Chapter 3: The Stack ADT

Section 3.1: Stacks

True / False

1. True or False? In a non-empty stack, the item that has been in the stack the longest is at the bottom of the stack.

Answer: True

2. True or False? The first element to be stored in a stack is also the first element removed from the stack.

Answer: False

3. True or False? A stack is a first in, first out structure.

Answer: False

4. True or False? A stack is a last in, first out structure.

Answer: True

5. True or False? For a non-empty stack, top returns the value of the most recent item pushed onto the stack.

Answer: True

6. True or False? For a non-empty stack, top returns the value of the most recent item popped from the stack.

Answer: False

Multiple Choice

7. The following sequence of operations essentially leaves a stack unchanged.

A. pop followed by push
B. pop followed by top
C. push followed by pop
D. push followed by top
E. top followed by push

Fill-in-the-Blank

8. The operation that adds an element to a stack is called _____ .

Answer: push

9. The operation that removes the top element of the stack is called _____ .

Answer: pop

10. The operation that returns the top element of the stack is called _____ .

Answer: top

Short Answer

11. Define a stack.

Answer: A stack is a structure in which elements are added and removed at only one end; a "last in, first out" structure.

12. Explain the difference between the "classic" definition of the pop operation on a stack and the "modern" definition we use in the textbook.

Answer: Classically the pop operation both removes and returns the element at the top of the stack. With the textbook's modern approach the pop operation only removes the element at the top of the stack.

Section 3.2: Collection Elements

True / False

13. True or False? The textbook defines stacks, queues, and graphs using generics.

Answer: True

14. True or False? The textbook defines stacks that hold elements of class *Object*.

Answer: False

Fill-in-the-Blank

15. An object whose purpose is to hold other objects is called a _____ object.

Answer: Collection

16. All other Java classes ultimately inherit from the _____ class.

Answer: *Object*

Short Answer

17. What is a collection object?

Answer: An object whose purpose is to hold other objects.

18. What are the standard operations we are interested in performing on collect objects?

Answer: Inserting and removing objects and iterating through the objects in the collection.

19. Suppose a collection object is defined to hold elements of class *Object*, and you use it to store *String* objects. Describe what you must do when you retrieve an object from the collection and intend to use it as a *String*.

Answer: You must *cast* the object into a *String*.

Section 3.3: Exceptional Situations

True / False

20. True or False? Thrown exceptions are always fatal (i.e. the program will stop running).

Answer: False

21. True or False? When handling an exception explicitly, it is best to do so at the highest level of abstraction possible.

Answer: False

22. True or False? When handling an exception explicitly, it is best to do so at the lowest level of abstraction possible.

Answer: True

23. True or False? It is usually better for a program to “bomb” then to produce erroneous results.

Answer: True

24. True or False? Java programmers can define their own exceptions.

Answer: True

25. True or False? In Java, exceptions that are not handled explicitly in the program are thrown out to the interpreter,

Answer: True

26. True or False? A method should test that its preconditions are met and if not, throw an exception.

Answer: False

Multiple Choice

27. This Java statement is used to “announce” that an exception has occurred.

A. throw
B. raise
C. catch
D. try

E. None of these

Answer: B

Fill-in-the-Blank

28. Java programmers define their own exceptions by _____ the library's *Exception* class.

Answer: extending

29. When defining an exception a common approach is to create constructors that just call the constructors of the _____ .

Answer: superclass or *Exception* class

30. An exception of the _____ class is treated uniquely by the Java environment in that it does not have to be explicitly handled by the method within which it might be raised.

Answer: java.lang.RunTimeException

Short Answer

31. What are the three major parts (actions that can be taken) of the Java exception mechanism?

Answer: Exceptions can be defined, raised or generated, and handled.

32. What are the three options we have for dealing with error situations within our ADT methods?

Answer: We can detect and handle the error within the method itself. We can detect the error within the method and throw an exception related to the error. We can ignore the error situation.

Section 3.4: Formal Specification

True / False

33. True or False? Because *StackUnderflowException* is an unchecked exception, if it is raised and not caught, it is eventually thrown out to the run-time environment.

Answer: True

34. True or False? Because *StackUnderflowException* is an unchecked exception, if it is raised and not caught, it is just ignored by the system.

Answer: False

35. True or False? Our *UnboundedStackInterface* extends our *BoundedStackInterface*.

Answer: False

36. True or False? The *StackOverflowException* is an unchecked exception.

Answer: True

37. True or False? In Java, an interface can extend another interface.

Answer: True

38. True or False? In Java, an interface cannot extend another interface.

Answer: False

Multiple Choice

39. The *StackUnderflowException* could be thrown by the following stack method(s).

A. push and pop
B. isEmpty
C. the constructors
D. push and top
E. pop and top

Answer: E

40. The *StackOverFlow* exception could be thrown by the following stack method(s).

A. push
B. isFull
C. the constructors
D. top
E. pop and top

Answer: A

41. In Java, an interface can extend

A. A class
B. An object
C. An array
D. Another interface
E. None of these

42. Our BoundedStackInterface extends our

A. UnboundedStackInterface
B. StackInterface
C. StackOverflowException
D. StackUnderflowException
E. None of these

Answer: B

Fill-in-the-Blank

43. The Java _____ construct is used to create a formal specification of our Stack ADT.

Answer: interface

44. An application programmer can prevent accessing an empty stack by using the _____ method in an if statement to prevent such access.

Answer: isEmpty

45. The *StackUnderflowException* class extends the Java _____ class.

Answer: *RuntimeException*

Short Answer

46. What operation is required by our *BoundedStackInterface* but not by our *UnboundedStackInterface*?

Answer: The *BoundedStackInterface* requires an *isFull* operation, while the *UnboundedStackInterface* does not.

47. Describe the two ways an application programmer can address the exceptional situation of attempting to pop something from an empty stack.

Answer: They can use the *isEmpty* operation to test if a stack is empty and if so, avoid using the pop operation. Or they can surround the pop operation with a try-catch statement that catches and handles the *StackUnderflowException*.

48. What are the three interfaces we defined related to our Stack ADT?

Answer: The *StackInterface*, the *UnboundedStackInterface*, and the *BoundedStackInterface*.

49. What is the difference between the *push* method defined in our *UnboundedStackInterface* and the *push* method defined in our *BoundedStackInterface*?

Answer: The one in the *BoundedStackInterface* is defined as potentially throwing a *StackOverfowException* while the one in the *UnboundedStackInterface* is not.

50. Show what is written by the following segment of code, given that item1, item2, and item3 are int variables, and stack is an object that fits our abstract description of a stack. Assume that you can store and retrieve variables of type int on stack.

```
item1 = 1;
item2 = 0;
item3 = 4;
stack.push(item2);
stack.push(item1);
stack.push(item1 + item3);
item2 = stack.top();
stack.push (item3*item3);
stack.push(item2);
stack.push(3);
item1 = stack.top();
stack.pop();
System.out.println(item1 + " " + item2 + " " + item3);
while (!stack.isEmpty())
{
  item1 = stack.top();
  stack.pop();
```

```
  System.out.println(item1);
}
```

Answer:

```
3 5 4
5
16
5
1
0
```

Section 3.5: Array-Based Implementation

True / False

51. True or False? Our *ArrayStack* class implements a bounded stack.

Answer: True

52. True or False? Our *ArrayListStack* class implements a bounded stack.

Answer: False

53. True or False? When an object of class *ArrayStack* represents an empty stack, its *topIndex* variable is 0.

Answer: False

Short Answer

54. Complete the following table by entering the Big-O execution time of the indicated operations when using the *ArrayStack* class, assuming a maximum stack size of N.

method	efficiency
the constructor	_____
push	_____
pop	_____
top	_____
isFull	_____
isEmpty	_____

Answer:

method	efficiency
the constructor	O(N)
push	O(1)
pop	O(1)
top	O(1)
isFull	O(1)
isEmpty	O(1)

Questions 55 to 59 assume the student has access to the beginning of the *ArrayStack* class, as developed in the textbook on page 188:

```
//-----------------------------------------------------------------
// ArrayStack.java       by Dale/Joyce/Weems             Chapter 3
//
// Implements BoundedStackInterface using an array to hold the
// 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 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

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

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

55. Complete the implementation of the *isEmpty* method:

```
public boolean isEmpty()
// Returns true if this stack is empty, otherwise returns false.
{
  // complete the method body
}
```

Answer:

```
  if (topIndex == -1)
    return true;
  else
     return false;
```

56. Complete the implementation of the *isFull* method:

```
public boolean isFull()
// Returns true if this stack is full, otherwise returns false.
{
  // complete the method body
}
```

Answer:

```
  if (topIndex == (stack.length - 1))
    return true;
  else
    return false;
```

57. Complete the implementation of the *push* method:

```
public void push(T element)
// Throws StackOverflowException if this stack is full,
// otherwise places element at the top of this stack.
{
  // complete the method body
}
```

Answer:

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

58. Complete the implementation of the *pop* method:

```
public void pop()
// Throws StackUnderflowException if this stack is empty,
// otherwise removes top element from this stack.
{
  // complete the method body
}
```

Answer:

```
  if (!isEmpty())
  {
    stack[topIndex] = null;
    topIndex--;
  }
  else
```

```
    throw new StackUnderflowException("Pop attempted on an empty stack.");
```

59. Complete the implementation of the *top* method:

```
public T top()
// Throws StackUnderflowException if this stack is empty,
// otherwise returns top element from this stack.
{
  // complete the method body
}
```

Answer:

```
  T topOfStack = null;
  if (!isEmpty())
    topOfStack = stack[topIndex];
  else
    throw new StackUnderflowException("Top attempted on an empty
stack.");
  return topOfStack;
```

Section 3.6: Well-Formed Expressions

In this section of the textbook we define and build an application that uses the Stack ADT. This section is best used as fodder for student experimentation and projects. We do not provide test questions for this section – the best questions would be related to any specific programming project assigned to the students.

Section 3.7: Link-Based Implementation

True / False

60. True or False? The *LLNode* class is a self-referential class.

Answer: True

61. True or False? Our *LinkedStack* class implements a bounded stack.

Answer: False

62. True or False? Our *LinkedStack* class implements an unbounded stack.

Answer: True

63. True or False? When an object of class *LinkedStack* represents an empty stack, its *top* variable is 0.

Answer: False

64. True or False? When an object of class *LinkedStack* represents an empty stack, its *top* variable is *null*.

Answer: True

Short Answer

65. Complete the following table by entering the Big-O execution time of the indicated operations when using the *LinkedStack* class, assuming a maximum stack size of N.

method	efficiency
the constructor	_____
push	_____
pop	_____
top	_____
isEmpty	_____

Answer:

method	efficiency
the constructor	O(1)
push	O(1)
pop	O(1)

top	O(1)
isEmpty	O(1)

Questions 66 to 69 assume the student has access to the beginning of the *LinkedStack* class, as developed in the textbook on page 205:

```
//-----------------------------------------------------------------
// LinkedStack.java      by Dale/Joyce/Weems             Chapter 3
//
// Implements UnboundedStackInterface using a linked list
// to hold the stack elements.
//-----------------------------------------------------------------

package ch03.stacks;

import support.LLNode;

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

  public LinkedStack()
  {
    top = null;
  }
```

66. Complete the implementation of the *isEmpty* method:

```
public boolean isEmpty()
// Returns true if this stack is empty, otherwise returns false.
{
  // complete the method body
}
```

Answer:

```
if (top == null)
    return true;
  else
    return false;
}
```

or

```
return (top == null);
```

67. Complete the implementation of the *push* method:

```
public void push(T element)
// Places element at the top of this stack.
{
  // complete the method body
}
```

Answer:

```
LLNode<T> newNode = new LLNode<T>(element);
newNode.setLink(top);
top = newNode;
```

68. Complete the implementation of the *pop* method:

```
public void pop()
// Throws StackUnderflowException if this stack is empty,
// otherwise removes top element from this stack.
{
  // complete the method body
}
```

Answer:

```
if (!isEmpty())
  {
    top = top.getLink();
  }
  else
    throw new StackUnderflowException("Pop attempted on an empty stack.");
```

69. Complete the implementation of the *top* method:

```
public T top()
// Throws StackUnderflowException if this stack is empty,
// otherwise returns top element from this stack.
{
  // complete the method body
}
```

Answer:

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