KEY CONCEPTS

Pseudocode for the ADT Stack Operations

```
// StackItemType is the type of the items stored in the stack.

+createStack()
// Creates an empty stack.

+isEmpty():boolean {query}
// Determines whether a stack is empty.

+push(in newItem:StackItemType) throws StackException
// Adds newItem to the top of the stack. Throws
// StackException if the insertion is not successful.

+pop():StackItemType throws StackException
// Retrieves and then removes the top of the stack (the
// item that was added most recently). Throws
// StackException if the deletion is not successful.

+popAll()
// Removes all items from the stack.

+peek():StackItemType {query} throws StackException
// Retrieves the top of the stack. That is, peek
// retrieves the item that was added most recently.
// Retrieval does not change the stack. Throws
// StackException if the retrieval is not successful.
```

7.3 Implementations of the ADT Stack

This section develops three Java implementations of the ADT stack. The first implementation uses an array to represent the stack, the second uses a linked list, and the third uses the ADT list. Figure 7-4 illustrates these three implementations. The following interface `StackInterface` is used to provide a common specification for the three implementations.

```java
public interface StackInterface {
  public boolean isEmpty();
  // Determines whether the stack is empty.
  // Precondition: None.
  // Postcondition: Returns true if the stack is empty;
  // otherwise returns false.

  public void popAll();
  // Removes all the items from the stack.
  // Precondition: None.
  // Postcondition: Stack is empty.

  public void push(Object newItem) throws StackException;
  // Adds an item to the top of a stack.
  // Precondition: newItem is the item to be added.
  // Postcondition: If insertion is successful, newItem
  // is on the top of the stack.
  // Exception: Some implementations may throw
  // StackException when newItem cannot be placed on
  // the stack.

  public Object pop() throws StackException;
  // Removes the top of a stack.
  // Precondition: None.
  // Postcondition: If the stack is not empty, the item
  // that was added most recently is removed from the
  // stack and returned.
  // Exception: Throws StackException if the stack is
  // empty.

  public Object peek() throws StackException;
  // Retrieves the top of a stack.
  // Precondition: None.
  // Postcondition: If the stack is not empty, the item
  // that was added most recently is returned. The
  // stack is unchanged.
  // Exception: Throws StackException if the stack is
  // empty.
}  // end StackInterface
```

```
public class StackException
             extends java.lang.RuntimeException {
  public StackException(String s) {
    super(s);
  }  // end constructor
}  // end StackException
```

Note that *StackException* extends *java.lang.RuntimeException*, so that the calls to methods that throw *StackException* do not have to be enclosed in *try* blocks. This is a reasonable choice for the operations *pop* and *peek*, since you can avoid the exception by checking to see whether the stack is empty before calling these operations. But *push* can also throw *StackException* in an array-based implementation when a fixed-size array is used and becomes full. You can avoid this exception in an array-based implementation by providing a method *isFull* that determines whether the stack is full; you call *isFull* before you call *push*.

In a reference-based implementation, this *isFull* method would not be necessary. Also, although the *push* method throws *StackException* in the interface specification, the *throws* clause could be omitted in a reference-based implementation of *push*.

An Array-Based Implementation of the ADT Stack

Figure 7-5 suggests that you use an array of *Objects* called *items* to represent the items in a stack and an index *top* such that *items[top]* is the stack's top. We want to define a class whose instances are stacks and whose private data fields are *items* and *top*.

The following class is an array-based implementation of the ADT stack. The default constructor for this class corresponds to and replaces the ADT operation *createStack*. Note that the preconditions and postconditions given earlier in *StackInterface* apply here as well, and so are omitted to save space.

```
public class StackArrayBased implements StackInterface {
  final int MAX_STACK = 50;  // maximum size of stack
  private Object items[];
  private int top;
```

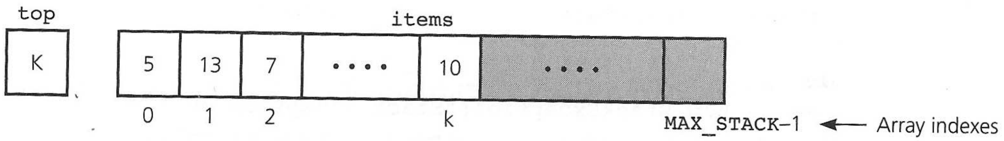

FIGURE 7-5

An array-based implementation

```
public StackArrayBased() {
  items = new Object[MAX_STACK];
  top = -1;
}  // end default constructor

public boolean isEmpty() {
  return top < 0;
}  // end isEmpty

public boolean isFull() {
  return top == MAX_STACK-1;
}  // end isFull

public void push(Object newItem) throws StackException {
  if (!isFull()) {
    items[++top] = newItem;
  }
  else {
    throw new StackException("StackException on " +
                             "push: stack full");
  }  // end if
}  // end push

public void popAll() {
  items = new Object[MAX_STACK];
  top = -1;
}  // end popAll

public Object pop() throws StackException {
  if (!isEmpty()) {
    return items[top--];
  }
  else {
    throw new StackException("StackException on " +
                             "pop: stack empty");
  }  // end if
}  // end pop

public Object peek() throws StackException {
  if (!isEmpty()) {
    return items[top];
  }
  else {
    throw new StackException("Stack exception on " +
                             "peek - stack empty");
  }  // end if
  }  // end peek
}  // end StackArrayBased
```

A program that uses a stack could begin as follows:

```
public class StackTest {
  public static final int MAX_ITEMS = 15;

  public static void main(String[] args) {
    StackArrayBased stack = new StackArrayBased();
    Integer items[] = new Integer[MAX_ITEMS];
    for (int i=0; i<MAX_ITEMS; i++) {
      items[i] = new Integer(i);
      if (!stack.isFull()) {
        stack.push(items[i]);
      }  // end if
    }  // end for
    while (!stack.isEmpty()) {
      // cast result of pop to Integer
      System.out.println((Integer)(stack.pop()));
    }  // end while
    ...
```

By implementing the stack as a class, and by declaring *items* and *top* as private, you ensure that the client cannot violate the ADT's walls. If you did not hide your implementation within a class, or if you made the array *items* public, the client could access the elements in *items* directly instead of by using the operations of the ADT stack. Thus, the client could access any elements in the stack, not just its top element. You might find this capability attractive, but in fact it violates the specifications of the ADT stack. If you truly need to access all the items of your ADT randomly, do not use a stack!

Private data fields are hidden from the client

Again, note that *StackException* provides a simple way for the implementer to indicate to the stack's client unusual circumstances, such as an attempted insertion into a full stack or a deletion from an empty stack.

StackException provides a simple way to indicate unusual events

Finally, note that instances of *StackArrayBased* cannot contain items of a primitive type such as *int*, because *int* is not derived from *Object*. If you need a stack of integers, for example, you will have to use the corresponding wrapper class, which in this case is *Integer*. Finally, *pop* and *peek* return an item that is an instance of *Object*. You must cast this item back to the subtype of *Object* that you pushed onto the stack. Otherwise, methods available for the subtype will not be accessible.

A Reference-Based Implementation of the ADT Stack

Many applications require a reference-based implementation of a stack so that the stack can grow and shrink dynamically. Figure 7-6 illustrates a reference-based

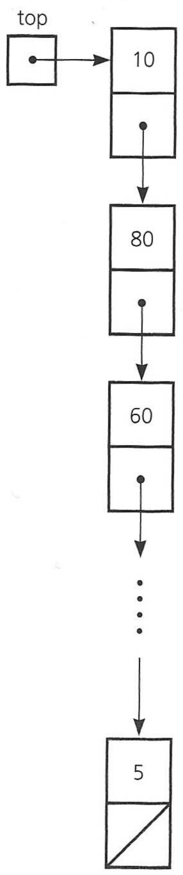

FIGURE 7-6

A reference-based implementation

implementation of a stack where *top* is a reference to the head of a linked list of items. The implementation uses the same node class developed for the linked list in Chapter 5.

Note that the preconditions and postconditions given earlier in *Stack-Interface* apply here as well, and so are omitted to save space.

```
public class StackReferenceBased
                implements StackInterface {
  private Node top;

  public StackReferenceBased() {
    top = null;
  }  // end default constructor

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

  public void push(Object newItem) {
    top = new Node(newItem, top);
  }  // end push

  public Object pop() throws StackException {
    if (!isEmpty()) {
      Node temp = top;
      top = top.next;
      return temp.item;
    }
    else {
      throw new StackException("StackException on " +
                               "pop: stack empty");
    }  // end if
  }  // end pop

  public void popAll() {
    top = null;
  }  // end popAll

  public Object peek() throws StackException {
    if (!isEmpty()) {
      return top.item;
    }
    else {
      throw new StackException("StackException on " +
                               "peek: stack empty");
    }  // end if
  }  // end peek
}  // end StackReferenceBased
```