

Additional methods:

top(): Return the top object on the stack, without removing it.
Input: None; Output: Object.
 an error occurs if the stack is empty.

isEmtpy(): Return a boolean indicating if the stack is empty.
Input: None; Output: Boolean.

size(): Return the number of objects in the stack.
Input: None; Output: Integer.

Additionally, let us also define the following supporting methods:

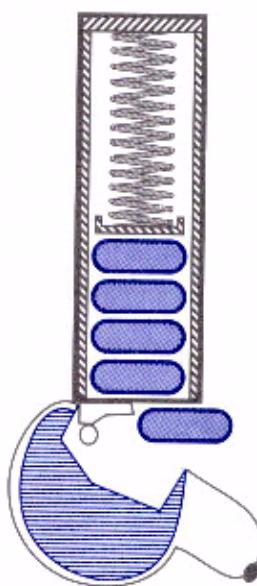
pop(): Remove from the stack and return the top object on the stack; an error occurs if the stack is empty.
Input: None; Output: Object.

push(o): Insert object o at the top of the stack.
Input: Object; Output: None.

A stack S is an abstract data type (ADT) that supports the following two fundamental methods:

3.1.1 The Stack Abstract Data Type

Figure 3.1: A schematic drawing of a PEZ® dispenser, a physical implementation of the stack ADT. (PEZ® is a registered trademark of PEZ Candy, Inc.)



Example 3.3: The following table shows a series of stack operations and their effects on an initially empty stack S of integer objects. For simplicity, we show integers as arguments of the operations.

Operation	Output	S
push(5)	-	(5)
push(3)	-	(5, 3)
pop()	3	(5)
push(7)	-	(5, 7)
pop()	7	(5)
top()	5	(5)
pop()	5	()
pop()	"error"	()
isEmpty()	true	()
push(9)	-	(9)
push(7)	-	(9, 7)
push(3)	-	(9, 7, 3)
push(5)	-	(9, 7, 3, 5)
size()	4	(9, 7, 3, 5)
pop()	5	(9, 7, 3)

A Stack Interface in Java

Because of its importance, the stack data structure is included as a “built-in” class in the `java.util` package of Java. Class `java.util.Stack` is a data structure that stores generic Java objects and includes, among others, the methods `push(obj)`, `pop()`, `peek()` (equivalent to `top()`), `size()`, and `empty()` (equivalent to `isEmpty()`). Methods `pop()` and `peek()` throw exception `StackEmptyException` if they are called on an empty stack. In Section 3.5.3, we show an example of a `Stack` object. While it is convenient to just use the built-in class `java.util.Stack`, it is instructive to learn how to design and implement a stack “from scratch.”

Implementing an abstract data type in Java involves two steps. The first step is the definition of a Java *Application Programming Interface* (API), or simply *interface*, which describes the names of the methods that the ADT supports and how they are to be declared and used. A Java interface for the stack ADT is given in Code Fragment 3.1. Note that this interface is very general since it specifies that objects of arbitrary and possibly heterogeneous classes can be inserted into the stack.

The error condition that occurs when calling method `pop()` or `top()` on an empty stack is signaled by throwing an exception of type `StackEmptyException`, which is defined in Code Fragment 3.2.

stack is stored in the cell $S[t]$.

Figure 3.2: Realization of a stack by means of an array S . The top element in the



(See Figure 3.2.)

array S plus an integer variable t that gives the index of the top element in array S . For our stack, say, $N = 1,000$ elements. Our stack then consists of an N -element array. Since an array's size needs to be determined when it is created, one of the important details of our implementation is that we specify some maximum size N in this subsection, we show how to realize a stack by storing its elements in an array.

3.1.2 A Simple Array-Based Implementation

Implementation of the Stack interface in the following subsection. For a given ADT to be of any use, we need to provide a concrete class that implements the methods of the interface associated with that ADT. We give a simple implementation of the Stack interface in the following subsection.

Code Fragment 3.1: Interface Stack.

Code Fragment 3.2: Exception thrown by methods pop() and top() of the Stack interface when called on an empty stack.

```
public class StackEmptyException extends RuntimeException {  
    public StackEmptyException(String err) {  
        super(err);  
    }  
}
```

Code Fragment 3.1: Interface Stack.

```
public interface Stack {  
    // accessor methods  
    public int size(); // return the number of elements stored in the stack  
    public boolean isEmpty(); // test whether the stack is empty  
    public Object top(); // return the top element  
    public void push(Object element); // insert an element onto the stack  
    public Object pop(); // remove the top element  
    throws StackEmptyException; // thrown if called on an empty stack  
    // update methods  
    throws StackEmptyException; // thrown if called on an empty stack  
    public void clear(); // remove all elements from the stack  
    public void swap(int i, int j); // swap the elements at positions i and j  
    // iterator methods  
    public Iterator iterator(); // return an iterator over the stack  
    throws StackEmptyException; // thrown if called on an empty stack  
}
```

Recalling that arrays start at index 0 in Java, we initialize t to -1 . We also introduce a new type of exception, called `StackFullException`, to signal the error condition that arises if we try to insert a new element and the array S is full. We can then implement the stack ADT methods as described in Code Fragment 3.3.

```
Algorithm size():
    return t + 1

Algorithm isEmpty():
    return (t < 0)

Algorithm top():
    if isEmpty() then
        throw a StackEmptyException
    return S[t]

Algorithm push(o):
    if size() = N then
        throw a StackFullException
    t ← t + 1
    S[t] ← o

Algorithm pop():
    if isEmpty() then
        throw a StackEmptyException
    e ← S[t].
    S[t] ← null
    t ← t - 1
    return e
```

Code Fragment 3.3: Implementation of a stack by means of an array.

The correctness of these methods follows immediately from the definition of the methods themselves. There is, nevertheless, a mildly interesting point in this implementation involving the implementation of the `pop` method. Note that we could have avoided resetting the old $S[t]$ to `null` and we would still have a correct method. There is a trade-off in being able to avoid this assignment in Java, however. The trade-off involves the Java *garbage collection* mechanism that searches memory for objects that are no longer referenced by active objects, and reclaims their space for future use. (For more details, see Section 9.4.5.) Let $e = S[t]$ be the top element before the `pop` method is called. By making $S[t]$ a null reference, we

The array implementation of a stack is both simple and efficient, and is widely used in a variety of computing applications. Nevertheless, this implementation has one negative aspect: it must assume a fixed upper bound N on the ultimate size of the stack. In Code Fragment 3.4 we choose the capacity value $N = 1,000$ more-or-less arbitrarily. An application may actually need much less space than this, in which case we would be wasting memory. Alternatively, an application may need more space than this, in which case our stack implementation may "crash" due to the application with an error as soon as it tries to push its $(N + 1)$ st object on the stack.

Thus, given with its simplicity and efficiency, the array-based stack implementation is not necessarily ideal. Fortunately, there are other stack implementations that do not have a size limitation and use space proportionally to the actual number of elements stored in the stack. In cases where we have a good estimate on the number of items needed to go in the stack, however, the array-based implementation is hard to beat. Stacks serve a vital role in a number of computing applications, so it is helpful to have a fast stack ADT implementation, such as the simple array-based implementation.

Table 3.1: Performance of a stack realized by an array. The space usage is $O(n)$.

Method	Time
pop	$O(1)$
push	$O(1)$
top	$O(1)$
isEmpty	$O(1)$
size	$O(1)$

Table 3.1 shows the running times for methods in a realization of a stack by an array. Each of the stack methods in the array realization executes a constant number of statements involving arithmetic operations, comparisons, and assignments. Thus, in this implementation of the Stack ADT, each method runs in constant time, that is, they each run in $O(1)$ time.

indicate that the stack no longer needs to hold a reference to object *e*. Indeed, it there are no other active references to *e*, then the memory space taken by *e* will be reclaimed by the Garbage collector. A concrete Java implementation of the above pseudo-code specification, by means of Java class `ArrayList` implements the Stack interface, as given in Code Fragment 3.4.

```

public class ArrayStack implements Stack {
    // Implementation of the Stack interface using an array.

    public static final int CAPACITY = 1000; // default capacity of the stack
                                                // maximum capacity of the stack.
    private int capacity;                    // S holds the elements of the stack
    private Object S[];                      // the top element of the stack.
    private int top = -1;

    public ArrayStack() {                   // Initialize the stack with default capacity
        this(CAPACITY);
    }

    public ArrayStack(int cap) {           // Initialize the stack with given capacity
        capacity = cap;
        S = new Object[capacity];
    }

    public int size() {                   // Return the current stack size
        return (top + 1);
    }

    public boolean isEmpty() {           // Return true iff the stack is empty
        return (top < 0);
    }

    public void push(Object obj) {        // Push a new object on the stack
        if (size() == capacity)
            throw new StackFullException("Stack overflow.");
        S[++top] = obj;
    }

    public Object top() {                // Return the top stack element
        throws StackEmptyException {
        if (isEmpty())
            throw new StackEmptyException("Stack is empty.");
        return S[top];
    }

    public Object pop() {                // Pop off the stack element
        throws StackEmptyException {
        Object elem;
        if (isEmpty())
            throw new StackEmptyException("Stack is Empty.");
        elem = S[top];
        S[top--] = null;               // Dereference S[top] and decrement top
        return elem;
    }
}

```

Code Fragment 3.4: Array-based Java implementation of the Stack interface.