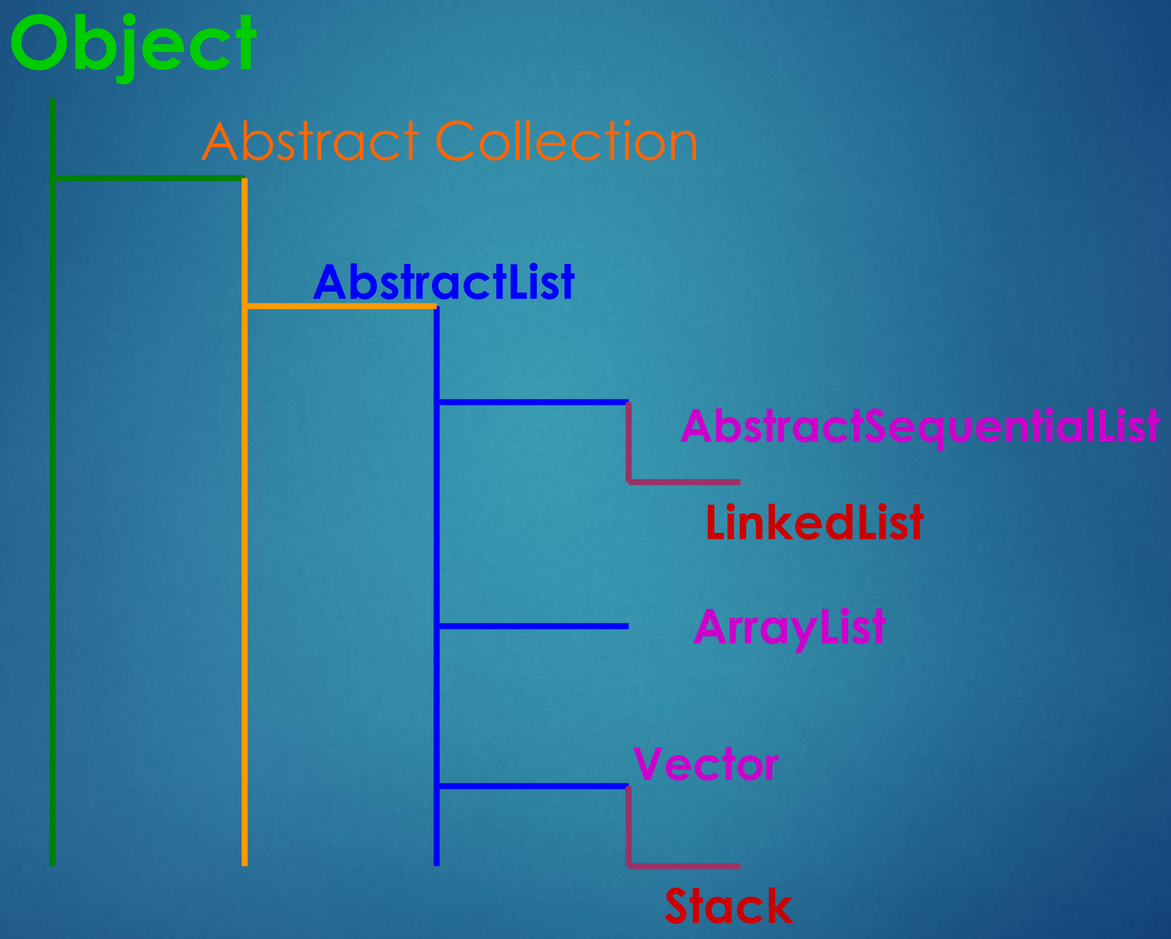
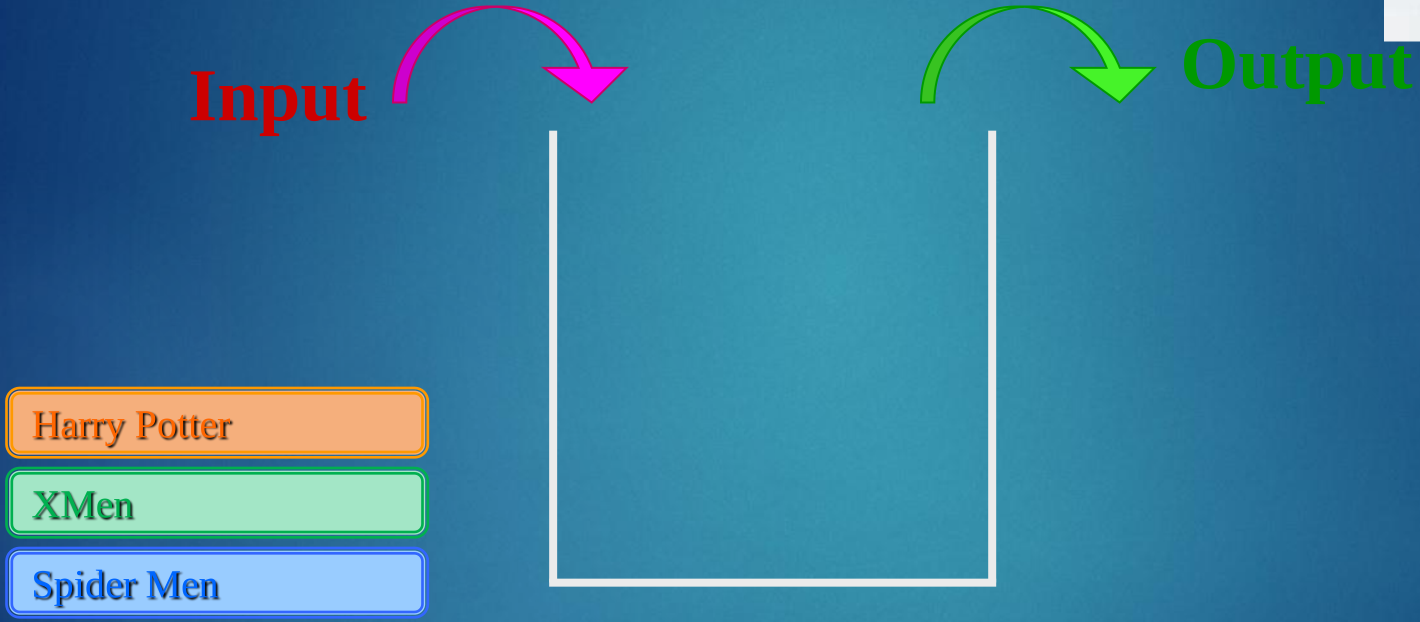
Stack

NGĂN XẾP

Stack in Java



What is Stack ?



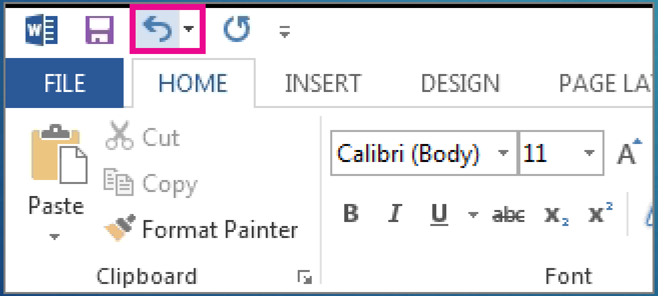
Definition of Stack

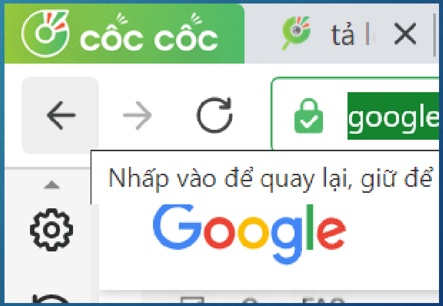
- Items are added to and removed from the top of the pile. Consider the pile of papers on your desk. Suppose you add papers only to the top of the pile or remove them only from the top of the pile. At any point in time, the only paper that is visible is the one on top. What you have is a *stack*.
- stack is a *last-in, first-out* or *LIFO* data structure

Application of Stack

Example 6.1: Internet Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.

Example 6.2: Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.

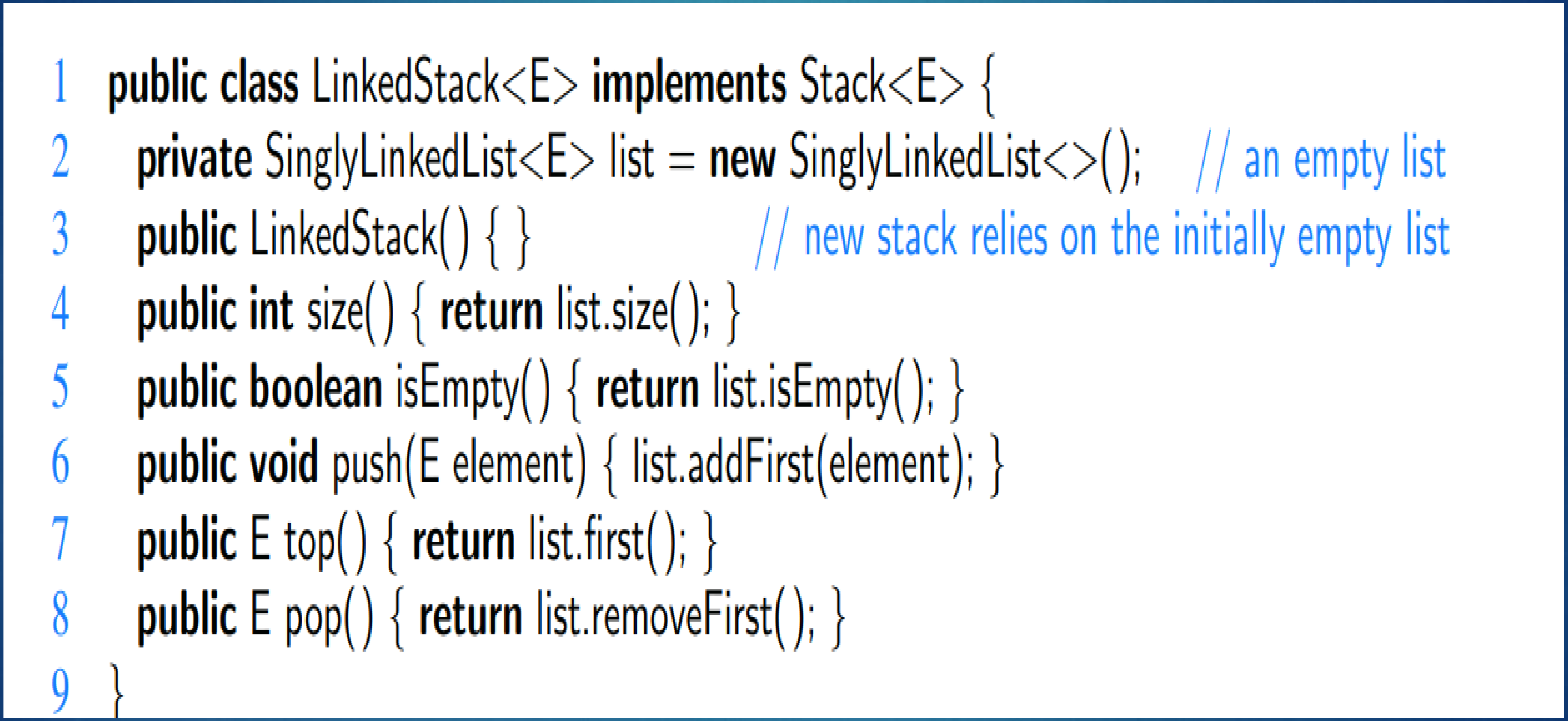




How to build Stack

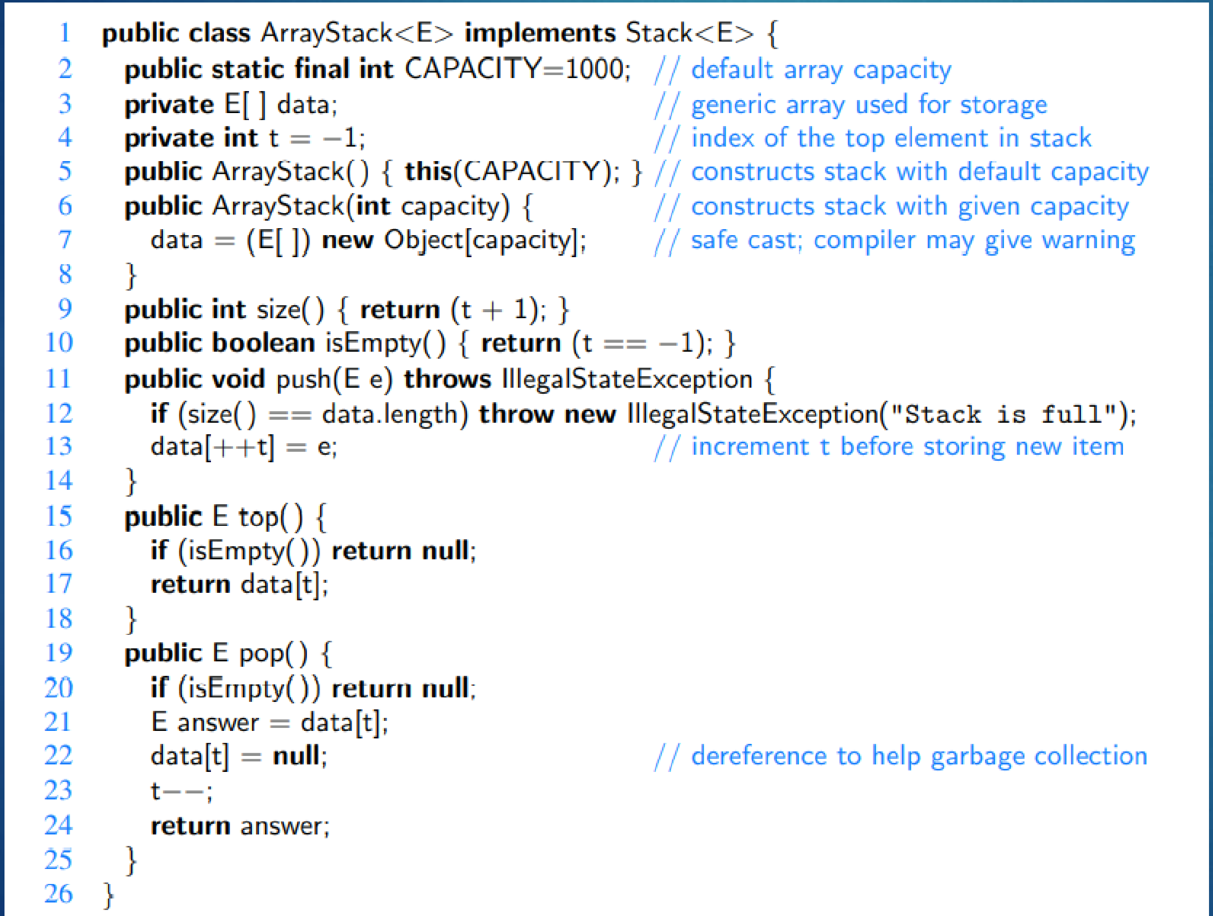
- Array ?
- Linked List?

Implement Stack with Singly Linked List



```
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedStack() { }                  // new stack relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(E element) { list.addFirst(element); }
  public E top() { return list.first(); }
  public E pop() { return list.removeFirst(); }
}
```

Implement Stack with Array



```
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000;  // default array capacity
  private E[ ] data;                      // generic array used for storage
  private int t = −1;                     // index of the top element in stack
  public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
  public ArrayStack(int capacity) {       // constructs stack with given capacity
    data = (E[ ]) new Object[capacity];   // safe cast; compiler may give warning
  }
  public int size( ) { return (t + 1); }
  public boolean isEmpty( ) { return (t == −1); }
  public void push(E e) throws IllegalStateException {
    if (size( ) == data.length) throw new IllegalStateException("Stack is full");
    data[++t] = e;                        // increment t before storing new item
  }
  public E top( ) {
    if (isEmpty( )) return null;
    return data[t];
  }
  public E pop( ) {
    if (isEmpty( )) return null;
    E answer = data[t];
    data[t] = null;                       // dereference to help garbage collection
    t−−;
    return answer;
  }
}
```

Code Fragment 6.2: Array-based implementation of the Stack interface.

Using Stack in Java

- Create a Stack

```
import java.util.Stack;
....
Stack s = new Stack();
```

- Put element in Stack

```
s.push(“learn Stack”);
s.push(“practise Stack”);
```

- Remove top element in Stack

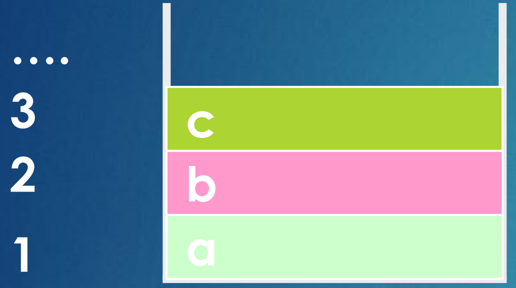
```
s.pop();
```

Methods of Stack

- The Stack interface extends the Container interface defined in Program _. Hence, it comprises all of the methods inherited from Container plus the three methods peek(), push(), and pop().
- Meaning of methods:
  - ✓ empty() return true if no element in stack.
  - ✓ **peek()** , firstElement() return top element of stack but not delete top element in stack.
  - ✓ **pop()** return top element of stack and delete top element in stack
  - ✓ **push(Object obj)**, add(Object obj) push the element in stack
  - ✓ elements() return iterator of elements
  - ✓ size() return the size of stack

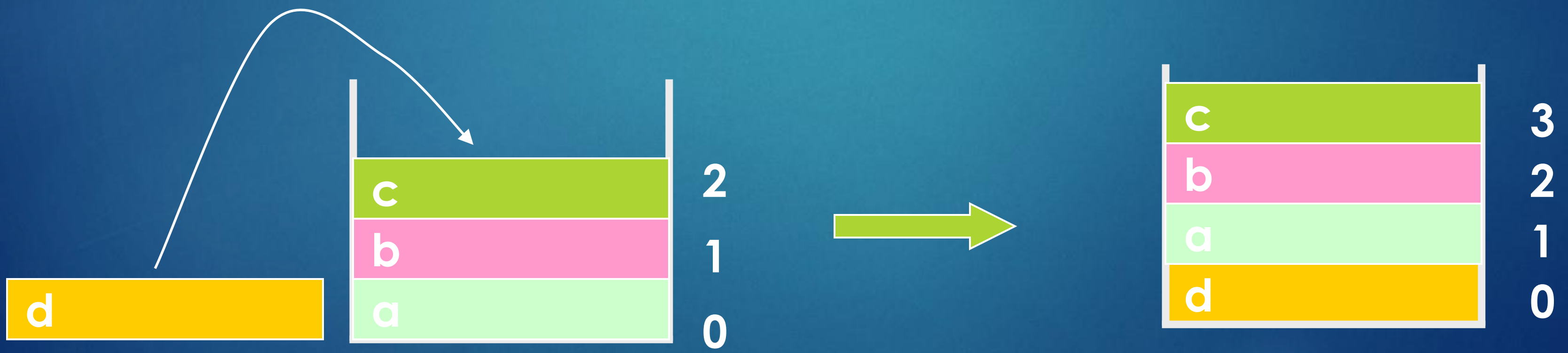
Methods of Stack (cont)

- ✓ search(Object obj) return index of element in stack



s.search(“a”) → 1

- ✓ add (int position, Object obj)



s.add(0,“d”)

Methods of Stack (cont)

- remove(int position) remove an object at input position.
- remove(Object obj) remove an object in Stack that equal input object.
- removeAllElements() return a empty Stack

Try to test

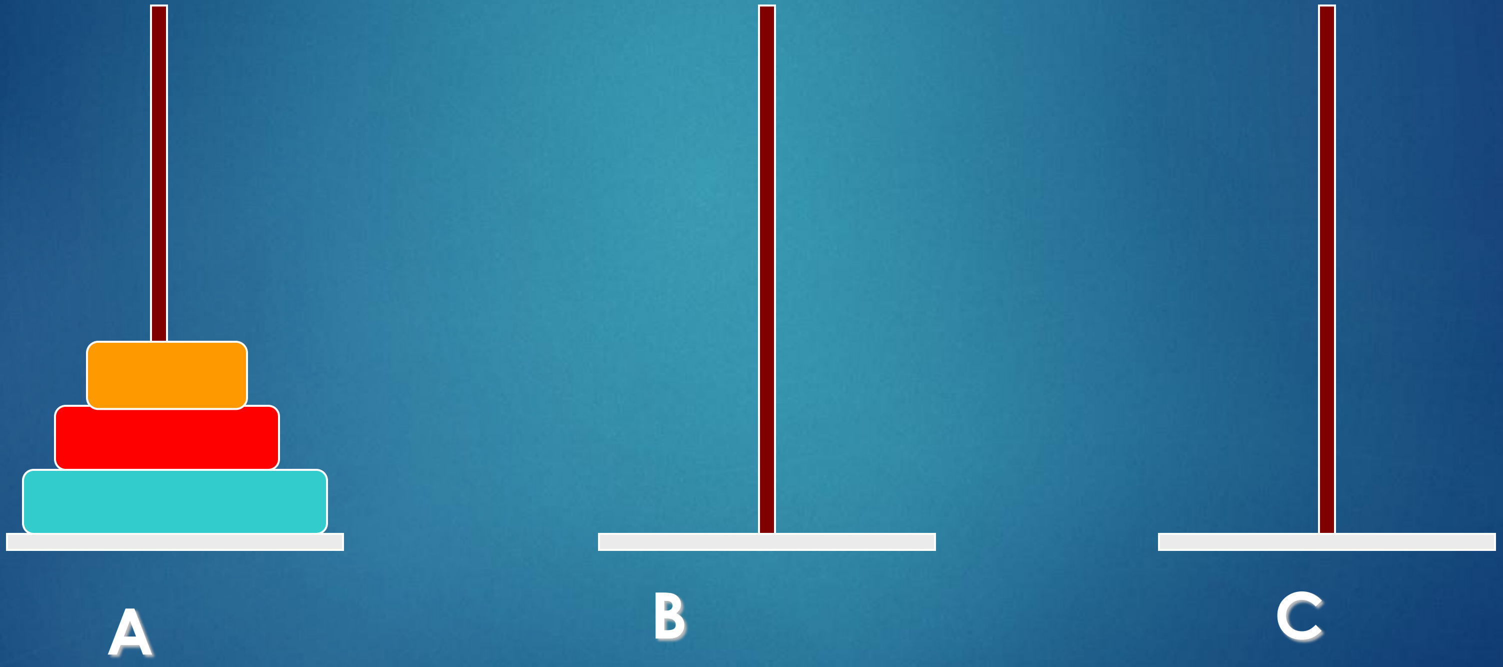
```
Stack<Integer> test = new Stack<>();
        test.push(2);
        test.push(4);
        test.push(5);
        test.push(7);
        System.out.println(test);
        test.peek();
        System.out.println(test);
        test.pop();
        System.out.println(test);
```

Predict the result of code

```
Stack<Integer> test1 = new Stack<>();
test1.push(0);
test1.peek();
test1.push(1);
test1.push(-1);
test1.pop();
test1.push(2);
System.out.println(test1);
```

Application of Stack

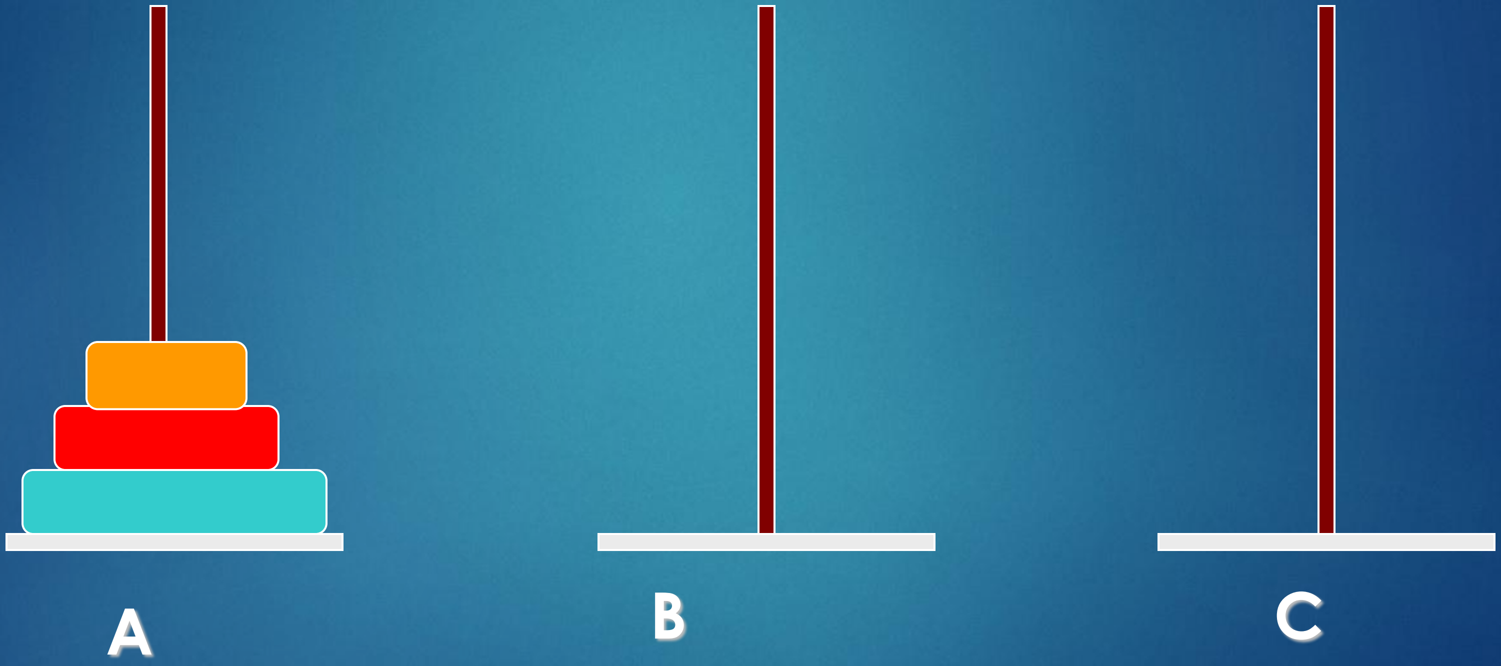
- Ha Noi tower game



How to move 3 disks from A column to C column with minimum of steps?

How to implement Stack

- Solution



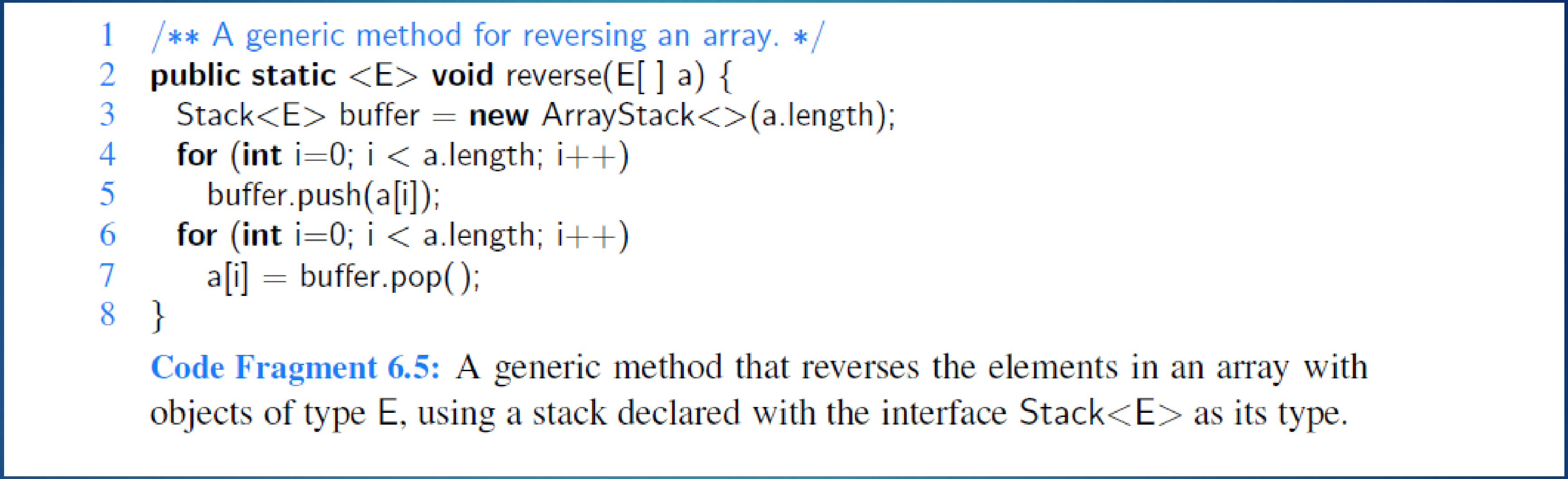
Application of Stack

- How to program this game?

1. Each column is a Stack
2. Using push, pop, peek to add and remove disk

Small exercises

1. Reverse an array by using Stack (pg 234)



```
/** A generic method for reversing an array. */
public static <E> void reverse(E[ ] a) {
  Stack<E> buffer = new ArrayStack<>(a.length);
  for (int i=0; i < a.length; i++)
    buffer.push(a[i]);
  for (int i=0; i < a.length; i++)
    a[i] = buffer.pop( );
}
```

Code Fragment 6.5: A generic method that reverses the elements in an array with objects of type E, using a stack declared with the interface Stack<E> as its type.

Small exercises

1. Matching Parentheses and HTML Tags(pg 234)

(Check each opening symbol must match its corresponding closing symbol.)

- Symbol:
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”
- An Algorithm for Matching Delimiters

We can use a stack to perform this task with a single left-to-right scan of the original string.

Each time we encounter an opening symbol, we push that symbol onto the stack, and each time we encounter a closing symbol, we pop a symbol from the stack (assuming it is not empty) and check that these two symbols form a valid pair.

If we reach the end of the expression and the stack is empty, then the original expression was properly matched. Otherwise, there must be an opening delimiter on the stack without a matching symbol. If the length of the original expression is n, the algorithm will make at most n calls to push and n calls to pop

Small exercises

2. Matching Parentheses(pg 234)

(Check each opening symbol must match its corresponding closing symbol.)

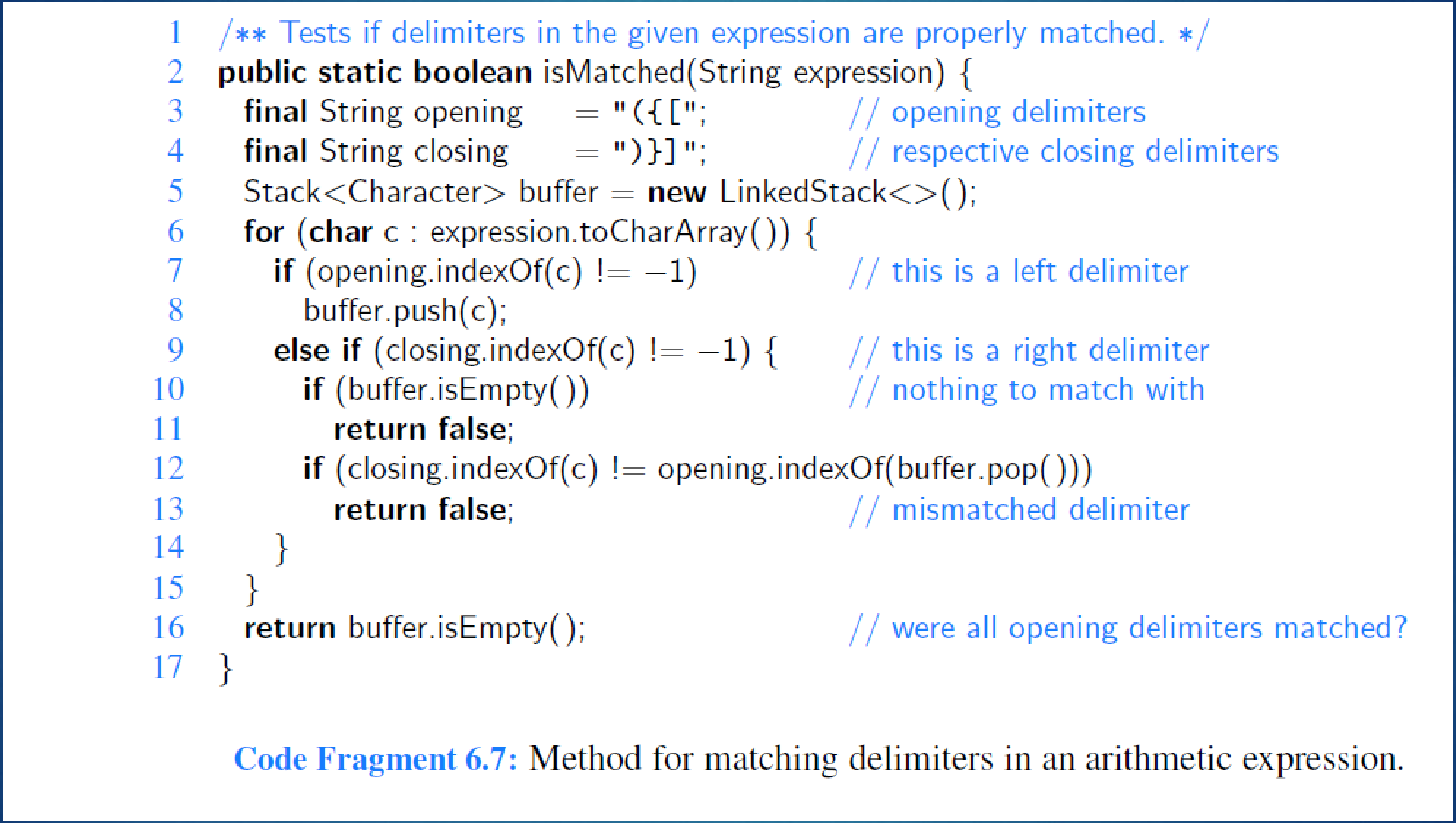
- Symbol:
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”
- An Algorithm for Matching Delimiters

We can use a stack to perform this task with a single left-to-right scan of the original string.

Each time we encounter an opening symbol, we push that symbol onto the stack, and each time we encounter a closing symbol, we pop a symbol from the stack (assuming it is not empty) and check that these two symbols form a valid pair.

If we reach the end of the expression and the stack is empty, then the original expression was properly matched. Otherwise, there must be an opening delimiter on the stack without a matching symbol. If the length of the original expression is n, the algorithm will make at most n calls to push and n calls to pop

Small exercises

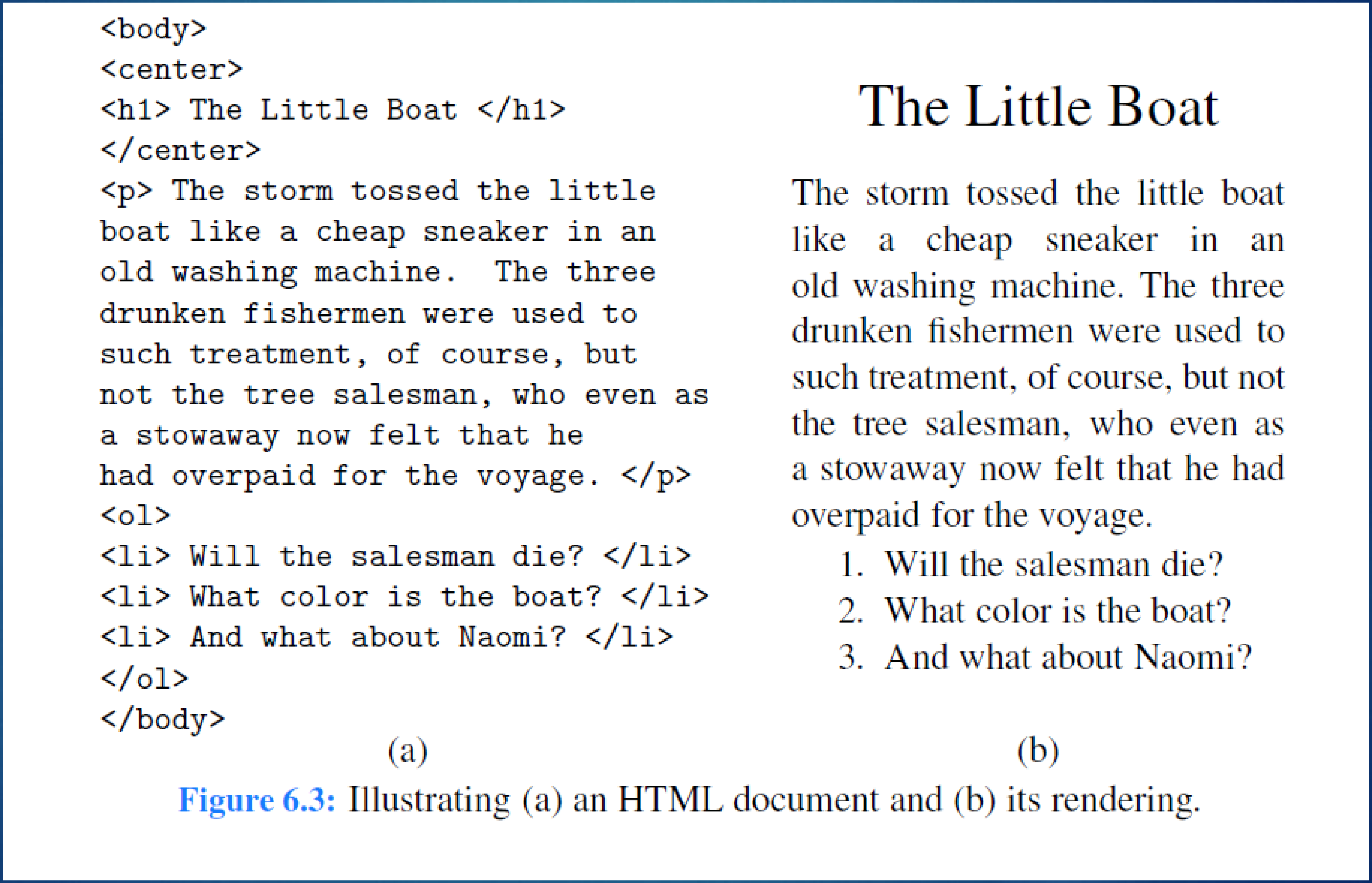


```
/** Tests if delimiters in the given expression are properly matched. */
public static boolean isMatched(String expression) {
  final String opening    = "({[";          // opening delimiters
  final String closing    = ")}]";          // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>();
  for (char c : expression.toCharArray()) {
    if (opening.indexOf(c) != −1)           // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) {    // this is a right delimiter
      if (buffer.isEmpty())                 // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
        return false;                       // mismatched delimiter
    }
  }
  return buffer.isEmpty();                  // were all opening delimiters matched?
}
```

Code Fragment 6.7: Method for matching delimiters in an arithmetic expression.

Small exercises

3. Matching Tags in a Markup Language(pg 236)



```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine.  The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

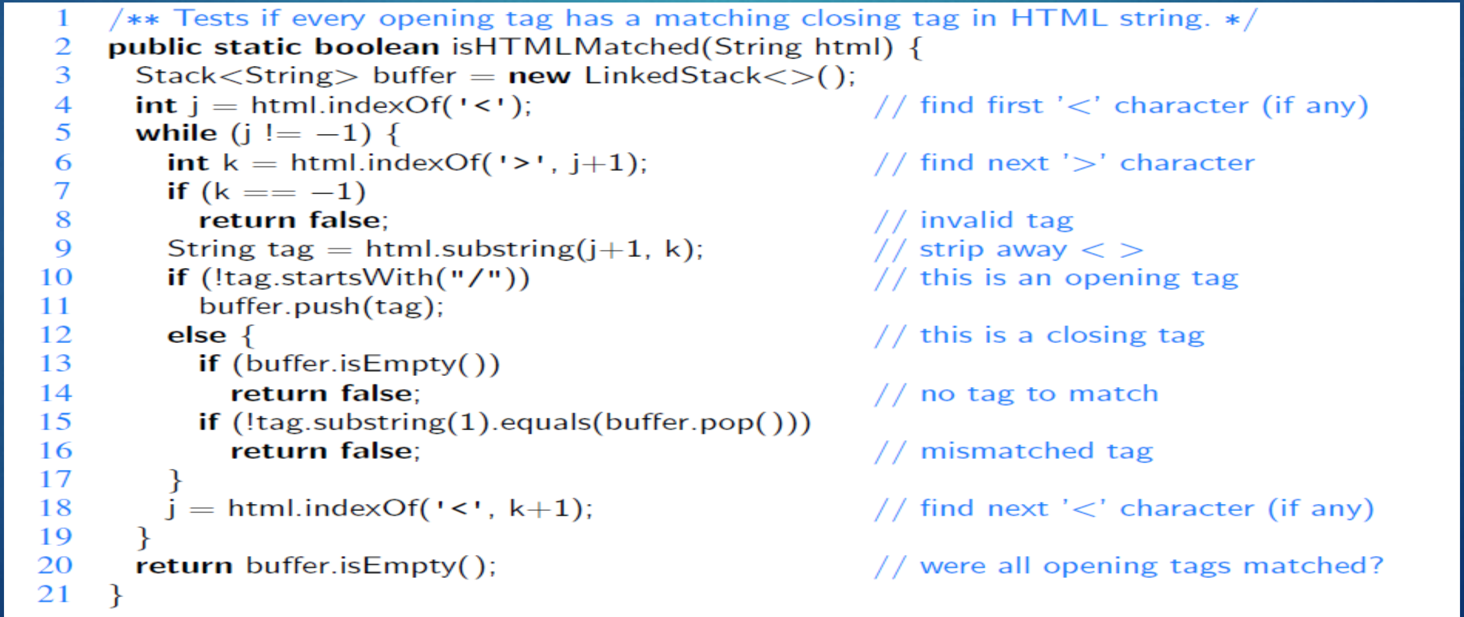
(b)

Figure 6.3: Illustrating (a) an HTML document and (b) its rendering.

Small exercises

3. Matching Tags in a Markup Language(pg 236)

- We make a left-to-right pass through the raw string, using index *j* to trackour progress.
- The indexOf method of the String class, which optionally accepts a starting index as a second parameter, locates the '<' and '>' characters that define the tags.
- Method substring, also of the String class, returns the substring starting at a given index and optionally ending right before another given index.



```
/** Tests if every opening tag has a matching closing tag in HTML string. */
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer = new LinkedStack<>();
  int j = html.indexOf('<');                          // find first '<' character (if any)
  while (j != -1) {
    int k = html.indexOf('>', j+1);                   // find next '>' character
    if (k == -1)
      return false;                                   // invalid tag
    String tag = html.substring(j+1, k);              // strip away < >
    if (!tag.startsWith("/"))                         // this is an opening tag
      buffer.push(tag);
    else {                                            // this is a closing tag
      if (buffer.isEmpty())
        return false;                                 // no tag to match
      if (!tag.substring(1).equals(buffer.pop()))
        return false;                                 // mismatched tag
    }
    j = html.indexOf('<', k+1);                       // find next '<' character (if any)
  }
  return buffer.isEmpty();                            // were all opening tags matched?
}
```

Code Fragment 6.8: Method for testing if an HTML document has matching tags.

Queue

HÀNG ĐỢI

Queues

- A *queue* is a pile in which items are added an one end and removed from the other. In this respect, a queue is like the line of customers waiting to be served by a bank teller. As customers arrive, they join the end of the queue while the teller serves the customer at the head of the queue. As a result, a *queue* is used when a sequence of activities must be done on a *first-come, first-served* basis.
- a queue is a *first-in, first-out* or *FIFO* data structure.

Queues (cont)

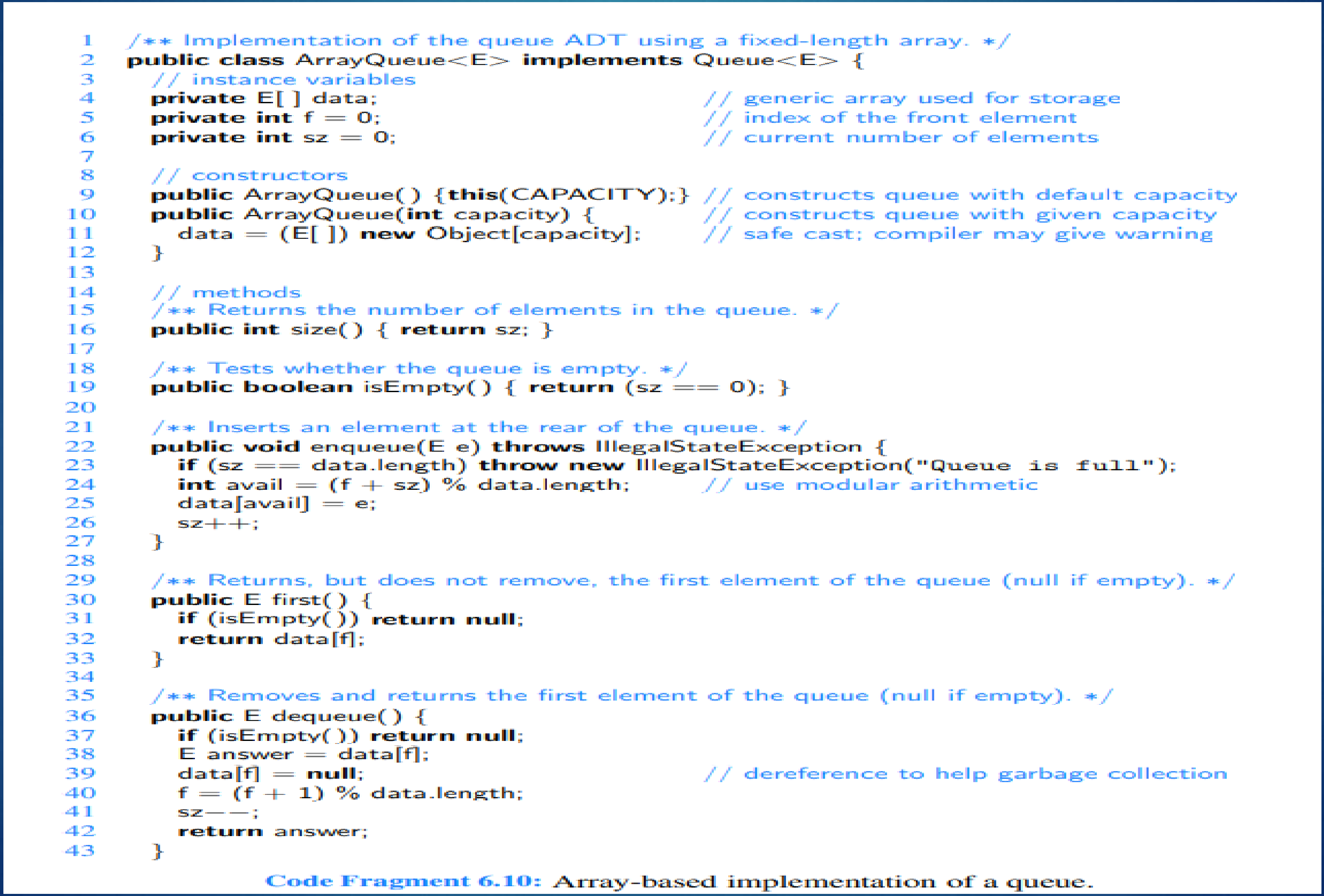
- enqueue(*e*): Adds element *e* to the back of queue.
- dequeue( ): Removes and returns the first element from the queue (or null if the queue is empty).
- first( ): Returns the first element of the queue, without removing it

(or null if the queue is empty).

- size( ): Returns the number of elements in the queue.
- isEmpty( ): Returns a boolean indicating whether the queue is empty.

Applications of Queue

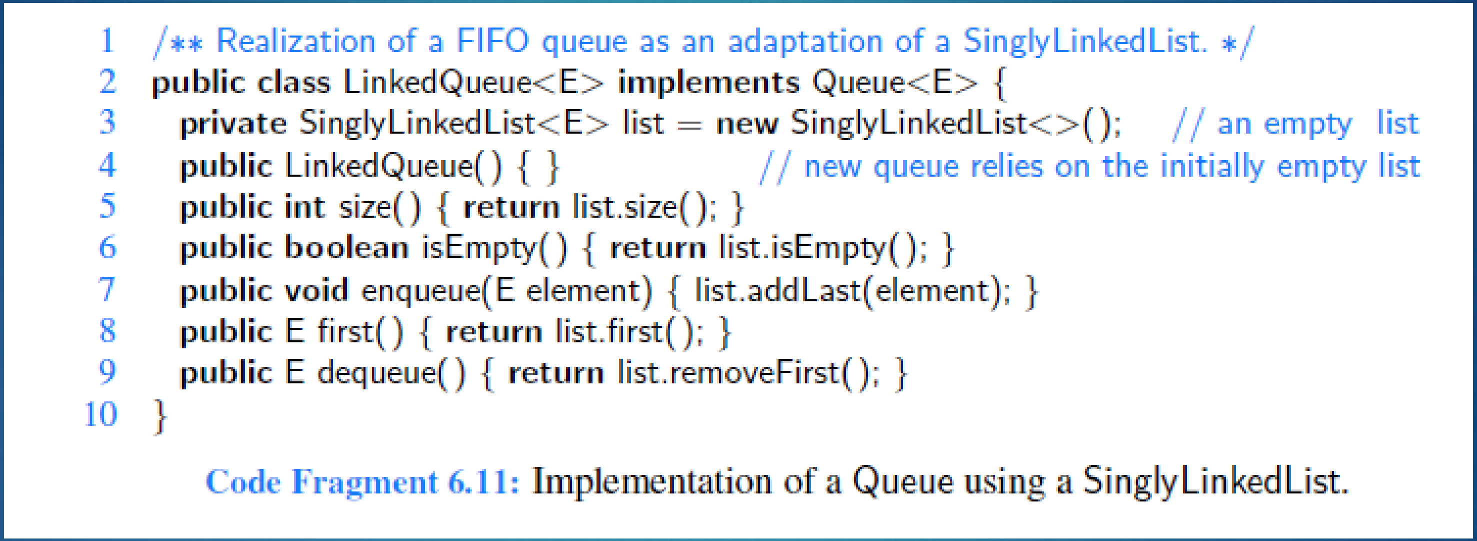
Implementing a Queue with an Array (pg 261)



```
/** Implementation of the queue ADT using a fixed-length array. */
public class ArrayQueue<E> implements Queue<E> {
  // instance variables
  private E[ ] data;                          // generic array used for storage
  private int f = 0;                          // index of the front element
  private int sz = 0;                         // current number of elements

  // constructors
  public ArrayQueue( ) {this(CAPACITY);}  // constructs queue with default capacity
  public ArrayQueue(int capacity) {          // constructs queue with given capacity
    data = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
  }

  // methods
  /** Returns the number of elements in the queue. */
  public int size( ) { return sz; }

  /** Tests whether the queue is empty. */
  public boolean isEmpty( ) { return (sz == 0); }

  /** Inserts an element at the rear of the queue. */
  public void enqueue(E e) throws IllegalStateException {
    if (sz == data.length) throw new IllegalStateException("Queue is full");
    int avail = (f + sz) % data.length;      // use modular arithmetic
    data[avail] = e;
    sz++;
  }

  /** Returns, but does not remove, the first element of the queue (null if empty). */
  public E first( ) {
    if (isEmpty( )) return null;
    return data[f];
  }

  /** Removes and returns the first element of the queue (null if empty). */
  public E dequeue( ) {
    if (isEmpty( )) return null;
    E answer = data[f];
    data[f] = null;                           // dereference to help garbage collection
    f = (f + 1) % data.length;
    sz--;
    return answer;
  }
```

Code Fragment 6.10: Array-based implementation of a queue.

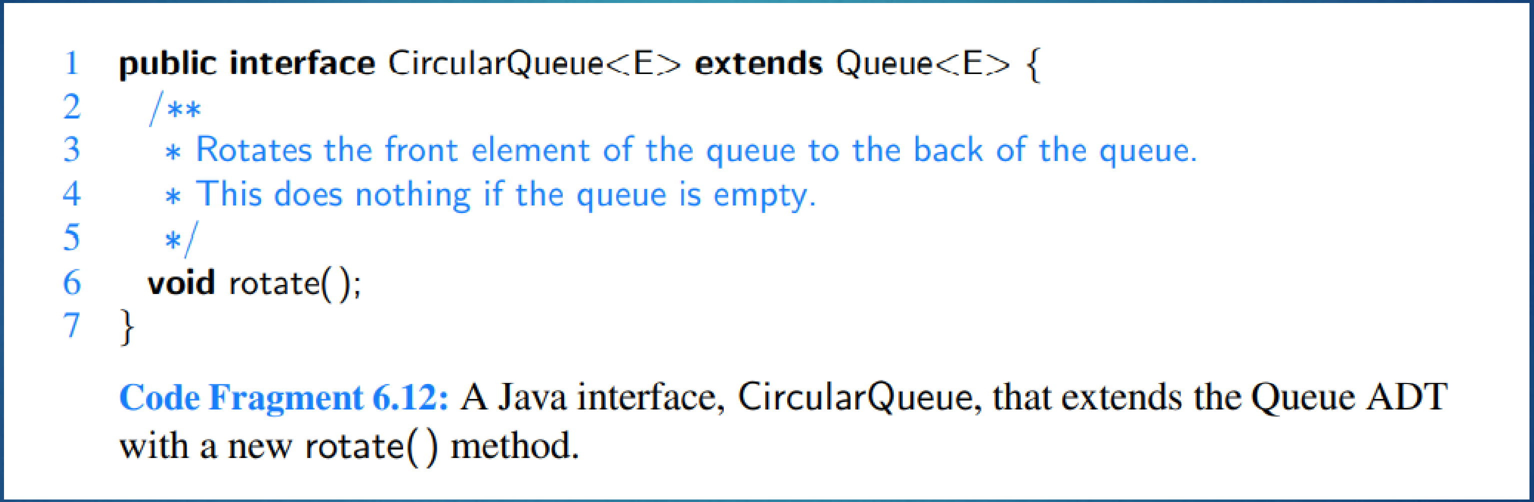
Implementing a Queue with a Singly Linked List (pg 263)



```
/** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedQueue() { }                  // new queue relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
}
```

Code Fragment 6.11: Implementation of a Queue using a SinglyLinkedList.

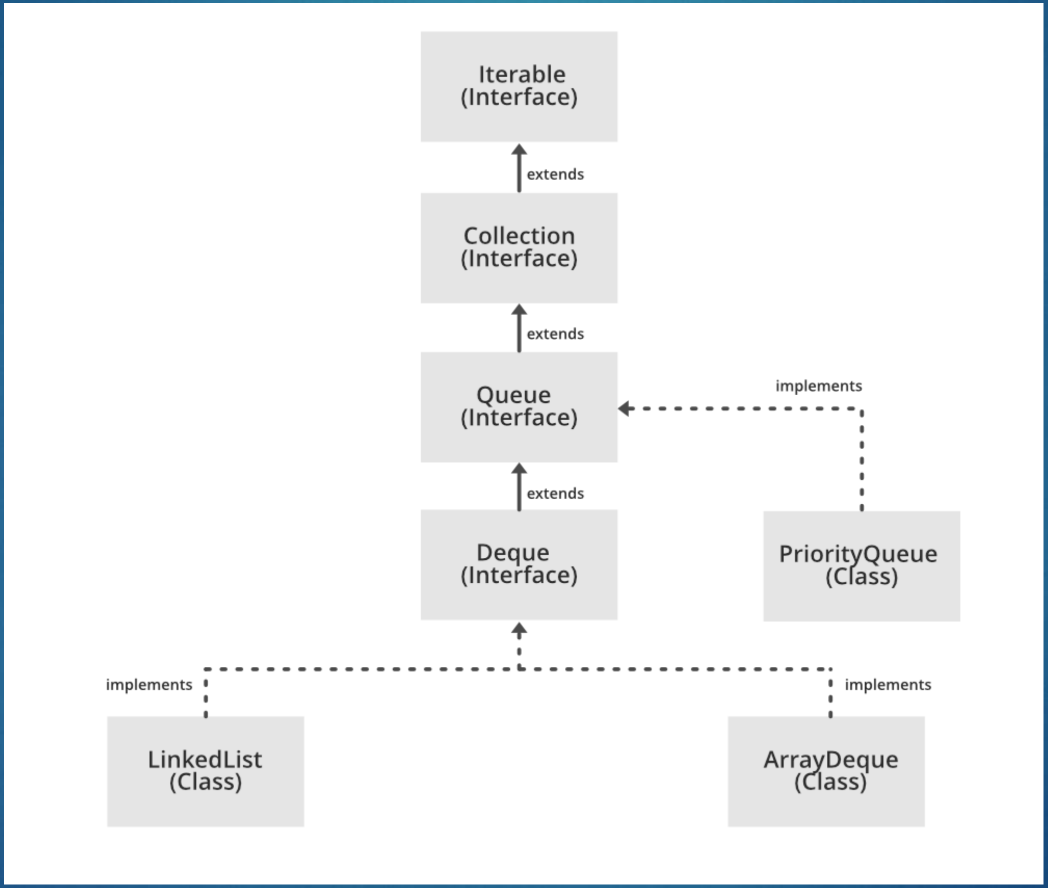
Implementing a Queue with a Circularly Linked List (pg 264)



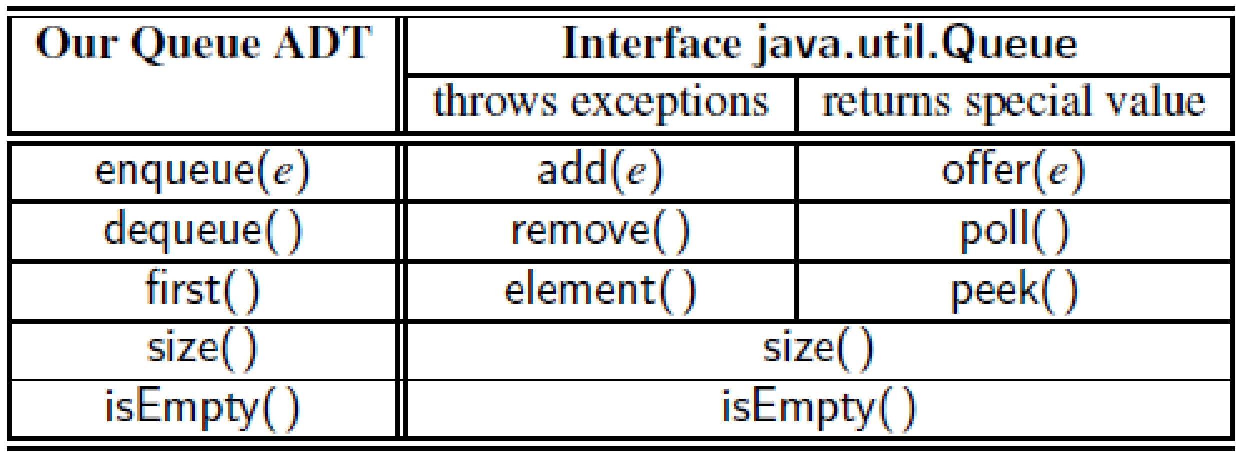
```
public interface CircularQueue<E> extends Queue<E> {
  /**
   * Rotates the front element of the queue to the back of the queue.
   * This does nothing if the queue is empty.
   */
  void rotate( );
}
```

Code Fragment 6.12: A Java interface, CircularQueue, that extends the Queue ADT with a new rotate( ) method.

Queue in Java



The java.util.Queue Interface in Java



Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Table 6.3: Methods of the queue ADT and corresponding methods of the interface java.util.Queue, when supporting the FIFO principle.

Using Queue in Java

```
Queue<Integer> q = new LinkedList<>();
    q.offer(9);
    q.offer(1);
    q.add(3);
    System.out.println(q);
    q.peek();
    System.out.println(q);
    q.remove();
    System.out.println(q);
    q.poll();
    System.out.println(q);
```

Double-Ended Queues

DEQUE

READING BOOK PG 266