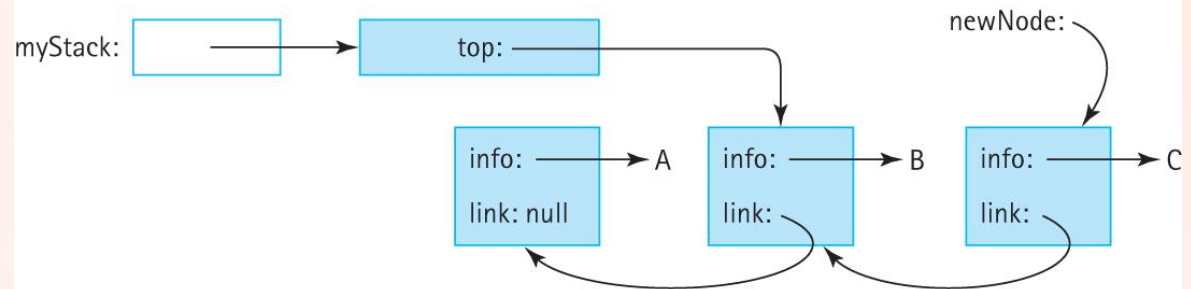
Code for the push method

```
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;
}
```
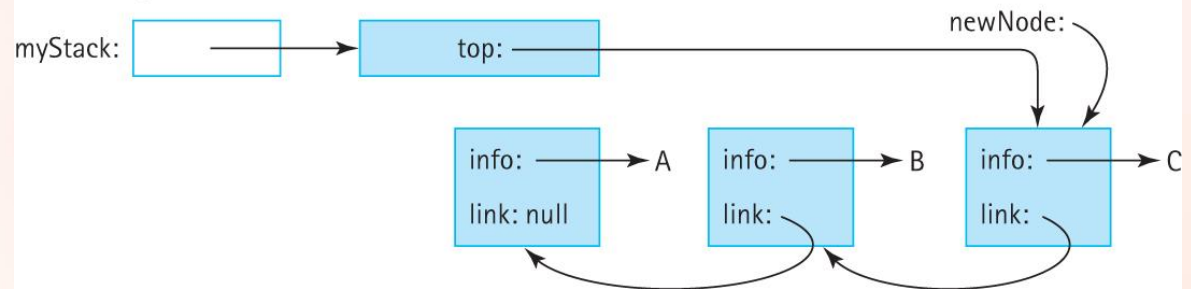
Result of push onto empty stack

Allocate space for the next stack node and set the node info to element

Set the node link to the previous top of stack

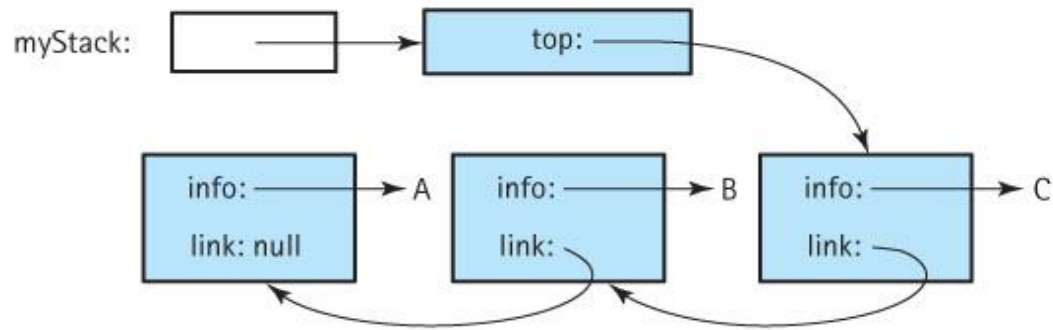Set the top of stack to the new stack node

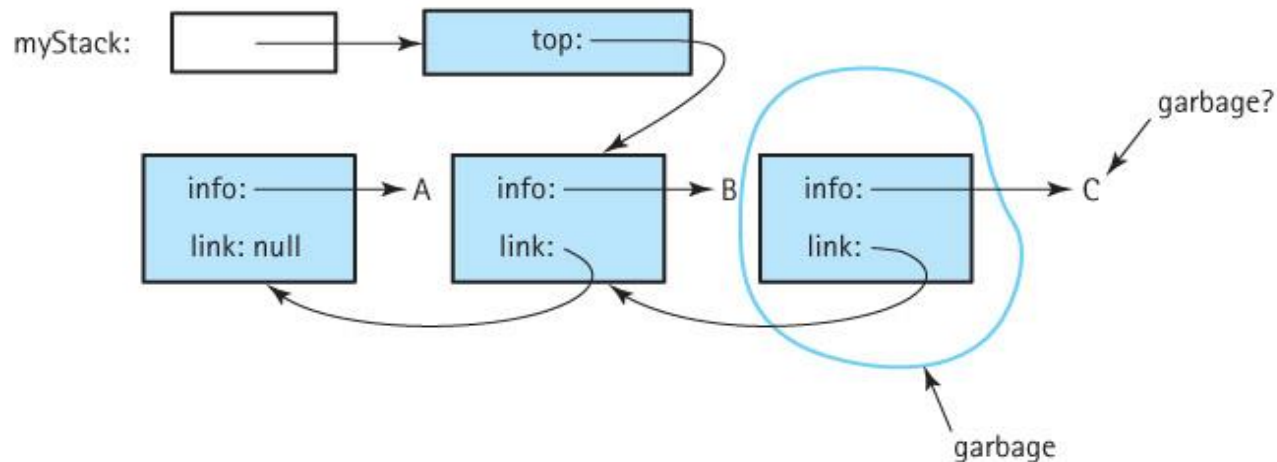# Code for the pop method

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

Pop from a stack with three elements

The remaining operations

```
public T top()
{
    if (isEmpty())
        throw new StackUnderflowException("Top attempted on an empty stack.");
    else
        return top.getInfo();
}



public boolean isEmpty()
{
    return (top == null);
}



public boolean isFull()
// Returns false – a linked stack is never full.
{
    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 maximum size
  - Link-based: takes space proportional to actual size of the stack (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: $O(1)$

Which is better?

- 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.

- 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.

Our Stack Classes and Interfaces
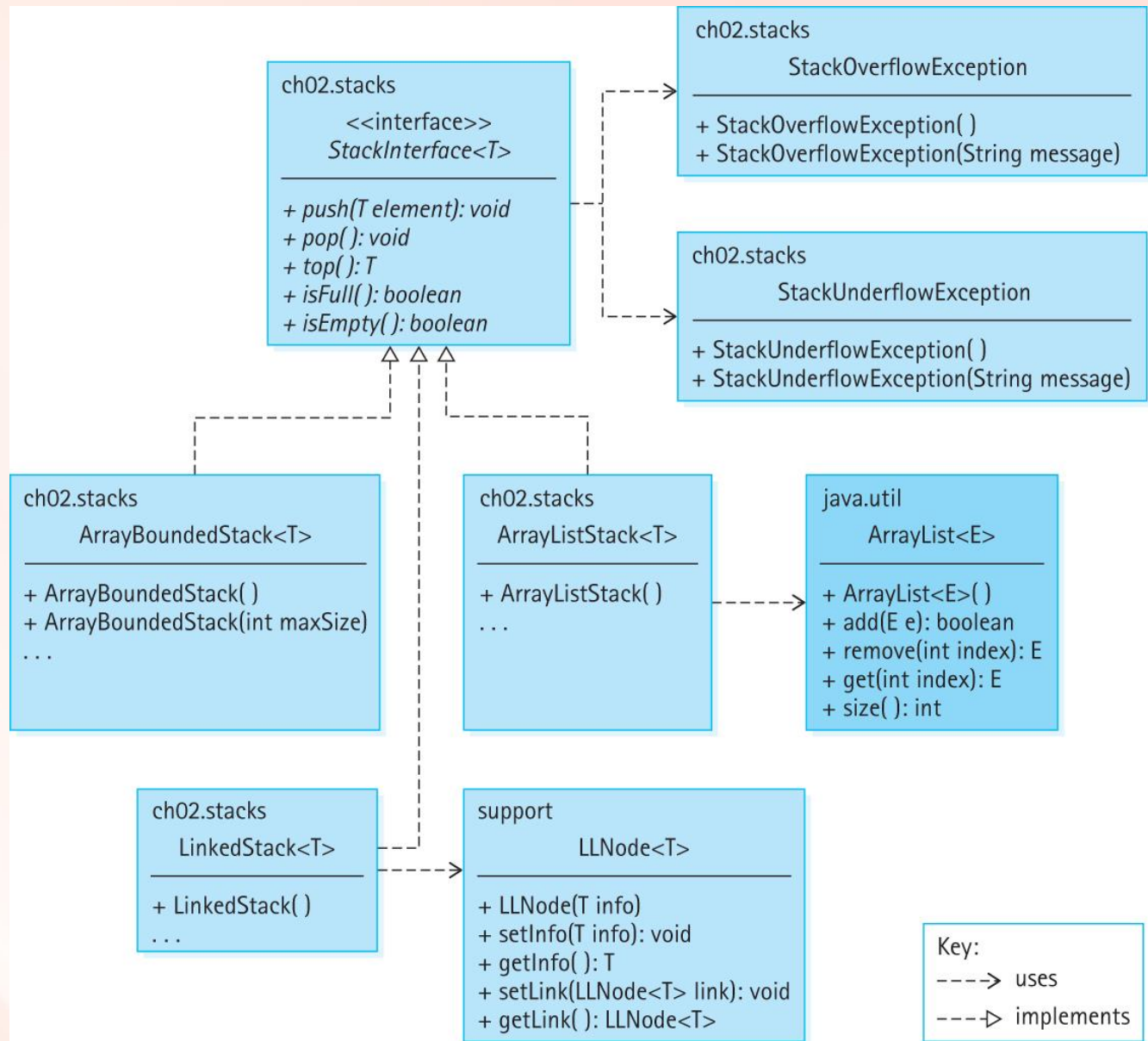
2.9 Application: Postfix Expression Evaluator

- Postfix notation is a notation for writing arithmetic expressions in which the operators appear after their operands.

- For example, instead of writing

    (2 + 14) × 23

we write

    2 14 + 23 ×

Examples

Postfix Expression	Infix Equivalent	Result
4 5 7 2 + – ×	$4 \times (5 - (7 + 2))$	–16
3 4 + 2 × 7 /	$((3 + 4) \times 2)/7$	2
5 7 + 6 2 – ×	$(5 + 7) \times (6 - 2)$	48
4 2 3 5 1 – + × + ×	$? \times (4 + (2 \times (3 + (5 - 1))))$	not enough operands
4 2 + 3 5 1 – × +	$(4 + 2) + (3 \times (5 - 1))$	18
5 3 7 9 + +	$(3 + (7 + 9)) ... 5???$	too many operands

Postfix Expression Evaluation Algorithm

while more items exist
   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

Example

$$5 \quad 7 \quad + \quad 6 \quad 2 \quad - \quad *$$

Example

$$5 \quad 7 \quad + \quad 6 \quad 2 \quad - \quad *$$

$$5$$

Example

5 7 + 6 2 — *

7
5
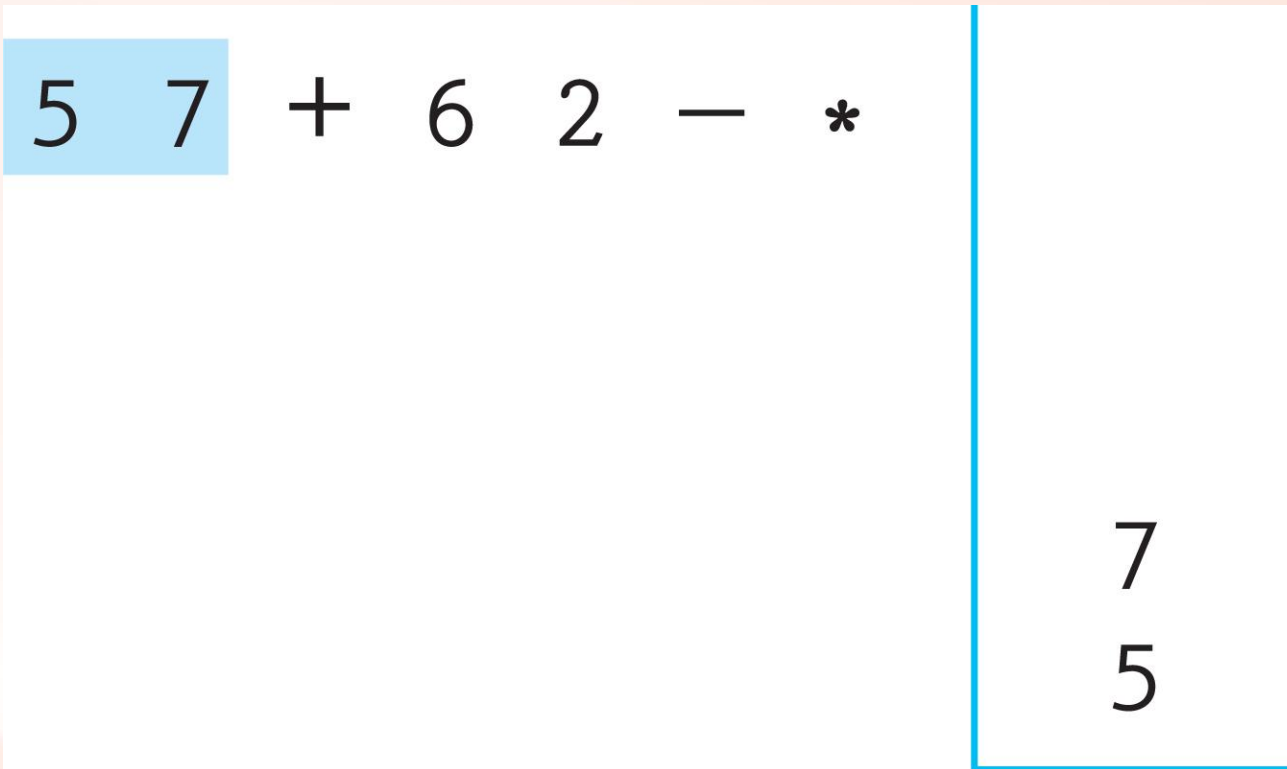
Example

$$5 \quad 7 \quad + \quad 6 \quad 2 \quad - \quad *$$

$$12$$

Example

$$5 \quad 7 \quad + \quad 6 \quad 2 \quad - \quad *$$

$$\begin{array}{c} 2 \\ 6 \\ 12 \end{array}$$

Example

5 7 + 6 2 — *

4
12

5 7 + 6 2 − *

48

Example

$$5 \quad 7 \quad + \quad 6 \quad 2 \quad - \quad *$$

Result

48

Code and Demo

- Instructors can now walk through and demonstrate the code contained in `PostFixEvaluator.java` from the `ch02.postfix` package and `PFixCLI.java` and/or `PFixGUI.java` from the `ch02.apps` package.

2.10 Stack Variations

- Changes we could make to our Stack
  - Use stacks of class `Object` instead of generic stacks
  - Implement the classic *pop* operation (both remove and return top element) instead of the combination of *pop* and *top* operations
  - Instead of throwing exceptions we could
    - for example, state as a precondition that *pop* will not be called on an empty stack
    - for example, state that *pop* returns a boolean indicating success

The Java Collections Framework
`Stack`

- extends the `Vector` class which extends the `Object` class
  - inherits several useful operations such as `capacity`, `clear`, `clone`, `contains`, `isEmpty`, `toArray`, and `toString`.

- in addition to our `push` and `top` methods, implements
  - the classic *pop* operation (both remove and return top element)
  - a *search* operation that returns the position of a specified object in the stack