© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

chapter 3

The Stack ADT

Goals

Knowledge Goals

You should be able to

- describe a stack and its operations at a logical level
- list three options for making a collection ADT generally usable
- explain three ways to "handle" exceptional situations when defining an ADT
- explain the difference between the formal definitions of bounded and unbounded stacks
- describe an algorithm for determining whether grouping symbols (such as parentheses) within a string are balanced using a stack
- describe algorithms for implementing stack operations using an array
- describe algorithms for implementing stack operations using an `ArrayList`
- describe algorithms for implementing stack operations using a linked list
- use Big-O analysis to describe and compare the efficiency of algorithms for implementing stack operations using various data structuring mechanisms
- define inheritance of interfaces and multiple inheritance of interfaces
- describe an algorithm for evaluating postfix expressions, using a stack

Skill Goals

You should be able to

- use the Java generics mechanism when designing/implementing a collections ADT
- implement the Stack ADT using an array
- implement the Stack ADT using the Java library's `ArrayList` class
- implement the Stack ADT using a linked list
- draw diagrams showing the effect of stack operations for a particular implementation of a stack
- create a Java exception class
- throw Java exceptions from within an ADT and catch them within an application that uses the ADT
- use a Stack ADT as a component of an application
- evaluate a postfix expression "by hand"

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

In this chapter we investigate the stack, an important data structure. As we described in Chapter 1, a stack is a "last in, first out" structure. We study the stack as an ADT, looking at it from the logical, application, and implementation levels. At the logical level we formally define our Stack ADT using a Java `interface`. We discuss many applications of stacks and look in particular at how stacks are used to determine whether a set of grouping symbols is well formed and to support evaluation of mathematical expressions. We investigate the implementation of stacks using our two basic approaches: arrays and linked lists. We also investigate an approach using the Java library's `ArrayList` class.

This chapter will also expand your understanding of ADTs and your practical knowledge of the Java language. Early in the chapter we look at ways to make an ADT generally usable and options for addressing exceptional situations. Java topics in this chapter include a closer look at the exception mechanism and the introduction of generics and the inheritance of interfaces.

3.1 Stacks

Consider the items pictured in Figure 3.1. Although the objects are all different, each illustrates a common concept—the stack. At the logical level, a stack is an ordered group of homogeneous elements. The removal of existing elements and the addition of new ones can take place only at the top of the stack. For instance, if your favorite blue shirt is underneath a faded, old, red one in a stack of shirts, you first take the red shirt from the top of the stack. Then you remove the blue shirt, which is now at the top of the stack. The red shirt may then be put back on the top of the stack. Or it could be thrown away!

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

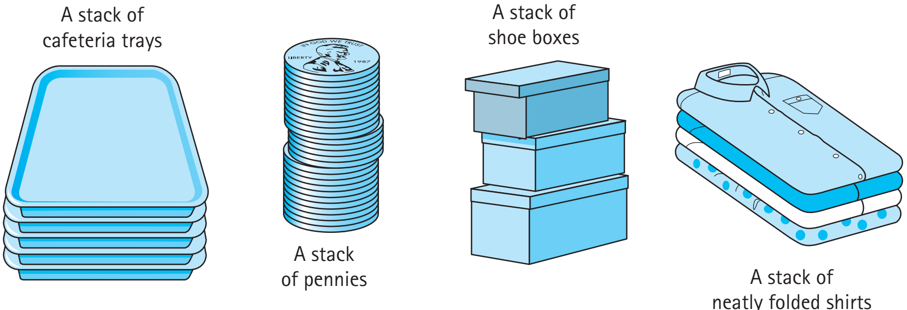


Figure 3.1 *Real-life stacks*

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

A stack may be considered "ordered" because elements occur in sequence according to how long they've been in the stack. The elements that have been in the stack the longest are at the bottom; the most recent are at the top. At any time, given any two elements in a stack, one is higher than the other. (For instance, the red shirt was higher in the stack than the blue shirt.)

Because elements are added and removed only from the top of the stack, the last element to be added is the first to be removed. There is a handy mnemonic to help you remember this rule of stack behavior: A stack is a LIFO ("last in, first out") structure.

The accessing protocol for a stack is summarized as follows: Both to retrieve elements and to store new elements, access only the top of the stack.

Operations on Stacks

The logical picture of the structure is only half the definition of an abstract data type. The other half is a set of operations that allows the user to access and manipulate the elements stored in the structure. What operations do we need to use a stack?

When we begin using a stack, it should be empty. Thus we assume that our stack has at least one class constructor that sets it to the empty state.

The operation that adds an element to the top of a stack is usually called *push*, and the operation that removes the top element off the stack is referred to as *pop*. Classically, the *pop* operation has both removed the top element of the stack and returned the top element to the client program that invoked *pop*. More recently, programmers have been defining two separate operations to perform these actions because operations that combine observations and transformation can result in confusing programs.

We follow modern convention and define a *pop* operation that removes the top element from a stack and a *top* operation that returns the top element of a stack.[1] Our *push* and *pop* operations are strictly transformers, and our *top* operation is strictly an observer. Figure 3.2 shows how a stack, envisioned as a stack of building blocks, is modified by several *push* and *pop* operations.

Using Stacks

Stacks are very useful ADTs, especially in the field of computing system software. They are most often used in situations in which we must process nested components. For example, programming language systems typically use a stack to keep track of operation calls. The main program calls operation A, which in turn calls operation B, which in turn calls operation C. When C finishes, control returns to B; when B finishes, control returns to A; and so on. The call and return sequence is essentially a last in, first out sequence, so a stack is the perfect structure for tracking it, as shown in Figure 3.3.

---

1. Another common approach is to define a `pop` operation in the classical way—that is, it removes and returns the top element—and to define another operation, often called `peek`, that simply returns the top element.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

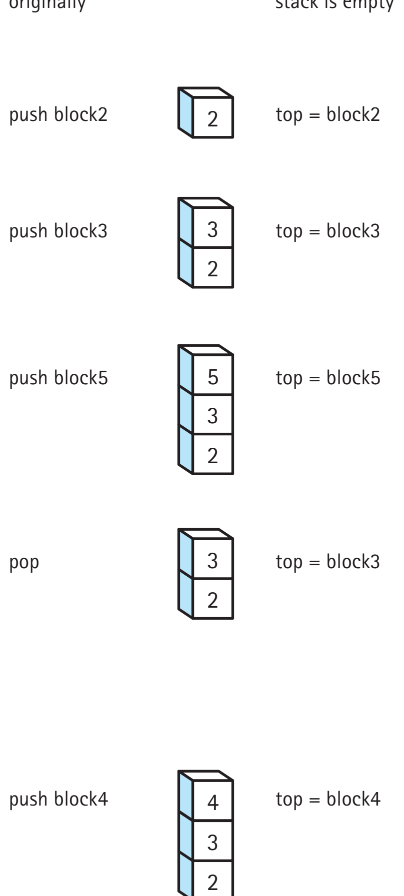


Figure 3.2 *The effects of `push` and `pop` operations*

You may have encountered a case where a Java exception has produced an error message that mentions "a system stack trace." This trace shows the nested sequence of method calls that ultimately led to the exception being thrown. These calls were saved on the "system stack."

Compilers use stacks to analyze language statements. A program often consists of nested components—for example, a *for* loop containing an *if-then* statement that contains a *while* loop. As a compiler is working through such nested constructs, it "saves" information about what it is currently working on in a stack; when it finishes its work on the innermost construct, it can "retrieve" its previous status from the stack and pick up where it left off.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

	Stack contains
main running	Top: <empty>
main calls method A method A running	Stack contains Top: main
method A calls method B method B running	Stack contains Top: A main
method B calls method C method C running	Stack contains Top: B A main
method C returns method B running	Stack contains Top: A main
method B returns method A running	Stack contains Top: main
method A returns main running	Stack contains Top: <empty>

Figure 3.3 *Call-return stack*

Similarly, an operating system sometimes saves information about the current executing process on a stack so that it can work on a higher-priority, interrupting process. If that process is interrupted by an even higher-priority process, its information can also be pushed on the process stack. When the operating system finishes its work on the highest-priority process, it retrieves the information about the most recently stacked process and continues working on it.

3.2 Collection Elements

A stack is an example of a collection ADT. A stack *collects* together elements for future use, while maintaining a first in, last out ordering among the elements. Before continuing our coverage of stacks, we examine the question of which types of elements can be stored in a collection. We look at several variations that are possible when structuring collections of elements and describe the approaches we adopt for use throughout this text. It is important to understand the various options, along with their strengths and weaknesses, so that you can make informed decisions about which approach to use based on your particular situation.

Collection An object that holds other objects. Typically we are interested in inserting, removing, and iterating through the contents of a collection.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Generally Usable Collections

The StringLog ADT we constructed in Chapter 2 is also a collection ADT. It was constrained to holding data of one specific type—namely, strings. Based on the approach used in that chapter, if we wanted to have a log of something else—say, integers or programmer-defined bank account objects—we would have to design and code additional ADTs. (See Figure 3.4a.)

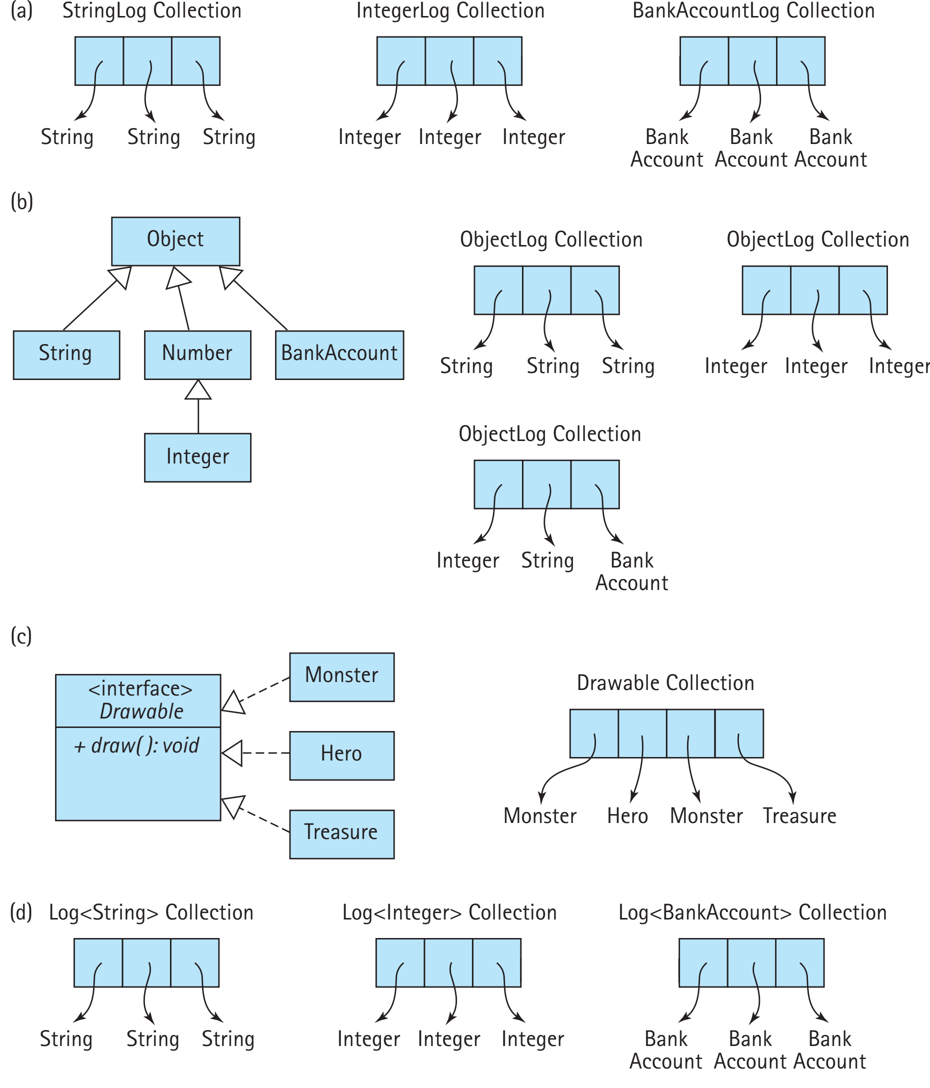


Figure 3.4 *Options for collection elements*

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Although the StringLog ADT is handy, a Log ADT would be much more useful if it could hold any kind of information. In Chapter 2 our goal was to present basic ADT concepts using a simple example, so we were content to create an ADT restricted to a single type of element. In this section we present several ways to design our collections so that they hold different types of information, making them more generally usable.

Collections of Class Object

One approach to creating generally usable collections is to have the collection ADT hold variables of class `Object`. Because all Java classes ultimately inherit from `Object`, such an ADT is able to hold a variable of any class. (See Figure 3.4b.) This approach works well, especially when the elements of the collection don't have any special properties—for example, if the elements don't have to be sorted.

Although this approach is simple, it is not without problems. One drawback: Whenever an element is removed from the collection, it can be referenced only 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, suppose you place a string into a collection and then retrieve it. To use the retrieved object as a `String` object you must cast it, as emphasized here:

```
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
```

Without the cast you will get a compile error, because Java is a strongly typed language and will not allow you to assign a variable of type `Object` to a variable of type `String`. The `cast` operation tells the compiler that you, the programmer, are guaranteeing that the `Object` is, indeed, a `String`.

The `Object` approach works by converting every class of object into class `Object` as it is stored in the collection. Users of the collection must remember what kinds of objects have been stored in it, and then explicitly cast those objects back into their original classes when they are removed from the collection.

As shown by the third `ObjectLog` collection in Figure 3.4(b), this approach allows a program to mix the types of elements in a single collection. That collection holds an `Integer`, a `String`, and a `BankAccount`. In general, such mixing is *not* considered to be a good idea, and it should be used only in rare cases under careful control. Its use can easily lead to a program that retrieves one type of object—say, an `Integer`—and tries to cast it as another type of object—say, a `String`. This, of course, is an error.

Collections of a Class That Implements a Particular Interface

Sometimes we may want to ensure that all of the objects in a collection support a particular operation or set of operations. As an example, suppose the objects represent elements of a video game. Many different types of elements exist, such as monsters, heroes, and treasure. When an element is removed from the collection it is drawn on the

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

screen, using a `draw` operation. In this case, we would like to ensure that only objects that support the `draw` operation can be placed in the collection.

Recall from Chapter 2 that a Java interface can include only abstract methods—that is, methods without bodies. Once an interface is defined we can create classes that implement the interface by supplying the missing method bodies. For our video game example we could create an interface with an abstract `draw` method. A good name for the interface might be `Drawable`, as classes that implement this interface provide objects that can be drawn. The various types of video game elements that can be drawn on the screen should all be defined as implementing the `Drawable` interface.

Now we can ensure that the elements in our example collection are all "legal" by designing it as a collection of `Drawable` objects—in other words, objects that implement the `Drawable` interface. In this way we ensure that only objects that support a `draw` operation are allowed in the collection. (See Figure 3.4c.)

Later in the text we present two data structures, the sorted list and the binary search tree, whose elements are organized in a sorted order. Every element that is inserted into one of these collections must provide an operation that allows us to compare it to other objects in its class. That is, objects inserted into the collection must support a `compareTo` operation. To enforce this requirement we define our sorted list and binary search ADTs as collections of `Comparable` objects. The `Comparable` interface is defined within the `java.lang` package and includes exactly one abstract method: `compareTo`.

Generic Collections

Beginning with version 5.0, the Java language supports generics. Generics allow us to define a set of operations that manipulate objects of a particular class, without specifying the class of the objects being manipulated until a later time. Generics represented one of the most significant changes to the language in this version of Java.

In a nutshell, generics are *parameterized types*. Of course, you are already familiar with the concept of a parameter. For example, in our `StringLog` class the `insert` method has a `String` parameter named `element`. When we invoke that method we must pass it a `String` argument, such as "Elvis":

```
log.insert("Elvis");
```

Generics allow us to pass type names such as `Integer`, `String`, or `BankAccount` as arguments. Notice the subtle difference—with generics we actually pass a *type*, for example, `String`, instead of a *value* of a particular type, for example, "Elvis."

With this capability, we can define a collection class, such as `Log`, as containing elements of a type T, where T is a placeholder for the name of a type. We indicate the name of the placeholder (convention tells us to use T) within braces; that is, <T>, in the header of the class.

```
public class Log<T>
{
  private T[] log;             // array that holds objects of class T
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
  private int lastIndex = -1; // index of last T in the array
  ...
```

In a subsequent application we can supply the actual type, such as `Integer`, `String`, or `BankAccount`, when the collection is instantiated.

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

If we pass `BankAccount` as the argument, we get a `BankAccount` log; if we pass `String`, we get a `String` log; and so on.

In place of passing a type when instantiating a generic collection we have the option of passing a java `interface`, for example

```
Log<Drawable> avatars;
```

As discussed in the previous subsection, this could then allow us to include any object that implements the interface in the collection—the `avatars` log could contain humans, elves, and ogres, as long as each of their respective classes implements the `Drawable` interface.

Generics provide the flexibility to design generally usable collections yet retain the benefit of Java's strong type checking. They are an excellent solution and we will use this approach throughout most of the remainder of this textbook.

3.3 Exceptional Situations

There is one more topic to cover before formally specifying our Stack ADT. In this section we take a look at various methods of handling exceptional situations that might arise when running a program. For example, what should happen if a stack is empty and the *pop* operation is invoked? There is nothing to pop! As part of formally specifying a stack, or any ADT, we must determine how such exceptional situations will be addressed.

Handling Exceptional Situations

Many different types of exceptional situations can occur when a program is running. Exceptional situations alter the flow of control of the program, sometimes resulting in a crash. Some examples follow:

> **Exceptional situation** Associated with an unusual, sometimes unpredictable event, detectable by software or hardware, which requires special processing. The event may or may not be erroneous.

- A user enters an input value of the wrong type.
- While reading information from a file, the end of the file is reached.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

- A user presses a control key combination.
- An illegal mathematical operation occurs, such as divide-by-zero.
- An impossible operation is requested of an ADT, such as an attempt to `pop` an empty stack.

Working with these kinds of exceptional situations begins at the design phase, when several questions arise: What are the unusual situations that the program should recognize? Where in the program can the situations be detected? How should the situations be handled if they occur?

Java (along with some other languages) provides built-in mechanisms to manage exceptional situations. In Java an exceptional situation is referred to simply as an *exception*. The Java exception mechanism has three major parts:

- *Defining the exception* Usually as a subclass of Java's `Exception` class
- *Generating (raising) the exception* By recognizing the exceptional situation and then using Java's `throw` statement to "announce" that the exception has occurred
- *Handling the exception* Using Java's `try-catch` statement to discover that an exception has been thrown and then take the appropriate action

Java also includes numerous predefined built-in exceptions that are raised automatically under certain situations.

From this point on we use the Java term "exception," instead of the more general phrase *exceptional situation*. Here are some general guidelines for using exceptions:

- An exception may be handled anywhere in the software hierarchy—from the place in the program module where it is first detected through the top level of the program.
- Unhandled built-in exceptions carry the penalty of program termination.
- Where in an application an exception is handled is a design decision; however, exceptions should always be handled at a level that knows what the exception means.
- An exception need not be fatal.
- For nonfatal exceptions, the thread of execution can continue from various points in the program, but execution should continue from the lowest level that can recover from the exception.

Exceptions and ADTs: An Example

When creating our own ADTs we identify exceptions that require special processing. If the special processing is application dependent, we use the Java exception mechanism to throw the problem out of the ADT and force the application programmers to handle it. Conversely, if the exception handling can be hidden within the ADT, then there is no need to burden the application programmers with the task.

For an example of an exception created to support a programmer-defined ADT, let's return to our `Date` class from Chapter 1.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
public class Date
{
  protected int year;
  protected int month;
  protected int day;
  public static final int MINYEAR = 1583;

  public Date(int newMonth, int newDay, int newYear)
  {
    month = newMonth;
    day = newDay;
    year = newYear;
  }

  public int getYear()
  {
    return year;
  }

  public int getMonth()
  {
    return month;
  }

  public int getDay()
  {
    return day;
  }

  public int lilian()
  {
    // Returns the Lilian Day Number of this date.
    // Algorithm goes here.
    // See "Lilian Day Numbers" feature, Chapter 1, for details.
  }

  public String toString()
  {
    return(month + "/" + day + "/" + year);
  }
}
```

As currently defined, an application could invoke the `Date` constructor with an impossible date—for example, 25/15/2000. We can avoid the creation of such dates by

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

checking the legality of the month argument passed to the constructor. But what should our constructor do if it discovers an illegal argument? Here are some options:

- Write a warning message to the output stream. This is not a good option because within the `Date` ADT we don't really know which output stream is used by the application.
- Instantiate the new `Date` object to some default date, perhaps 0/0/0. The problem with this approach is that the application program may just continue processing as if nothing is wrong and produce erroneous results. In general it is better for a program to "bomb" than to produce erroneous results that may be used to make bad decisions.
- Throw an exception. This way, normal processing is interrupted and the constructor does not have to return a new object; instead, the application program is forced to acknowledge the problem (catch the exception) and either handle it or throw it to the next level.

Once we have decided to handle the situation with an exception, we must decide whether to use one of the Java library's predefined exceptions or to create one of our own. A study of the library in this case reveals a candidate exception called `DataFormatException`, to be used to signal data format errors. We could use that exception but we decide it doesn't really fit: It's not the format of the data that is the problem in this case, it's the value of the data.

We decide to create our own exception, `DateOutOfBounds`. We could call it "MonthOutOfBounds" but we decide that we want to use the exception to indicate other potential problems with dates, not just problems with the month value.

We create our `DateOutOfBounds` exception by extending the library's `Exception` class. It is customary when creating your own exceptions to define two constructors, mirroring the two constructors of the `Exception` class. In fact, the easiest thing to do is define the constructors so that they just call the corresponding constructors of the superclass:

```
public class DateOutOfBoundsException extends Exception
{
  public DateOutOfBoundsException()
  {
    super();
  }
  public DateOutOfBoundsException(String message)
  {
    super(message);
  }
}
```

The first constructor creates an exception without an associated message. The second constructor creates an exception with a message equal to the string argument passed to the constructor.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Next we need to consider where, within our `Date` ADT, we throw the exception. All places within our ADT where a date value is created or changed should be examined to see if the resultant value could be an illegal date. If so, we should create an object of our exception class with an appropriate message and throw the exception.

Here is how we might write a `Date` constructor to check for legal months and years:

```
public Date(int newMonth, int newDay, int newYear)
                               throws DateOutOfBoundsException
{
  if ((newMonth <= 0) || (newMonth > 12))
    throw new DateOutOfBoundsException("month " + newMonth + "out of range");
  else
    month = newMonth;

  day = newDay;

  if (newYear < MINYEAR)
    throw new DateOutOfBoundsException("year " + newYear + " is too early");
  else
    year = newYear;
}
```

Notice that the message defined for each `throw` statement pertains to the problem discovered at that point in the code. This should help the application program that is handling the exception, or at least provide pertinent information to the user of the program if the exception is propagated all the way to the user level.

Finally, let's see how an application program might use the revised `Date` class. Consider a program called `UseDates` that prompts the user for a month, day, and year and creates a `Date` object based on the user's responses. In the following code we hide the details of how the prompt and response are handled, by replacing those statements with comments. This way we can emphasize the code related to our current discussion:

```
public class UseDates
{
  public static void main(String[] args)
                         throws DateOutOfBoundsException
  {
    Date theDate;

    // Program prompts user for a date.
    // M is set equal to user's month.
    // D is set equal to user's day.
    // Y is set equal to user's year.
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
    theDate = new Date(M, D, Y);

    // Program continues ...
  }
}
```

When this program runs, if the user responds with an illegal value—for example, a year of 1051—the `DateOutOfBoundsException` is thrown by the `Date` constructor; because it is not caught and handled within the program, it is thrown to the interpreter as indicated by the emphasized `throws` clause. The interpreter stops the program and displays a message like this:

```
Exception in thread "main" DateOutOfBoundsException: year 1051 is too
early
          at Date.<init>(Date.java:18)
          at UseDates.main(UseDates.java:57)
```

The interpreter's message includes the name and message string of the exception as well as a trace of calls leading up to the exception (the system stack trace mentioned in Section 3.1.)

Alternatively, the `UseDates` class could catch and handle the exception itself, rather than throwing it to the interpreter. The application could ask for a new date when the exception occurs. Here is how `UseDates` can be written to do this (again we ignore user interface details and emphasize code related to exceptions):

```
public class UseDates
{
  public static void main(String[] args)
  {
    Date theDate;
    boolean DateOK = false;

    while (!DateOK)
    {
      // Program prompts user for a date.
      // M is set equal to user's month.
      // D is set equal to user's day.
      // Y is set equal to user's year.
      try
      {
        theDate = new Date(M, D, Y);
        DateOK = true;
      }
      catch(DateOutOfBoundsException DateOBExcept)
      {
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
                output.println(DateOBExcept.getMessage());
            }
        }

        // Program continues ...
    }
}
```

If the `new` statement executes without any trouble, meaning the `Date` constructor did not throw an exception, then the `DateOK` variable is set to `true` and the *while* loop terminates. However, if the `DateOutOfBounds` exception is thrown by the `Date` constructor, it is caught by the `catch` statement. This, in turn, prints the message from the exception and the `while` loop is reexecuted, again prompting the user for a date. The program repeatedly prompts for date information until it is given a legal date. Notice that the `main` method no longer throws `DateOutOfBoundsException`, as it handles the exception itself.

One last important note about exceptions. The `java.lang.RunTimeException` class is treated uniquely by the Java environment. Exceptions of this class are thrown when a standard run-time program error occurs. Examples of run-time errors include division-by-zero and array-index-out-of-bounds. Because run-time exceptions can happen in virtually any method or segment of code, we are not required to explicitly handle these exceptions. Otherwise, our programs would become unreadable because of so many `try`, `catch`, and `throw` statements. These errors are classified as unchecked exceptions. The exceptions we create later in this chapter to support our Stack ADT are extensions of the Java `RunTimeException` class and, therefore, are unchecked.

Unchecked exception An exception of the `RunTimeException` class. It does not have to be explicitly handled by the method within which it might be raised.

Error Situations and ADTs

When dealing with error situations within our ADT methods, we have several options.

First, we can detect and handle the error within the method itself. This is the best approach if the error can be handled internally and if it does not greatly complicate the design. For example, if an illegal value is passed to a method, we may be able to replace it with a useful default value. Suppose we have a method used to set the employee discount for an online store. Passing it a negative number might be construed as an error—it might be reasonable in such a case to set the value to zero and continue processing. When handling a problem internally in this way, it may be possible to pass information about the situation from the method to the caller through a return value. For example, we could design a *push* operation for a stack that returns a `boolean` value of `true` if the operation is successful and the argument is pushed onto the stack, and a value of `false` if the operation fails for some reason (e.g., because the

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

stack is full). As another example, consider a *top* operation that returns the object from the top of the stack. Instead of terminating the program if the stack is empty, the operation could return the value `null`, indicating that the operation "failed." In these examples the caller is responsible for checking the returned value and acting appropriately if failure is indicated.

Second, we can detect the error within the method, throw an exception related to the error, and thereby force the calling method to deal with the exception. If it is not clear how to handle a particular error situation, this approach might be best—throw it to a level where it can be handled. For example, if an application passes a nonsensical date to the `Date` class constructor, it is best to throw an exception—the constructor doesn't "know" what the ramifications of the impossible date are, but the application should. Another example is when an application attempts to *pop* something from an empty stack. The Stack ADT doesn't "know" what this erroneous situation means, but the application should. Of course, if the caller does not catch and handle the thrown exception, it will continue to be thrown until it is either handled or thrown all the way to the interpreter, causing program termination.

Third, we can ignore the error situation. Recall the "programming by contract" discussion related to preconditions in Chapter 2. With this approach, if the preconditions of a method are not met, the method is not responsible for the consequences. For example, suppose a method requires a prime number as an argument. If this is a precondition, then the method assumes that the argument is prime—it does not test the primality of the number. If the number is not prime, then the results are undefined. It is the responsibility of the calling code to ensure that the precondition is met. See the feature "Programming by Contract" for more information.

When we define an ADT, we partition error situations into three sets: those to be handled internally, those to be thrown back to the calling process, and those that are assumed not to occur. We document this third approach in the preconditions of the appropriate methods. Our goal is to strike a balance between the complexity required to handle every error situation internally and the lack of safety resulting from handling everything by contract.

Programming by Contract

Let's revisit briefly our "programming by contract" approach. We want to emphasize the way we handle method preconditions, because some programmers use a different methodology: They test preconditions within a method and throw exceptions when preconditions aren't met. They treat unmet preconditions as errors. We don't.

We don't believe in unmet preconditions. If a condition might not be true when a method is called, then it shouldn't be listed as a precondition. It should be listed as an error condition. The point of a precondition is to simplify our code and make it more efficient, not to complicate things with extra levels of unneeded testing.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Why is our approach more efficient? Preconditions are always supposed to be true. Thus testing them each time a method is called is a waste of time.

Consider the example of the method that requires a prime number as an argument. Suppose `methodA` obtains a prime number and passes it to `methodB`. It guarantees that the number is prime. If we require `methodB` to test the primality of the number (a nontrivial task), we are unnecessarily complicating `methodB` and slowing down our program. Instead, we simply state within our preconditions that the argument is prime and no longer worry about it:

```
public void methodB(int primenumber)
// Precondition:   primenumber is prime.
. . .
```

On the other hand, if we cannot assume that the number passed to `methodB` is prime, then we document this possibility as a potential error condition, test for it, and handle it if needed:

```
public void methodB(int primenumber) throws NotPrimeException
// Throws NotPrimeException if primenumber is not prime,
// otherwise ...
```

For any specific condition we use one or the other of these approaches, but not both!

3.4 Formal Specification

In this section we use the Java interface construct to create a formal specification of our Stack ADT. To specify any collection ADT we must determine which types of elements it will hold, which operations it will export, and how exceptional situations will be handled. Some of these decisions have already been documented.

Recall from Section 3.1 that a stack is a "last in, first out" structure, with three 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 to these operations we need a constructor that creates an empty stack.

As noted in Section 3.2, 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. Following the common Java coding convention, we use `<T>` to represent the class of objects stored in our stack.

Now we look at exceptional situations. As you'll see, this exploration can lead to the identification of additional operations.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Exceptional Situations

Are there any exceptional situations that require handling? The constructor simply initializes a new empty stack. This action, in itself, cannot cause an error—assuming, of course, that it is coded correctly.

The remaining operations all present potential problem situations. The descriptions of the `pop` and `top` operations both refer to manipulating the "top element of the stack." But what if the stack is empty? Then there is no top element to manipulate. We know that there are three ways to deal with this scenario. Can we handle the problem within the methods themselves? Should we detect the situation and throw an exception? Is it reasonable to state, as a precondition, that the stack be nonempty?

How might the problem be handled within the methods themselves? Given that the `pop` method is strictly a transformer, it could simply do nothing when it is invoked on an empty stack. In effect, it could perform a vacuous transformation. For `top`, which must return an `Object` reference, the response might be to return `null`. For some applications this might be a reasonable approach, but for most cases it would merely complicate the application code.

What if we state a precondition that a stack must not be empty before calling `top` or `pop`? Then we do not have to worry about handling the situation within the ADT. Of course, we can't expect every application that uses our stack to keep track of whether it is empty; that should be the responsibility of the Stack ADT itself. To address this requirement we define an observer called `isEmpty`, which returns a `boolean` value of `true` if the stack is empty. Then the application can prevent misuse of the `pop` and `top` operations.

```
if !myStack.isEmpty()
  myObject = myStack.top();
```

This approach appears promising but can place an unwanted burden on the application. If an application must perform a guarding test before every stack operation, its code might become inefficient and difficult to read.

It is also a good idea to provide an exception related to accessing an empty stack. Consider the situation where a large number of stack calls take place within a section of code. If we define an exception—for example, `StackUnderflowException`—to be thrown by both `pop` and `top` if they are called when the stack is empty, then such a section of code could be surrounded by a single *try-catch* statement, rather than use multiple calls to the `isEmpty` operation.

We decide to use this last approach. That is, we define a `StackUnderflowException`, to be thrown by both `pop` and `top` if they are called when the stack is empty. To provide flexibility to the application programmer, we also include the `isEmpty` operation in our ADT. Now the application programmer can decide either to prevent popping or accessing an empty stack by using the `isEmpty` operation as a guard or, as shown next, to "try" the operations on the stack and "catch and handle" the raised exception, if the stack is empty.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
try
{
  myObject = myStack.top();
  myStack.pop();
  myOtherObject = myStack.top();
  myStack.pop();
}
catch (StackUnderflowException underflow)
{
  System.out.println("There was a problem in the ABC routine.");
  System.out.println("Please inform System Control.");
  System.out.println("Exception: " + underflow.getMessage());
  System.exit(1);
}
```

We define `StackUnderflowException` to extend the Java `RuntimeException`, as it represents a situation that a programmer can avoid by using the stack properly. The `RuntimeException` class is typically used in such situations. Recall that such exceptions are unchecked; in other words, they do not have to be explicitly caught by a program.

Here is the code for our `StackUnderflowException` class. Note that it includes a `package` statement. This class is the first of several classes and interfaces we develop related to the stack data structure. We collect all of these together into a single package called `ch03.stacks`.[2]

```
package ch03.stacks;

public class StackUnderflowException extends RuntimeException
{
  public StackUnderflowException()
  {
    super();
  }

  public StackUnderflowException(String message)
  {
    super(message);
  }
}
```

---

2. The files can be found in the `stacks` subdirectory of the `ch03` subdirectory of the `bookFiles` directory that contains the program files associated with the textbook.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Because `StackUnderflowException` is an unchecked exception, if it is raised and not caught it is eventually thrown to the run-time environment, which displays an error message and halts. An example of such a message follows:

```
Exception in thread "main" ch03.stacks.StackUnderflowException: Top attempted on an empty stack.
at ch03.stacks.ArrayStack.top(ArrayStack.java:78)
at MyTestStack.main(MyTestStack.java:25)
```

On the other hand, if the programmer explicitly catches the exception, as we showed in the `try-catch` example, the error message can be tailored more closely to the specific problem:

```
There was a problem in the ABC routine.
Please inform System Control.
Exception: top attempted on an empty stack.
```

A consideration of the `push` operation reveals another potential problem: What if we try to push something onto a stack and there is no room for it? In an abstract sense, a stack is never conceptually "full." Sometimes, however, it is useful to specify an upper bound on the size of a stack. We might know that memory is in short supply or problem-related constraints may dictate a limit on the number of `push` operations that can occur without corresponding `pop` operations.

We can address this problem in a way analogous to the stack underflow problem. First, we provide an additional `boolean` observer operation called `isFull`, which returns `true` if the stack is full. The application programmer can use this operation to prevent misuse of the `push` operation. Second, we define `StackOverflowException`, which is thrown by the `push` operation if it is called when the stack is full. Here is the code for the `StackOverflowException` class:

```
package ch03.stacks;

public class StackOverflowException extends RuntimeException
{
  public StackOverflowException()
  {
    super();
  }

  public StackOverflowException(String message)
  {
    super(message);
  }
}
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

As with the underflow situation, the application programmer can decide either to prevent pushing information onto a full stack through use of the `isFull` operation or to "try" the operation on a stack and "catch and handle" any raised exception. The `StackOverflowException` is also an unchecked exception.

The Interfaces

We are now ready to formally specify our Stack ADT. As we planned, we use the Java `interface` construct. But how do we handle the fact that sometimes we may want to use a stack with an upper bound on its size and sometimes we want an unbounded stack?

We were faced with this same situation in developing our `StringLog` ADT in Chapter 2. In that case we decided to include the `isFull` operation as part of the single interface, even though its existence did not make sense for some implementations of the `StringLog`. Recall that in the linked-list-based implementation, `isFull` *always* returned `false`. For the Stack ADT we use a different approach: We define separate interfaces for the bounded and unbounded versions of the stack. In fact, we define three interfaces.

Whether a stack is bounded in size affects only the `push` operation and the need for an `isFull` operation. It has no effect on the `pop`, `top`, or `isEmpty` operations. First, we define a general stack interface, `StackInterface`, that contains the signatures of those three operations:

```
//----------------------------------------------------------------------------
// StackInterface.java          by Dale/Joyce/Weems                 Chapter 3
//
// Interface for a class that implements a stack of T.
// A stack is a last in, first out structure.
//----------------------------------------------------------------------------

package ch03.stacks;

public interface StackInterface<T>

{
  void pop() throws StackUnderflowException[3];
  // 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 from this stack.
```

3. Because our stack exceptions are unchecked exceptions, including them in the interface actually has no effect on anything from a syntactic or run-time error-checking point of view. They aren't checked. However, we still list them as being thrown because we are also trying to communicate our requirements to the implementation programmer.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


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

}
```

In Section 3.2 we presented our intention to create generic collection ADTs. This means that in addition to implementing our ADTs as generic classes—that is, classes that accept a parameter type upon instantiation—we also will define generic interfaces for those classes. Note the use of <T> in the header of `StackInterface`. As with generic classes, `<T>` used in this way indicates that T is a placeholder for a type provided by the client code. T represents the class of objects held by the specified stack. Since the `top` method returns one of those objects, in the interface it is listed as returning T. This same approach is used for ADT interfaces throughout the remainder of the textbook.

Note that we document the effects of the operations, the postconditions, as comments. For this ADT there are no preconditions because we have elected to throw exceptions for all error situations.

Next, we turn our attention to the bounded version of the stack, for which we create a second interface, `BoundedStackInterface`. A stack that is bounded in size must support all of the operations of a “regular” stack plus the `isFull` operation. It must also provide a `push` operation that throws an exception if the operation is invoked when the stack is full.

Java supports inheritance of interfaces. That is, one interface can extend another interface. (In fact, the language supports multiple inheritance of interfaces so that a single interface can extend any number of other interfaces.) The fact that the new interface requires all of the operations of our current `StackInterface` makes this a perfect place to use inheritance. We define our `BoundedStackInterface` as a new interface that extends `StackInterface` and adds `isFull` and `push` methods. Here is the code for the new interface (note the `extends` clause):

Inheritance of interfaces A Java interface can extend another Java interface, inheriting its requirements. If interface B extends interface A, then classes that implement interface B must also implement interface A. Usually, interface B adds abstract methods to those required by interface A.

Multiple inheritance of interfaces A Java interface may extend more than one interface.[4] If interface C extends both interface A and interface B, then classes that implement interface C must also implement both interface A and interface B. Sometimes multiple inheritance of interfaces is used simply to combine the requirements of two interfaces, without adding any more methods.

```
//----------------------------------------------------------------------------
// BoundedStackInterface.java        by Dale/Joyce/Weems            Chapter 3
//
// Interface for a class that implements a stack of T with a bound
// on the size of the stack. A stack is a last in, first out structure.
//----------------------------------------------------------------------------

package ch03.stacks;
```

4. In contrast, a Java class can extend only one other class.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
public interface BoundedStackInterface<T> extends StackInterface<T>

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

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

}
```

Finally, we create an interface for the unbounded case. An unbounded stack need not support an `isFull` operation, because it will "never" be full.[5] For this same reason, the `push` operation need not throw an exception.

```
//----------------------------------------------------------------------------
// UnboundedStackInterface.java       by Dale/Joyce/Weems            Chapter 3
//
// Interface for a class that implements a stack of T with no bound
// on the size of the stack. A stack is a last in, first out structure.
//----------------------------------------------------------------------------

package ch03.stacks;

public interface UnboundedStackInterface<T> extends StackInterface<T>

{
  void push(T element);
  // Places element at the top of this stack.

}
```

We have now defined our three interfaces, along with two exception classes. Their relationship is shown in the UML diagram in Figure 3.5. A specific implementation of a stack would `implement` either the `BoundedStackInterface` or the `UnboundedStack-Interface`. By virtue of the interface inheritance rules, it must also implement the `StackInterface`.

5. The only way it could be full is if the system runs out of space. In this rare case the Java run-time system raises an exception anyway.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

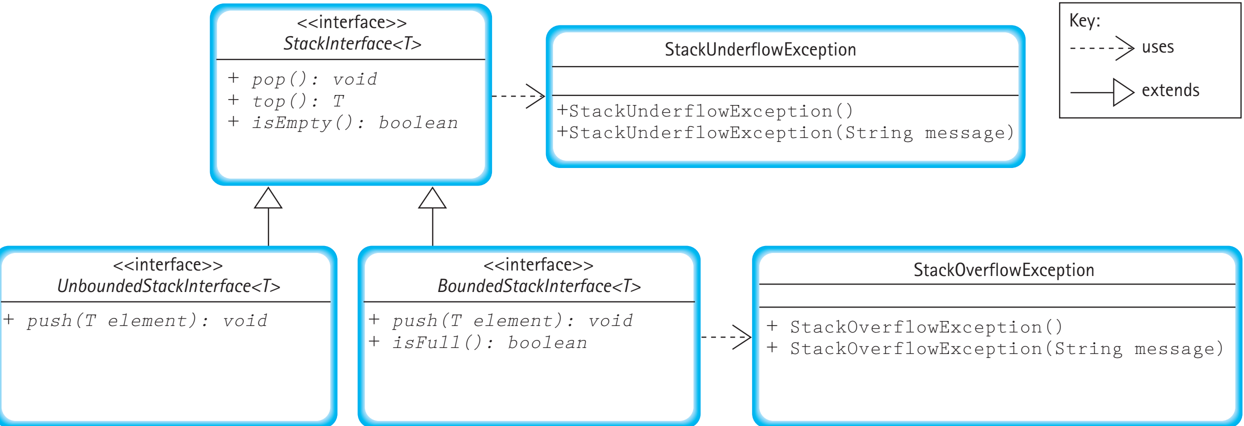


Figure 3.5 *UML diagram of our Stack ADT interfaces*

Note that our Stack ADT interfaces are not an example of multiple inheritance of interfaces. That occurs when one interface inherits from more than one other interface. Here we have defined two interfaces that inherit from the same interface, just as in a normal class hierarchy. We discussed multiple inheritance in this section simply because it is an aspect of the Java syntax for interface inheritance that you should be aware of.

Example Use

The simple `ReverseStrings` example shows how we can use a stack to store strings provided by a user and then to output the strings in the opposite order from which they were entered. The code uses the array-based implementation of a stack we develop in the following section. The parts of the code directly related to the creation and use of the stack are emphasized. We declare the stack to be of type `BoundedStackInterface<String>` and then instantiate it as an `ArrayStack<String>`. Within the *for* loop, three strings provided by the user are pushed onto the stack. The *while* loop repeatedly removes and prints the top string from the stack until the stack is empty. If we try to push any type of object other than a `String` onto the stack, we will receive a compile time error message saying that the `push` method cannot be applied to that type of object.

```
//----------------------------------------------------------------------
// ReverseStrings.java        by Dale/Joyce/Weems             Chapter 3
//
// Sample use of stack. Outputs strings in reverse order of entry.
//----------------------------------------------------------------------
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
import ch03.stacks.*;
import java.util.Scanner;

public class ReverseStrings
{
  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);

    BoundedStackInterface<String> stack;
    stack = new ArrayStack<String>(3);

    String line;

    for (int i = 1; i <= 3; i++)
    {
      System.out.print("Enter a line of text > ");
      line = conIn.nextLine();
      stack.push(line);
    }

    System.out.println("\nReverse is:\n");
    while (!stack.isEmpty())
    {
      line = stack.top();
      stack.pop();
      System.out.println(line);
    }
  }
}
```

Here is the output from a sample run:

```
Enter a line of text > the beginning of a story
Enter a line of text > is often different than
Enter a line of text > the end of a story

Reverse is:

the end of a story
is often different than
the beginning of a story
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

The Java Stack Class and the Collections Framework

The Java library provides classes that implement ADTs that are based on common data structures—stacks, queues, lists, maps, sets, and more. The library's `Stack` class is similar to the Stack ADT we develop in this chapter in that it provides a LIFO structure. However, in addition to our `push`, `top`, and `isEmpty`[6] operations, it includes two other operations:

- `pop` Removes and returns the top element from the stack.
- `search(Object o)` Returns the position of object `o` on the stack.

Because the library `Stack` class extends the library `Vector` class, it also inherits the many operations defined for `Vector` and its ancestors.

Here is how you might implement the reverse strings application using the `Stack` class from the Java library. The minimal differences between this application and the one using our Stack ADT are emphasized.

```
//---------------------------------------------------------------------
// ReverseStrings2.java          by Dale/Joyce/Weems          Chapter 3
//
// Sample use of the library Stack.
// Outputs strings in reverse order of entry.
//---------------------------------------------------------------------

import java.util.Stack;
import java.util.Scanner;

public class ReverseStrings2
{
  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);

    Stack<String> stack = new Stack<String>();

    String line;

    for (int i = 1; i <= 3; i++)
    {
      System.out.print("Enter a line of text > ");
      line = conIn.nextLine();
      stack.push(line);
    }
```

6. In the library `isEmpty` is called `empty`, and `top` is called `peek`.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
    System.out.println("\nReverse is:\n");
    while (!stack.empty())
    {
      line = stack.peek();
      stack.pop();
      System.out.println(line);
    }
  }
}
```

As discussed in Section 3.2, another term for a data structure is collection. The Java developers refer to the set of library classes, such as `Stack`, that support data structures as the *Collections Framework*. This framework includes both interfaces and classes. It also includes documentation that explains how the developers intend for us to use them. As of Java 5.0 all the structures in the Collections Framework support generics (see the subsection Generic Collections in Section 3.2).

The Collections Framework comprises an extensive set of tools. It does more than just provide implementations of data structures; it provides a unified architecture for working with collections. In this textbook we do not cover the framework in great detail. This textbook is meant to teach you about the fundamental nature of data structures and to demonstrate how we define, implement, and use them. It is not an exploration of how to use Java's specific library architecture of similar structures.

Before you become a professional Java programmer, you should carefully study the Collections Framework and learn how to use it productively. This textbook prepares you to do this not just for Java, but for other languages and libraries as well. Nevertheless, when we discuss a data structure that has a counterpart in the Java library, we will briefly describe the similarities and differences between our approach and the library's approach, as we did here for stacks.

If you are interested in learning more about the Java Collections Framework, you can study the extensive documentation available at Oracle's website.

3.5 Array-Based Implementations

In this section we study an array-based implementation of the Stack ADT. Additionally, in a feature section, we look at an alternative implementation that uses the Java library's `ArrayList` class.

Note that Figure 3.17, in the Summary on page 230, shows the relationships among the primary classes and interfaces created to support our Stack ADT, including those developed in this section.

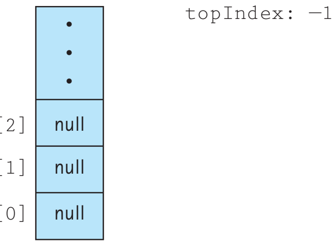
The ArrayStack Class

First we develop a Java class that implements the `BoundedStackInterface`. We call this class `ArrayStack`, in recognition of the fact that it uses an array as the underlying

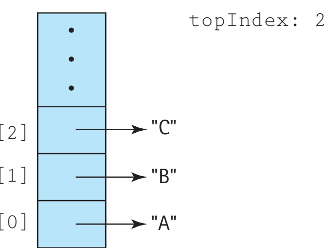
© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

structure. An array is a reasonable structure to contain elements of a stack. We can put elements into sequential slots in the array, placing the first element pushed onto the stack into the first array position, the second element pushed into the second array position, and so on. The floating "high-water" mark is the top element in the stack. Given that stacks grow and shrink from only one end, we do not have to worry about inserting an element into the middle of the elements already stored in the array.

What instance variables does our implementation need? We need the stack elements themselves and a variable indicating the top of the stack. We hold the stack elements in a protected array called `stack`. We use a protected integer variable called `topIndex` to indicate which element of the array is the top. We initialize `topIndex` to −1, as nothing is stored on the stack when it is first created.



As we push and pop elements, respectively, we increment and decrement the value of `topIndex`. For example, starting with an empty stack and pushing "A," "B," and "C" we would have



We provide two constructors for use by clients of the `ArrayStack` class: One allows the client to specify the maximum expected size of the stack, and the other assumes a default maximum size of 100 elements. To facilitate the latter constructor, we define a constant `DEFCAP` (default capacity) set to 100.

The beginning of the `ArrayStack.java` file is shown here:

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

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
// 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];6
  }

  public ArrayStack(int maxSize)
  {
    stack = (T[]) new Object[maxSize];6
  }
```

We can see that this class accepts a generic parameter `<T>` as listed in the class header. The `stack` variable is declared to be of type `T[]`, that is, an array of class `T`. This class implements a stack of `T`'s—the class of `T` is not yet determined. It will be specified by the client class that uses the bounded stack. Because the Java translator will not generate references to a generic type, our code must specify `Object` along with the *new* statement within our constructors. Thus, although we declare our array to be an array of class `T`, we must instantiate it to be an array of class `Object`. Then, to ensure that the desired type checking takes place, we cast array elements into class `T`, as shown here:

```
stack = (T[]) new Object[DEFCAP];
```

Even though this approach is somewhat awkward and typically generates a compiler warning, it is how we must create generic collections using arrays in Java. We could use the Java library's generic `ArrayList` to rectify the problem (see the feature *The*

6. An unchecked cast warning is generated because the compiler cannot ensure that the array contains objects of class T—the warning can safely be ignored.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

ArrayListStack Class at the end of this section), but we prefer to use the more basic array structure for pedagogic reasons. The compiler warning can safely be ignored.

Definitions of Stack Operations

As we are now implementing the BoundedStackInterface, we must provide a concrete implementation of the isFull method. For the array-based approach, the implementations of isFull and its counterpart, isEmpty, are both very simple. The stack is empty if the top index is equal to –1, and the stack is full if the top index is equal to one less than the size of the array.

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

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

Now let's write the method to push an element of type T onto the top of the stack. If the stack is already full when we invoke push, there is nowhere to put the element. Recall that this condition is called stack overflow. Our formal specifications state that the method should throw the StackOverflowException in this case. We include a pertinent error message when the exception is thrown. If the stack is not full, push must increment topIndex and store the new element into stack[topIndex]. The implementation of this method is straightforward.

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

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


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

The `pop` method is essentially the reverse of `push`: Instead of putting an element onto the top of the stack, we remove the top element from the stack by decrementing `topIndex`. It is good practice to also "null out" the array location associated with the current top. Setting the array value to `null` removes the physical reference. Figure 3.6 shows the difference between the "lazy" approach to coding `pop` and the "proper" approach.

If the stack is empty when we invoke `pop`, there is no top element to remove and we have stack underflow. As with the `push` method, the specifications say to throw an exception.

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

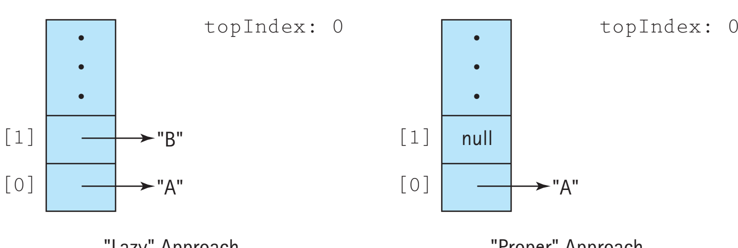


Figure 3.6 *Lazy versus proper* `pop` *approaches for an array-based stack after* `push("A")`, `push("B")`, *and* `pop()`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Finally, the top operation simply returns the top element of the stack, the element indexed by topIndex. Consistent with our generic approach, the top method shows type T as its return type. As with the pop operation, if we attempt to perform the top operation on an empty stack, a stack underflow results.

```
public T top()
// Throws StackUnderflowException if this stack is empty,
// otherwise returns top element from this stack.
{
  T topOfStack = null;
  if (!isEmpty())
    topOfStack = stack[topIndex];
  else
    throw new StackUnderflowException("Top attempted on an empty stack.");
  return topOfStack;
}
```

Test Plan

Our ArrayStack implementation can be tested using the general ADT testing approach described in Section 2.4, "Software Testing," where we presented an example based on the StringLog ADT. Unlike the StringLog ADT, our Stack ADT does not include a toString operation. Therefore it is not as easy to check the contents of a stack during testing. There are several ways we can address this problem. For instance, we could add a toString operation to our stack implementations. (See Exercise 29.) Alternatively, we could create an application-level method that is passed a stack; uses top, pop, and push to take the stack apart and display its contents; and then uses the same operations to put the stack back together. This approach requires a second stack to hold the contents of the original stack under investigation, while it is being "taken apart." (See Exercise 30d.)

Once the problem of viewing the contents of a stack has been solved, we can create an interactive test driver, as we did for the ArrayStringLog in Chapter 2. Such a driver helps us carry out our test plans.

Page 191 shows a short test plan for the ArrayStack. The test plan tests a stack of Integer. The type of data stored in the stack has no effect on the operations that manipulate the stack, so testing an Integer stack suffices. We set the stack size to 5, to keep our test cases manageable.

One final note about using an array to implement a stack. We implemented the BoundedStackInterface using an array because the size of an array is fixed. We can also use arrays to implement the UnboundedStackInterface. One approach is to instantiate increasingly larger arrays, as needed during processing, copying the current array into the larger, newly instantiated array. We investigate this approach when we implement the Queue ADT in Chapter 5.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Operation to be Tested and Description of Action	Input Values	Expected Output
ArrayStack	5	
apply isEmpty immediately		Stack is empty
push, pop, and top		
push 4 items, top/pop and print	5,7,6,9	9,6,7,5
push with duplicates and top/pop and print	2,3,3,4	4,3,3,2
interlace operations		
push	5	
pop		
push	3	
push	7	
pop		
top and print		3
isEmpty		
invoke when empty		Stack is empty
push and invoke		Stack is not empty
pop and invoke		Stack is empty
isFull		
push 4 items and invoke		Stack is not full
push 1 item and invoke		Stack is full
throw StackOverflowException push 5 items then push another item		Outputs string: "Push attempted on a full stack." Program terminates
throw StackUnderflowException when stack is empty attempt to pop		Outputs string: "Pop attempted on an empty stack." Program terminates
when stack is empty attempt to top		Outputs string: "Top attempted on an empty stack." Program terminates

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

The ArrayListStack Class

There are often many ways to implement an ADT. In this feature, we present an alternate implementation for the Stack ADT based on the `ArrayList`[7] class of the Java class library. The `ArrayList` is part of the Java Collections Framework discussed at the end of Section 3.4.

The defining quality of the `ArrayList` class is that it can grow and shrink in response to the program's needs. As a consequence, when we use the `ArrayList` approach we do not have to worry about our stacks being bounded. Instead of implementing the `BoundedStackInterface`, we implement the `UnboundedStackInterface`. Our constructor no longer needs to declare a stack size. We do not implement an `isFull` operation. We do not have to handle stack overflows.

One could argue that if a program runs completely out of memory, then the stack could be considered full and should throw `StackOverflowException`. However, in that case the run-time environment throws an "out of memory" exception anyway; we do not have to worry about the situation going unnoticed. Furthermore, running out of system memory is a serious problem (and ideally a rare event) and cannot be handled in the same way as a Stack ADT overflow.

The fact that an `ArrayList` automatically grows as needed makes it a good choice for implementing our unbounded Stack ADT. Additionally, it provides a `size` method that we can use to keep track of the top of our stack. The index of the top of the stack is always the `size` minus one.

Study the following code. Compare this implementation to the previous implementation. They are similar, yet different. One is based directly on arrays, whereas the other uses arrays indirectly through the `ArrayList` class. One nice benefit of using the `ArrayList` approach is we no longer receive the annoying unchecked cast warning from the compiler. This is because an `ArrayList` object, unlike the basic array, is a first-class object in Java and fully supports the use of generics. Despite the obvious benefits of using `ArrayList` we will continue to use arrays as one of our basic ADT implementation structures throughout most of the rest of the book. Learning to use the standard array is important for future professional software developers.

```
//----------------------------------------------------------------------
// ArrayListStack.java        by Dale/Joyce/Weems             Chapter 3
//
// Implements UnboundedStackInterface using an ArrayList to
// hold the stack elements.
//----------------------------------------------------------------------

package ch03.stacks;

import java.util.*;
```

7. Appendix E contains information concerning the Java `ArrayList` class.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


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

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

  public void push(T element)
  // Places element at the top of this stack.
  {
    stack.add(element);
  }

  public void pop()
  // Throws StackUnderflowException if this stack is empty,
  // otherwise removes top element from this stack.
  {
    if (!isEmpty())
    {
      stack.remove(stack.size() - 1);
    }
    else
      throw new StackUnderflowException("Pop attempted on an empty " +
                                        "stack.");
  }

  public T top()
  // Throws StackUnderflowException if this stack is empty,
  // otherwise returns top element from this stack.
  {
    T topOfStack = null;
    if (!isEmpty())
      topOfStack = stack.get(stack.size() - 1);
    else
      throw new StackUnderflowException("Top attempted on an empty " +
                                        "stack.");
    return topOfStack;
  }

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

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

3.6 Application: Well-Formed Expressions

Stacks are great for "remembering" things that have to be "taken care of" at a later time. In this sample application we tackle a problem that perplexes many beginning programmers: matching parentheses, brackets, and braces in writing code. Matching *grouping symbols* is an important problem in the world of computing. For example, it is related to the legality of arithmetic equations, the syntactical correctness of computer programs, and the validity of XHTML tags used to define web pages. This problem is a classic situation for using a stack, because we must "remember" an open symbol (e.g., `(`, `[`, or `{`) until it is "taken care of" later by matching a corresponding close symbol (e.g., `)`, `]`, or `}`, respectively). When the grouping symbols in an expression are properly matched, computer scientists say that the expression is *well formed* and that the grouping symbols are *balanced*.

Given a set of grouping symbols, our problem is to determine whether the open and close versions of each symbol are matched correctly. We'll focus on the normal pairs: `()`, `[]`, and `{}`. In theory, of course, 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 open grouping symbol, and each open grouping symbol must have a matching close symbol. Thus, matching symbols can be unbalanced for two reasons: There is a mismatching close symbol (e.g., `{]`) or there is a missing close symbol (e.g., `{{[]}`). Figure 3.7 shows examples of both well-formed and ill-formed expressions.

The Balanced Class

To help solve our problem we create a class called `Balanced`, with a single exported method `test` that takes an expression as a string argument and checks whether the grouping symbols in the expression are balanced. As there are two ways that an expression can fail the balance test, there are three possible results. We use an integer to indicate the result of the test:

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`

Well-Formed Expressions	Ill-Formed Expressions
(xx (xx ( )) xx)	(xx (xx ( )) xxx) xxx)
[ ] ( ) { }	] [
([ ] { xxx } xxx ( ) xxx)	(xx [xxx) xx]
([{ [(([{ x }]) x)] } x])	([{ [(([{ x }]) x)] } x })
xxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxx {

Figure 3.7 *Well-formed and ill-formed expressions*

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

We include a single constructor for the `Balanced` class. To make the class more generally usable, we allow the application to specify the open and close symbols. We thus define two string parameters for the constructor, `openSet` and `closeSet`, through which the user can pass the symbols. The symbols in the two sets match up by position. For our specific problem the two arguments could be "`([{`" and "`)]}`."

It is important that each symbol in the combined open and close sets is unique and that the sets be the same size. Otherwise, it is impossible to determine matching criteria. We use programming by contract and state these criteria in a precondition of the constructor.

```
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;
}
```

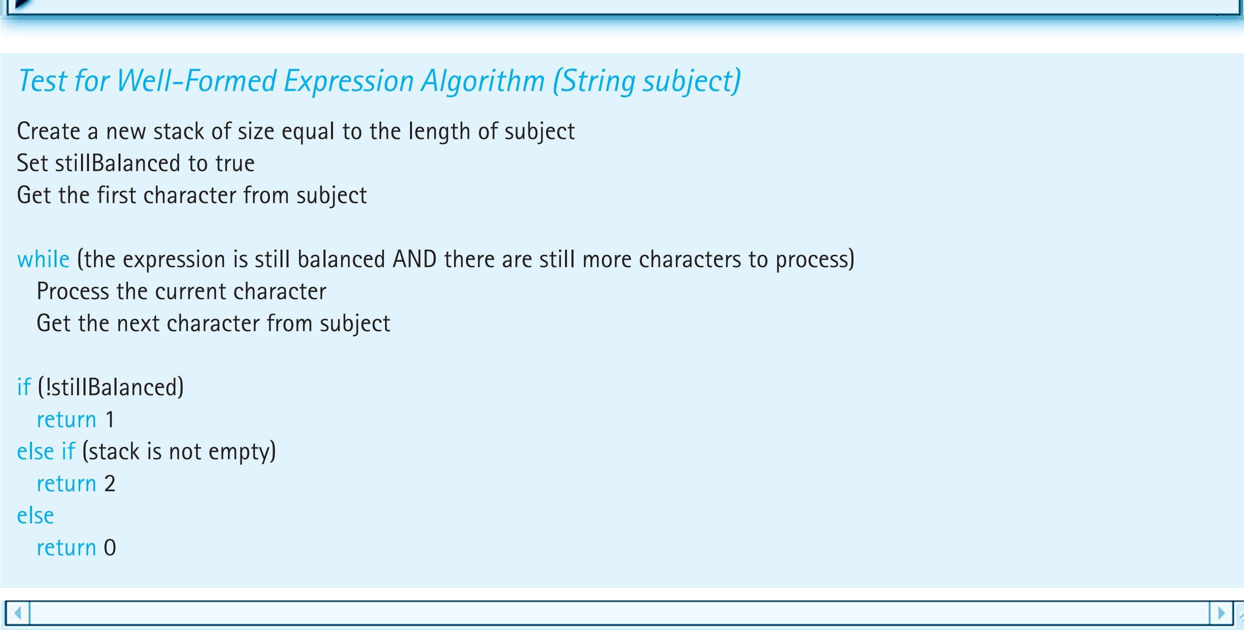
Now we turn our attention to the `test` method. It is passed a `String` argument through its `subject` parameter and must determine, based on the characters in `openSet` and `closeSet`, whether the symbols in `subject` are balanced. The method processes the characters in `subject` one at a time. For each character, it performs one of three tasks:

- If the character is an open symbol, it is pushed on the stack.
- If the character is a close symbol, it is 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.

The stack is the appropriate data structure in which to save the open symbols because we always need to examine the most recent one. When all of the characters have been processed, the stack should be empty—otherwise, there are open symbols left over.

Now we are ready to write the main algorithm for `test`. We assume an instance of a Stack ADT as defined by `BoundedStackInterface`. We use a bounded stack because we know the stack cannot contain more elements than the number of characters in the expression. We also declare a `boolean` variable `stillBalanced`, initialized to `true`, to record whether the expression, as processed so far, is balanced.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION



Test for Well-Formed Expression Algorithm (String subject)

```
Create a new stack of size equal to the length of subject
Set stillBalanced to true
Get the first character from subject

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

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

The part of this algorithm that requires expansion before moving on to the coding stage is the "Process the current character" command. We previously described how to handle each type of character. Here are those steps in algorithmic form:

```
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
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

The code for the `Balanced` class is listed next. Because the focus of this chapter is stacks, we have emphasized the calls to the stack operations in the code listing. There are several interesting things to note about the `Balanced` class:

1. We declare our stack to be of type `BoundedStackInterface`, but instantiate it as class `ArrayStack`, following the convention suggested in "Using the StringLogInterface" at the end of Section 2.2.
2. We use a shortcut for determining whether a close symbol matches an open symbol. According to our rules, the symbols match if they share the same relative position in their respective sets. This means that when we encounter an open special symbol, rather than save the actual character on the stack, we can push its *position* in the `openSet` string onto the stack. Later in the processing, when we encounter a close symbol, we can just compare its position with the position value on the stack. Thus, rather than push a character value onto the stack, we push an integer value.
3. We instantiate our stacks to hold elements of type `Integer`. But, as just mentioned, in the `test` method we push elements of the primitive type `int` onto our stack. How can this be? As of Java 5.0, Java includes a feature called *Autoboxing*. If a programmer uses a value of a primitive type as an `Object`, it is automatically converted (boxed) into an object of its corresponding wrapper class. So when the `test` method says

   ```
   stack.push(openIndex);
   ```

   the integer value of `openIndex` is automatically converted to an `Integer` object before being stored on the stack. In previous versions of Java we would have needed to state the conversion explicitly:

   ```
   Integer openIndexObject = new Integer(openIndex);
   stack.push(openIndexObject);
   ```

4. A corresponding feature, introduced with Java 5.0, called *Unboxing*, reverses the effect of the *Autoboxing*. When we access the top of the stack with the statement

   ```
   openIndex = stack.top();
   ```

   the `Integer` object at the top of the stack is automatically converted to an integer value. In previous versions of Java we would have needed to write

   ```
   openIndex = stack.top().intValue();
   ```

5. In processing a closing symbol, we access the stack to see if its top holds the corresponding opening symbol. If the stack is empty, it indicates an unbalanced expression. We have two ways to check whether the stack is empty: We can use the `isEmpty` method or we can try to access the stack and catch a `StackUnderflowException`. We choose the latter approach. It seems to fit the spirit of the algorithm, because we expect to find the open symbol and finding the stack empty is the "exceptional" case.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

6. In contrast, we use `isEmpty` to check for an empty stack at the end of processing the expression. Here, we don't want to extract an element from the stack—we just need to know whether it is empty.

Here is the code for the entire class:

```
//----------------------------------------------------------------
// Balanced.java       by Dale/Joyce/Weems             Chapter 3
//
// Checks for balanced expressions using standard rules.
//
// Matching pairs of open and close symbols are provided to the
// constructor through two string parameters.
//----------------------------------------------------------------

import ch03.stacks.*;

public class Balanced
{
  private String openSet;
  private String closeSet;

  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;
  }

  public int test(String expression)
  // Returns 0 if expression is balanced.
  // Returns 1 if expression has unbalanced symbols.
  // Returns 2 if expression came to end prematurely.
  {
    char currChar;          // current expression character being studied
    int  currCharIndex;     // index of current character
    int  lastCharIndex;     // index of last character in the expression

    int openIndex;          // index of current character in openSet
    int closeIndex;         // index of current character in closeSet

    boolean stillBalanced = true;   // true as long as expression is balanced
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
// holds unmatched open symbols
BoundedStackInterface<Integer> stack;
stack = new ArrayStack<Integer>(expression.length());

currCharIndex = 0;
lastCharIndex = expression.length() - 1;

while (stillBalanced && (currCharIndex <= lastCharIndex))
// while expression still balanced and not at end of expression
{
  currChar = expression.charAt(currCharIndex);
  openIndex = openSet.indexOf(currChar);

  if(openIndex != -1)   // if the current character is in the openSet
  {
    // Push the index onto the stack.
    stack.push(openIndex);
  }
    else
    {
      closeIndex = closeSet.indexOf(currChar);
      if(closeIndex != -1) // if the current character is in the closeSet
      {
        try                     // try to pop an index off the stack
        {
          openIndex = stack.top();
          stack.pop();

          if (openIndex != closeIndex)   // if popped index doesn't match
          {
            stillBalanced = false;       // then expression not balanced
          }
        }
        catch(StackUnderflowException e) // if stack was empty
        {
          stillBalanced = false;         // then expression not balanced
        }
      }
    }
    currCharIndex++;              // set up processing of next character
  }

  if (!stillBalanced)
    return 1;               // unbalanced symbols
  else
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
      if (!stack.isEmpty())
        return 2;             // premature end of expression
      else
        return 0;             // expression is balanced
  }
}
```

The Application

Now that we have the `Balanced` class, it is not difficult to finish our application. Of course, we should carefully test the class first—but in this case we can use our application as a test driver.

Because the `Balanced` class is responsible for determining whether grouping symbols are balanced, all that remains is to implement the user input and output. Rather than processing just one expression, we allow the user to enter a series of expressions, asking whether he or she wishes to continue after each result is output. Following our standard approach, we implement a console-based system. It is straightforward to convert this application to a GUI, if you are familiar with Java's Swing classes. We call our program `BalancedApp`. Note that when the `Balanced` class is instantiated, the constructor is passed the strings "`({[`" "`)]}`" so that it corresponds to our specific problem.

```
//----------------------------------------------------------------------
// BalancedApp.java        by Dale/Joyce/Weems                 Chapter 3
//
// Checks for balanced grouping symbols.
// Input consists of a sequence of expressions, one per line.
// Special symbol types are (), [], and {}.
//----------------------------------------------------------------------

import java.util.Scanner;

public class BalancedApp
{
  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);

    // Instantiate new Balanced class with grouping symbols.
    Balanced bal = new Balanced("([{", ")]}");

    int result;                   // 0 = balanced, 1 = unbalanced,
                                  // 2 = premature end

    String expression = null;     // expression to be evaluated
    String more = null;           // used to stop or continue processing
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
    do
    {
      // Get next expression to be processed.
      System.out.println("Enter an expression to be evaluated: ");
      expression = conIn.nextLine();

      // Obtain and output result of balanced testing.
      result = bal.test(expression);
      if (result == 1)
        System.out.println("Unbalanced symbols ");
      else
      if (result == 2)
        System.out.println("Premature end of expression");
      else
        System.out.println("The symbols are balanced.");

      // Determine if there is another expression to process.
      System.out.println();
      System.out.print("Evaluate another expression? (Y=Yes): ");
      more = conIn.nextLine();
      System.out.println();
    }
    while (more.equalsIgnoreCase("y"));
  }
}
```

Here is the output from a sample run:

```
Enter an expression to be evaluated:
(xx[yy]{ttt})
The symbols are balanced.

Evaluate another expression? (Y=Yes): Y

Enter an expression to be evaluated:
((())
Premature end of expression

Evaluate another expression? (Y=Yes): Y

Enter an expression to be evaluated:
(ttttttt]
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

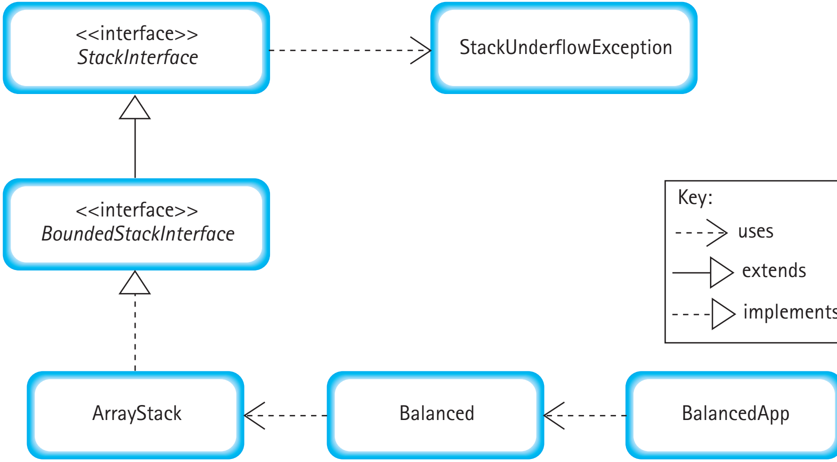


Figure 3.8 *Program architecture*

```
Unbalanced symbols

Evaluate another expression? (Y=Yes): Y

Enter an expression to be evaluated:
(){}[][({{[{({})}]}})]
The symbols are balanced.

Evaluate another expression? (Y=Yes): n
```

Figure 3.8 is a UML diagram showing the relationships among our interfaces and classes used in this program. The intent is to show the general architecture, so we do not include details about attributes and operations.

3.7 Link-Based Implementation

In Chapter 2 we introduced linked lists and explained how they provide an alternative to arrays when implementing collections. It is important for you to learn both approaches. Recall that a "link" is the same thing as a "reference." The Stack ADT implementation presented in this section is therefore referred to as a reference- or link-based implementation.

Figure 3.17, in the Summary on page 230, shows the relationships among the primary classes and interfaces created to support our Stack ADT, including those developed in this section.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

The LLNode Class

Recall from Chapter 2 that to create a linked list we needed to define a self-referential class to act as the nodes of the list. Our approach there was to define the `LLStringNode` class that allowed us to create a linked list of strings. If we merely needed to create stacks of strings, we could reuse the `LLStringNode` class to support our link-based implementation of stacks. However, the class of objects held by our stacks must be parametizable—it will be specified by the client whenever a stack is instantiated. Therefore we define a class that is analogous to the `LLStringNode` class called `LLNode`. Figure 3.9 shows the corresponding UML class diagram. The self-referential nature of the class is evident from the fact that an `LLNode` has an instance variable, `link`, of class `LLNode`. Because we plan to use this class to support our development of several data structures, we place it in a package named `support`.[8]

The implementation of the `LLNode` class is essentially the same as that of the `LLStringNode` class. The `info` variable contains a reference to the object of class `T` representing the information held by the node, and the `link` variable holds a reference to the next `LLNode` on the list. The code includes a constructor that accepts an object of class `T` as an argument and constructs a node that references that object. An abstract view of a single `LLNode` is pictured in Figure 3.10. The code also includes setters and getters for both the `info` and `link` attributes so that we can create, manipulate, and traverse the linked list of `T` nodes.

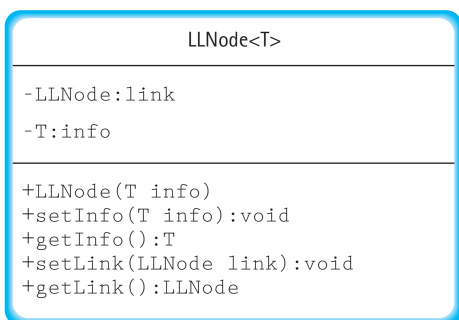


Figure 3.9 *UML class diagram of* `LLNode`

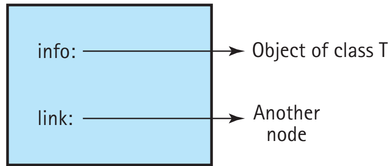


Figure 3.10 *A single node*

8. The `LLNode` class file can be found in the `support` subdirectory of the `bookFiles` directory that contains the program files associated with the textbook.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
//----------------------------------------------------------------------------
// LLNode.java              by Dale/Joyce/Weems                     Chapter 3
//
// Implements <T> nodes for a linked list.
//----------------------------------------------------------------------------

package support;

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

  public LLNode(T info)
  {
    this.info = info;
    link = null;
  }

  public void setInfo(T info)
  // Sets info of this LLNode.
  {
    this.info = info;
  }

  public T getInfo()
  // Returns info of this LLNode.
  {
    return info;
  }

  public void setLink(LLNode link)
  // Sets link of this LLNode.
  {
    this.link = link;
  }

  public LLNode getLink()
  // Returns link of this LLNode.
  {
    return link;
  }
}
```

The LinkedStack Class

We call our new stack class `LinkedStack`, to differentiate it from the array-based classes of the previous section. `LinkedStack` implements the `UnboundedStackInterface`.

We need to define only one instance variable in the `LinkedStack` class, to hold a reference to the linked list of objects that represents the stack. Because we just need

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

quick access to the top of the stack, we maintain a reference to the node representing the most recent element pushed onto the stack. That node will, in turn, hold a reference to the node representing the next most recent element. That pattern continues until a particular node holds a null reference in its `link` attribute, signifying the bottom of the stack. We call the original reference variable `top`, as it will always reference the top of the stack. It is a reference to a `LLNode`. When we instantiate an object of class `LinkedStack`, we create an empty stack by setting `top` to `null`. The beginning of the class definition is shown here. Note the `import` statement that allows us to use the `LLNode` class.

```
//----------------------------------------------------------------------------
// 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;
  }
. . .
```

Now, let's see how we implement our link-based stack operations.

The push Operation

Pushing an element onto the stack means creating a new node and linking it to the current chain of nodes. Figure 3.11 shows the result of the sequence of operations listed here. It graphically demonstrates the dynamic allocation of space for the references to the stack elements. Assume A, B, and C represent objects of class `String`.

```
UnboundedStackInterface<String> myStack;
myStack = new LinkedStack<String>();
myStack.push(A);
myStack.push(B);
myStack.push(C);
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

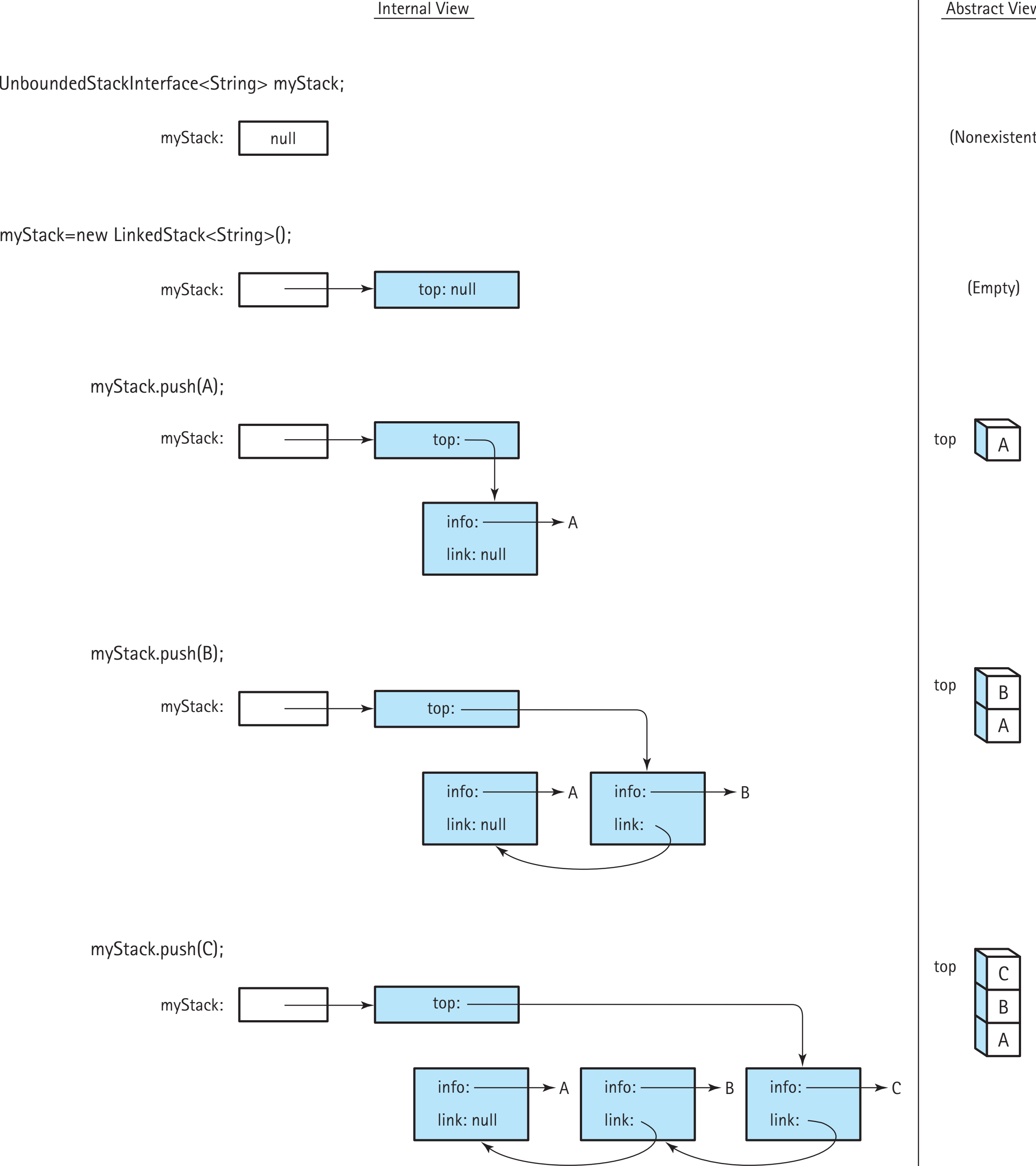


Figure 3.11 *Results of stack operations using* `LLNode`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

When performing the push operation we must allocate space for each new node dynamically. Here is the general algorithm:

push(element)

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

Figure 3.12 graphically displays the effect of each step of the algorithm, starting with a stack that already contains A and B and showing what happens when C is pushed onto it. This is the same algorithm we studied previously in Section 2.5 for insertion into the beginning of a linked list. We have arranged the node boxes visually to emphasize the last in, first out nature of a stack.

Let's look at the algorithm line by line, creating our code as we go. Follow our progress through both the algorithm and Figure 3.12 during this discussion. We begin by allocating space for a new stack node and setting its info attribute to the element:

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

Thus, newNode is a reference to an object that contains two attributes: info of class T and link of the class LLNode. The constructor has set the info attribute to reference element, as required. Next we need to set the value of the link attribute:

```
newNode.setLink(top);
```

Now info references the element pushed onto the stack, and link references the previous top of stack. Finally, we need to reset the top of the stack to reference the new node:

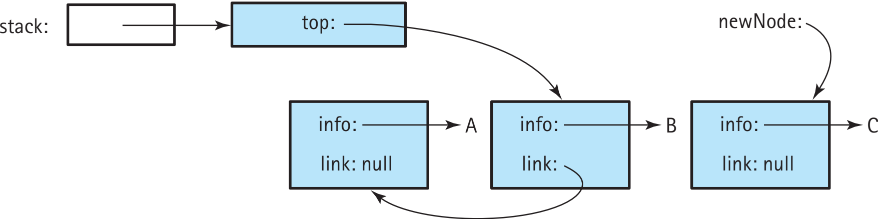
```
top = newNode;
```

Putting it all together, the code for the push method is

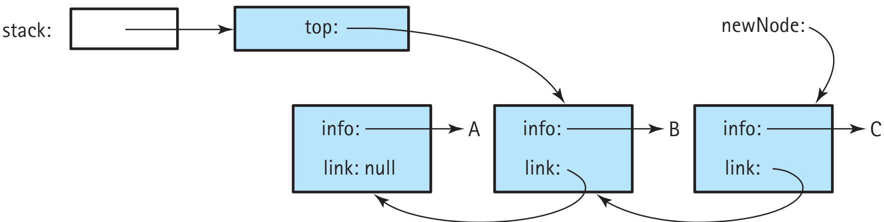
```
public void push(T element)
// Places element at the top of this stack.
{
  LLNode<T> newNode = new LLNode<T>(element);
  newNode.setLink(top);
  top = newNode;
}
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

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

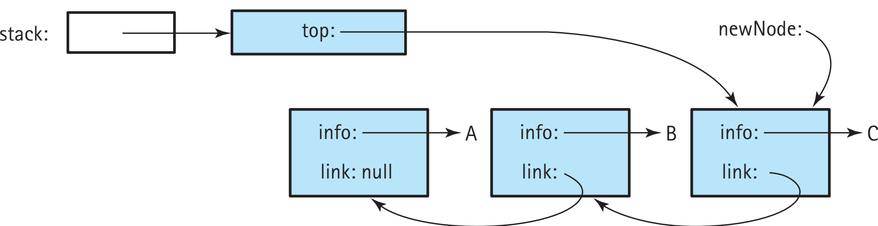


Figure 3.12 *Results of `push` operation*

Note that the order of these tasks is critical. If we reset the `top` variable before setting the `link` of the new node, we would lose access to the stack nodes! This situation is generally true when we are dealing with a linked structure: You must be very careful to change the references in the correct order, so that you do not lose access to any of the data.

You have seen how the algorithm works on a stack that contains elements. What happens if the stack is empty? Although we verified in Section 2.5 that our approach works in this case, let's trace through it again. Figure 3.13 shows graphically what occurs.

Space is allocated for the new node and the node's `info` attribute is set to reference `element`. Now we need to correctly set the various links. The `link` of the new node is assigned the value of `top`. What is this value when the stack is empty? It is `null`, which is exactly what we want to put into the `link` of the last (bottom) node of a linked stack. Then `top` is reset to point to the new node, making the new node the top of the stack. The result is exactly what we would expect—the new node is the only node on the linked list and it is the current top of the stack.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

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

stack: → top: null

newNode: → info: → A; link: null

Set the node link to the previous top of stack

stack: → top: null

newNode: → info: → A; link: null

Set the top of stack to the new stack node

stack: → top: → (node)

newNode: → info: → A; link: null

Figure 3.13 *Results of* `push` *operation on an empty stack*

The pop Operation

The `pop` operation is equivalent to deleting the first node of a linked list. It is essentially the reverse of the `push` operation.

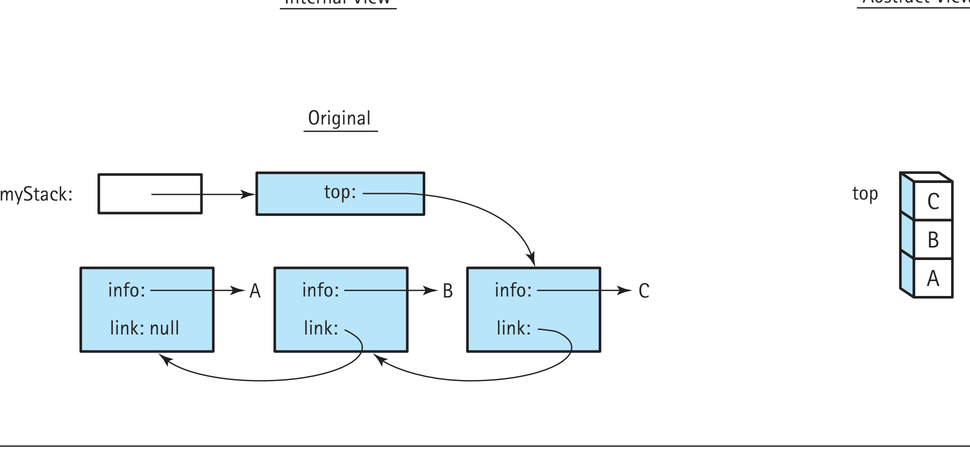
To accomplish it we simply reset the stack's `top` variable to reference the node that represents the next element. That is all we really have to do. Resetting `top` to the next stack node effectively removes the top element from the stack. See Figure 3.14. This requires only a single line of code:

```
top = top.getLink();
```

The assignment copies the reference from the `link` attribute of the top stack node into the variable `top`. After this code is executed, `top` refers to the `LLNode` object just below the prior top of the stack. We can no longer use `top` to reference the previous top object, because we overwrote our only reference to it.

As indicated in Figure 3.14, the former top of the stack is labeled as garbage; the system garbage collector will eventually reclaim the space. If the `info` attribute of this object is the only reference to the data object labeled C in the figure, it, too, is garbage and its space will be reclaimed.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION



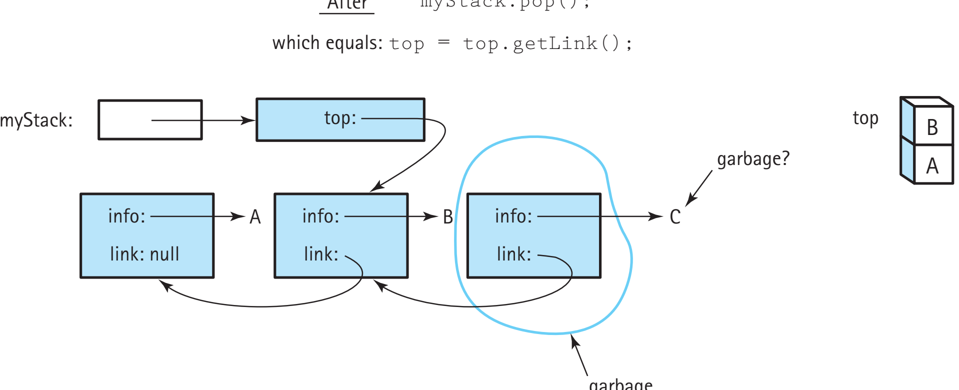


Figure 3.14 *Results of `pop` operation*

Are there any special cases to consider? Given that we are removing an element from the stack, we should be concerned with empty stack situations. What happens if we try to pop an empty stack? In this case the `top` variable contains `null` and the assignment statement "`top = top.getLink;`" results in a run-time error: `NullPointerException`. To control this problem ourselves, we protect the assignment statement using the Stack ADT's `isEmpty` operation. The code for our `pop` method is shown next.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


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

We use the same `StackUnderflowException` we used in our array-based approaches.

There is one more special case—popping from a stack with only one element. We need to make sure that this operation results in an empty stack. Let's see if it does. When our stack is instantiated, `top` is set to `null`. When an element is pushed onto the stack, the `link` of the node that represents the element is set to the current `top` variable; therefore, when the first element is pushed onto our stack, the `link` of its node is set to `null`. Of course, the first element pushed onto the stack is the last element popped off. This means that the last element popped off the stack has an associated `link` value of `null`. Because the `pop` method sets `top` to the value of this `link` attribute, after the last value is popped `top` again has the value `null`, just as it did when the stack was first instantiated. We conclude that the `pop` method works for a stack of one element. Figure 3.15 graphically depicts pushing a single element onto a stack and then popping it off.

The Other Stack Operations

Recall that the `top` operation simply returns a reference to the top element of the stack. At first glance this might seem very straightforward. Simply code

```
return top;
```

as `top` references the element on the top of the stack. However, remember that `top` references an `LLNode` object. Whatever program is using the Stack ADT is not concerned about `LLNode` objects. The client program is only interested in the object that is referenced by the `info` variable of the `LLNode` object.

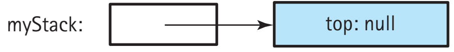
Let's try again. To return the `info` of the top `LLNode` object we code

```
return top.getInfo();
```

That's better, but we still need to do a little more work. What about the special case when the stack is empty? In that situation we need to throw an exception instead of returning an object. The final code for the `top` method is shown next.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Empty Stack:



After `myStack.push(A);`

myStack:
top:
info:
A
link: null

And then `myStack.pop();`

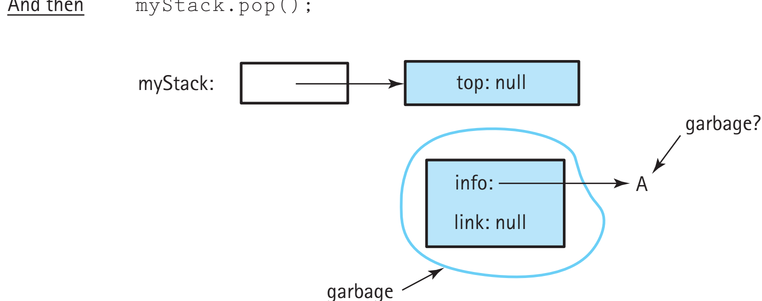


Figure 3.15 *Results of `push`, then `pop` on an empty stack*

```
public T top()
// Throws StackUnderflowException if this stack is empty,
// otherwise returns top element from this stack.
{
  if (!isEmpty())
    return top.getInfo();
  else
    throw new StackUnderflowException("Top attempted on an empty stack.");
}
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

That wasn't bad, but the `isEmpty` method is even easier. If we initialize an empty stack by setting the `top` variable to `null`, then we can detect an empty stack by checking for the value `null`.

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

An even simpler way of writing this is

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

The linked implementation of the Stack ADT can be tested using the same test plan that was presented for the array-based version, except we would not have to test an `isFull` operation.

Comparing Stack Implementations

Let's compare our two classic implementations of the Stack ADT, `ArrayStack` and `LinkedStack`, in terms of storage requirements and efficiency of the algorithms. First we consider the storage requirements. An array that is instantiated to match the maximum stack size takes the same amount of memory, no matter how many array slots are actually used. The linked implementation, using dynamically allocated storage, requires space only for the number of elements actually on the stack at run time. Note, however, that the elements are larger because we must store the reference to the next element as well as the reference to the user's data.

We now compare the relative execution "efficiency" of the two implementations in terms of Big-O notation. The implementations of `isFull` and `isEmpty`, where required, are clearly O(1); they always take a constant amount of work. What about `push`, `pop`, and `top`? Does the number of elements in the stack affect the amount of work required by these operations? No, it does not. In both implementations, we directly access the top of the stack, so these operations also take a constant amount of work. They, too, have O(1) complexity.

Only the class constructor differs from one implementation to the other in terms of the Big-O efficiency. In the array-based implementation, when the array is instantiated, the system creates and initializes each of the array locations. As it is an array of objects, each array slot is initialized to `null`. The number of array slots is equal to the maximum number of possible stack elements. We call this number N and say that the array-based constructor is $O(N)$. For the linked approach, the constructor simply sets the `top` variable to `null`, so it is only O(1).

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Overall the two stack implementations are roughly equivalent in terms of the amount of work they do.

So, which is better? The answer, as usual, is "It depends." 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. Why would we ever want to use the array-based implementation? Because it's short, simple, and efficient. If pushing occurs frequently, the array-based implementation executes faster than the link-based implementation because it does not incur the run-time overhead of repeatedly invoking the `new` operation. When the maximum size is small and we know the maximum size with certainty, the array-based implementation is a good choice.

3.8 Case Study: Postfix Expression Evaluator

Postfix notation[9] is a notation for writing arithmetic expressions in which the operators appear after their operands. For example, instead of writing

$$(2 + 14) \times 23$$

we write

$$2 \;\; 14 + \;\; 23 \times$$

With postfix notation, there are no precedence rules to learn, and parentheses are never needed. Because of this simplicity, some popular handheld calculators of the 1980s used postfix notation to avoid the complications of the multiple parentheses required in traditional algebraic notation. Postfix notation is also used by compilers for generating nonambiguous expressions.

In this case study, we create a computer program that evaluates postfix expressions.

Discussion

In elementary school you learned how to evaluate simple expressions that involve the basic binary operators: addition, subtraction, multiplication, and division. These are called *binary operators* because they each operate on two operands. It is easy to see how a child would solve the following problem:

$$2 + 5 = ?$$

---

9. Postfix notation is also known as reverse Polish notation (RPN), so named after the Polish logician Jan Lukasiewicz (1875–1956) who developed it.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

As expressions become more complicated, the pencil-and-paper solutions require a little more work. Multiple tasks must be performed to solve the following problem:

$$(((13 - 1) / 2) \times (3 + 5)) = ?$$

These expressions are written using a format known as *infix* notation, which is the same notation used for expressions in Java. The operator in an infix expression is written *in* between its operands. When an expression contains multiple operators such as

$$3 + 5 \times 2$$

we need a set of rules to determine which operation to carry out first. You learned in your mathematics classes that multiplication is done before addition. You learned Java's operator-precedence rules[10] in your first Java programming course. We can use parentheses to override the normal ordering rules. Still, it is easy to make a mistake when writing or interpreting an infix expression containing multiple operations.

Evaluating Postfix Expressions

Postfix notation is another format for writing arithmetic expressions. In this notation, the operator is written after (*post*) the two operands. Here are some simple postfix expressions and their results:

Postfix Expression	Result
4 5 +	9
9 3 /	3
17 8 −	9

The rules for evaluating postfix expressions with multiple operators are much simpler than those for evaluating infix expressions; simply perform the operations from left to right. Now, let's look at a postfix expression containing two operators.

$$6\ 2\ /\ 5\ +$$

We evaluate the expression by scanning from left to right. The first item, 6, is an operand, so we go on. The second item, 2, is also an operand, so again we continue. The third item is the division operator. We now apply this operator to the two previous operands. Which of the two saved operands is the divisor? The one we saw most

10. See Appendix B, Java Operator Precedence.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

recently. We divide 6 by 2 and substitute 3 back into the expression, replacing 6 2 /. Our expression now looks like this:

3 5 +

We continue our scanning. The next item is an operand, 5, so we go on. The next (and last) item is the operator +. We apply this operator to the two previous operands, obtaining a result of 8.

Here's another example:

5 7 + 6 2 − ×

As we scan from left to right, the first operator we encounter is +. Applying this to the two preceding operands (5 and 7), we obtain the expression

12 6 2 − ×

The next operator we encounter is −, so we subtract 2 from 6, obtaining

12 4 ×

We apply the last operator, ×, to its two preceding operands and obtain our final answer: 48.

Here are some more examples of postfix expressions containing multiple operators, equivalent expressions in infix notation, and the results of evaluating them. See if you get the same results when you evaluate the postfix expressions.

Postfix Expression	Infix Equivalent	Result
4 5 7 2 + − ×	4 × (5 − (7 + 2))	−16
3 4 + 2 × 7 /	((3 + 4) × 2)/7	2
5 7 + 6 2 − ×	(5 + 7) × (6 − 2)	48
4 2 3 5 1 − + × + ×	?×(4 + (2 × (3 + (5 − 1))))	not enough operands
4 2 + 3 5 1 − × +	(4 + 2) + (3 × (5 − 1))	18

Our task is to write a program that evaluates postfix expressions entered interactively from the keyboard. In addition to computing and displaying the value of an expression, our program must display error messages when appropriate ("not enough operands," "too many operands," and "illegal symbol"). Before we describe our specific requirements, let's look at the data structure and algorithm involved in postfix expression evaluation.

Postfix Expression Evaluation Algorithm

As so often happens, our by-hand algorithm can serve as a guideline for our computer algorithm. From the previous discussion, we see that there are two basic items in a post-

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

fix expression: operands (numbers) and operators. We access items (an operand or an operator) from left to right, one at a time. When the item we get is an operator, we apply it to the preceding two operands.

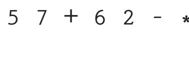
We must save previously scanned operands in a collection object of some kind. A stack is the ideal place to store the previous operands, because the top item is always the most recent operand and the next item on the stack is always the second most recent operand—just the two operands required when we find an operator. The following algorithm uses a stack to evaluate a postfix expression:

Evaluate Expression

```
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
```

Each iteration of the *while* loop processes one operator or one operand from the expression. When an operand is found, there is nothing to do with it (we haven't yet found the operator to apply to it), so we save it on the stack until later. When an operator is found, we get the two topmost operands from the stack, perform the operation, and put the result back on the stack; the result may be an operand for a future operator.

Let's trace this algorithm. Before we enter the loop, the input remaining to be processed and the stack look like this:



© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

After one iteration of the loop, we have processed the first operand and pushed it onto the stack.

```
5 7 + 6 2 - *      Stack: 5
```

After the second iteration of the loop, the stack contains two operands.

```
5 7 + 6 2 - *      Stack (top to bottom): 7, 5
```

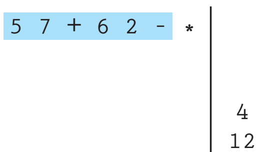
We encounter the + operator in the third iteration. We remove the two operands from the stack, perform the operation, and push the result onto the stack.

```
5 7 + 6 2 - *      Stack: 12
```

In the next two iterations of the loop, we push two operands onto the stack.

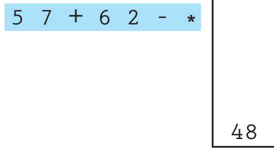
```
5 7 + 6 2 - *      Stack (top to bottom): 2, 6, 12
```

When we find the − operator, we remove the top two operands, subtract, and push the result onto the stack.

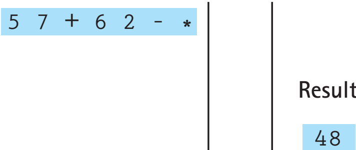


© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

When we find the * operator, we remove the top two operands, multiply, and push the result onto the stack.



Now that we have processed all of the items on the input line, we exit the loop. We remove the result, 48, from the stack.



Of course, we have glossed over a few "minor" details, such as how we recognize an operator and how we know when we are finished. All of the input values in this example were one-digit numbers. Clearly, this is too restrictive. We also need to handle invalid input. We discuss these challenges as we continue to evolve the solution to our problem.

Specification: Program Postfix Evaluation

Here is a more formal specification of our problem.

Function

The program evaluates postfix arithmetic expressions containing integers and the binary operators +, -, *, and /.

Interface

Following our established conventions, we are not specifying which type of interface the program should provide. We develop a console-based solution but also provide a GUI solution for your study. In either case, the program must allow the user to enter a postfix expression, have it evaluated, and see the results of the evaluation. The user should then have the option of entering additional expressions or ending the program.

Input

The input is a series of arithmetic postfix expressions, entered interactively from the keyboard. An expression is made up of operators (the characters +, -, *, and /) and integers (the operands). Operators and operands must be separated by at least one blank.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Data

All numbers input, manipulated, and output by the program are integers.

Output

After the evaluation of each expression, the results are displayed:

"Result = value"

Error Processing

The program should recognize illegal postfix expressions. Instead of displaying an integer result when the expression is entered, in such a case it should display error messages as follows:

Type of Illegal Expression	Error Message
An expression contains a symbol that is not an integer or not one of "+", "-", "*", and "/"	Illegal symbol
An expression requires more than 50 stack items	Too many operands—stack overflow
There is more than one operand left on the stack after the expression is processed; for example, the expression 5 6 7 + has too many operands	Too many operands—operands left over
There are not enough operands on the stack when it is time to perform an operation; for example, 6 7 + + +; and, for example, 5 + 5	Not enough operands—stack underflow

Assumptions

1. The operations in expressions are valid at run time. This means that we do not try to divide by zero. Also, we do not generate numbers outside of the range of the Java `int` type.
2. A postfix expression has a maximum of 50 operands.

Brainstorming and Filtering

A study of the specifications provides the following list of nouns that appear to be possibilities for classes: postfix arithmetic expressions, operators, result, operands, and error messages. Let's look at each in turn.

- The *postfix arithmetic expressions* are entered by the user and consist of both numbers and other characters. We conclude that an expression should be represented by a string.
- This means we can probably represent *operators* as strings, too. Another possibility is to hold the operators in an ADT that provides a "set" of characters. How-

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

ever, upon reflection, we realize that all we really have to do is recognize the operator character, and the built-in string and character operations we already have at our disposal should be sufficient.

- The *result* of an evaluation is an interesting case. Where does the result come from? We propose the creation of a separate class `PostFixEvaluator` that provides an `evaluate` method that accepts a postfix expression as a string and returns the value of the expression. Our main program will use this class (and a few others) to solve the problem.
- The *operands* are integers.
- The *error messages* we need to generate are all related to the evaluation of the postfix expression. Because the `PostFixEvaluator` class evaluates the postfix expression, it will discover the errors. Therefore, to communicate the error messages between `PostFixEvaluator` and the main program, we propose the creation of an exception class called `PostFixException`.

From our knowledge of the postfix expression evaluation algorithm we know we also need a stack. We decide to use our `ArrayStack` class, which implements the `BoundedStackInterface`, because the problem description places an upper bound of 50 on the size of the stack. Additionally, we intend to use our standard approach and create a main program that provides interaction with the user.

Let's look at a short scenario describing how these classes can be used to solve our problem. The main program will prompt the user for an expression and read it into a string variable. It can then pass this string to the `evaluate` method of the `PostFixEvaluator` class, which will use an `ArrayStack` object to help determine the value of the expression, assuming it is a legal expression. The `evaluate` method is used within a *try-catch* statement that allows the main program to determine whether any `PostFixException` exceptions have been thrown. In either case it reports the result to the user and prompts for another expression. We can proceed with confidence that our set of classes seems sufficient to solve the problem.

We now move on to the design, implementation, and testing of the classes. Note that we can test the classes together, once they have all been created, by evaluating a number of postfix expressions (both legal and illegal) with the application.

Evolving a Program

We present our case studies in an idealized fashion. We make a general problem statement; discuss it; define formal specifications; identify classes; design and code the classes; and then test the system. In reality, however, such an application would probably evolve gradually, with small unit tests performed along the way. Especially during design and coding, it is sometimes helpful to take smaller steps and to evolve your program rather than trying to create it all at once. For example, for this case study you could take the following steps:

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

1. Build a prototype of the main program that just provides input/output activity—it would not support any processing. Its purpose is to test the usability of the user interface and provide a driver for further development.
2. Build a small part of `PostFixEvaluator` and see if you can pass it a string from the interface at the appropriate time.
3. See if you can pass back some information—any information—about the string from `PostFixEvaluator` to the main program and have it display on the user interface. For example, you could display the number of tokens in the string.
4. Upgrade `PostFixEvaluator` so that it recognizes operands and transforms them into integers. Have it obtain an operand from the expression string, transform it, push the integer onto a stack, retrieve it, and pass it back for display.
5. Upgrade `PostFixEvaluator` to recognize operators and process expressions that are more complicated. Test some legal expressions.
6. Add the error trapping and reporting portion. Test using illegal expressions.

Devising a good program evolution plan is often the key to successful programming.

The PostFixEvaluator Class

The purpose of this class is to provide an `evaluate` method that accepts a postfix expression as a string and returns the value of the expression. We do not need any objects of the class, so we implement `evaluate` as a `public static` method. This means that it is invoked through the class itself, rather than through an object of the class.

The `evaluate` method must take a postfix expression as a string argument and return the value of the expression. The code for the class is listed below. It follows the basic postfix expression algorithm that we developed earlier, using an `ArrayStack` object to hold operands of class `Integer` until they are needed. Note that it instantiates a `Scanner` object to "read" the string argument and break it into tokens.

Let's consider error message generation. Look through the code for the lines that throw `PostFixException` exceptions. You should be able to see that we cover all of the error conditions required by the problem specification. As would be expected, the error messages directly related to the stack processing are all protected by *if* statements that check whether the stack is empty (not enough operands) or full (too many operands). The only other error trapping occurs if the string stored in operator does not match any of the legal operators, in which case we throw an exception with the message "Illegal symbol."

```
//----------------------------------------------------------------------
// PostFixEvaluator.java       by Dale/Joyce/Weems           Chapter 3
//
// Provides a postfix expression evaluation.
//----------------------------------------------------------------------

package ch03.postfix;
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
import ch03.stacks.*;
import java.util.Scanner;

public class PostFixEvaluator
{
  public static int evaluate(String expression)
  {
    BoundedStackInterface<Integer> stack = new ArrayStack<Integer>(50);

    int value;
    String operator;

    int operand1;
    int operand2;

    int result = 0;

    Scanner tokenizer = new Scanner(expression);

    while (tokenizer.hasNext())
    {
      if (tokenizer.hasNextInt())
      {
        // Process operand.
        value = tokenizer.nextInt();
        if (stack.isFull())
          throw new PostFixException("Too many operands - stack overflow");
        stack.push(value);
      }
      else
      {
        // Process operator.
        operator = tokenizer.next();

        // Obtain second operand from stack.
        if (stack.isEmpty())
          throw new PostFixException("Not enough operands - stack " +
                                     "underflow");
        operand2 = stack.top();
        stack.pop();

        // Obtain first operand from stack.
        if (stack.isEmpty())
          throw new PostFixException("Not enough operands - stack " +
                                     "underflow");
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
        operand1 = stack.top();
        stack.pop();

        // Perform operation.
        if (operator.equals("/"))
          result = operand1 / operand2;
        else
        if(operator.equals("*"))
          result = operand1 * operand2;
        else
        if(operator.equals("+"))
          result = operand1 + operand2;
        else
        if(operator.equals("-"))
          result = operand1 - operand2;
        else
          throw new PostFixException("Illegal symbol: " + operator);

        // Push result of operation onto stack.
        stack.push(result);
      }
    }

    // Obtain final result from stack.
    if (stack.isEmpty())
      throw new PostFixException("Not enough operands - stack underflow");
    result = stack.top();
    stack.pop();

    // Stack should now be empty.
    if (!stack.isEmpty())
      throw new PostFixException("Too many operands - operands left over");

    // Return the final result.
    return result;
  }
}
```

The PFixConsole Class

This class is the main driver for our console-based application. Using the `PostFixEvaluator` and `PostFixException` classes, it is easy to design our program. We follow the same basic approach we used for `BalancedApp` earlier in the chapter—namely, prompt the user for an expression, evaluate it, return the results to the user, and ask the user if he or

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

she would like to continue. Note that the main program does not directly use `ArrayStack`; it is used strictly by the `PostFixEvaluator` class when evaluating an expression.

```
//----------------------------------------------------------------------
// PFixConsole.java         by Dale/Joyce/Weems                Chapter 3
//
// Evaluates postfix expressions entered by the user.
// Uses a console interface.
//----------------------------------------------------------------------

import java.util.Scanner;
import ch03.postfix.*;

public class PFixConsole
{
  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);

    String line = null;          // string to be evaluated
    String more = null;          // used to stop or continue processing

    int result;                  // result of evaluation

    do
    {
      // Get next expression to be processed.
      System.out.println("Enter a postfix expression to be evaluated: ");
      line = conIn.nextLine();

      // Obtain and output result of expression evaluation.
      try
      {
        result = PostFixEvaluator.evaluate(line);

        // Output result.
        System.out.println();
        System.out.println("Result = " + result);
      }
      catch (PostFixException error)
      {
        // Output error message.
        System.out.println();
        System.out.println("Error in expression - " + error.getMessage());
      }
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
        // Determine if there is another expression to process.
        System.out.println();
        System.out.print("Evaluate another expression? (Y = Yes): ");
        more = conIn.nextLine();
        System.out.println();
      }
      while (more.equalsIgnoreCase("y"));

      System.out.println("Program completed.");
    }
  }
```

Here is a sample run of our console-based application:

```
Enter a postfix expression to be evaluated:
5 7 + 6 2 - *

Result = 48

Evaluate another expression? (Y = Yes): y

Enter a postfix expression to be evaluated:
4 2 3 5 1 - + * + *

Error in expression - Not enough operands - stack underflow

Evaluate another expression? (Y = Yes): n

Program completed.
```

Testing the Postfix Evaluator

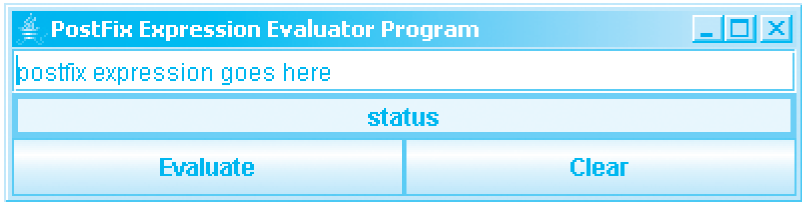
As mentioned earlier, we can test all of the classes created for this case study by simply running the postfix evaluator program and entering postfix expressions. We should test expressions that contain only additions, subtractions, multiplications, and divisions, as well as expressions that contain a mixture of operations. We should test expressions where the operators all come last and expressions where the operators are intermingled with the operands. Of course, we must evaluate all test expressions "by hand" to verify the correctness of the program's results. Finally, we must test that illegal expressions are correctly handled, as defined in the specifications. This includes a test of stack overflow, which requires at least 51 operands.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

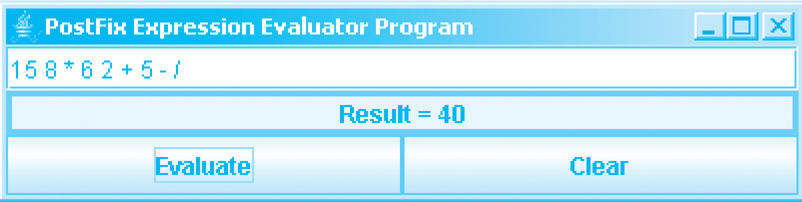
The GUI Approach

Most of the code in the `PFixConsole` program is responsible for presenting a console-based interface to the user. Just as that program used the `PostFixEvaluator` and `PostFixException` classes to do its primary processing, so can a program that presents a graphical user interface. Our `PFixGUI` program does just that. We do not list the code for this program here, but the interested reader can find it with the rest of the textbook code on the website. It uses the `Border` layout with nested containers.

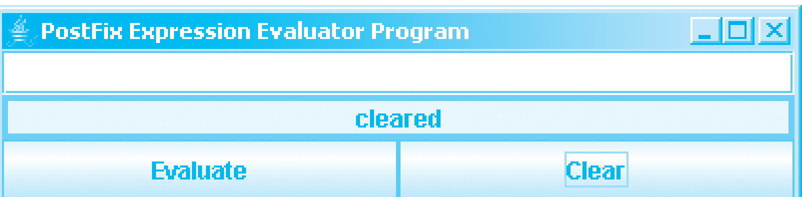
Here are a few screenshots from the running program. The first shows the interface as originally presented to the user:



Here's the result of a successful evaluation:



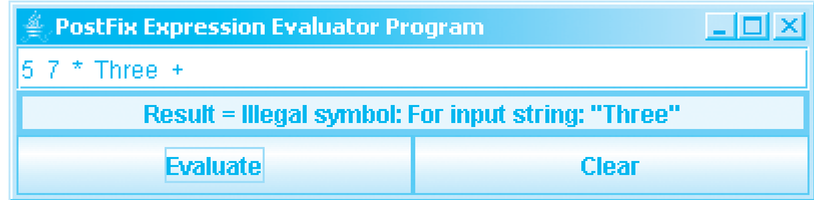
Next, the Clear button is clicked:



© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Here's what happens when the user enters an expression with too many operands:

And finally, here's what happens when an illegal operand is used:



Exercises

1. Revise and test the `PFixGUI` application to meet these specifications:

   a. Use `Flow` layout exclusively.

   b. Keep track of statistics about the numbers pushed onto the stack during the evaluation of an expression. The program should output the largest and smallest numbers pushed, how many numbers were pushed, and the average value of pushed numbers.

2. Revise and test the `PFixGUI` application so that it will step through the evaluation of a postfix expression one step at a time, showing the intermediate results as it goes. Include a Step button on the interface so the user can control when a step is taken. For example, if the original expression is `2 3 4 + + 5 -`, clicking Step once will display in the expression box `2 7 + 5 -`, clicking it again will display `9 5 -`, and clicking it one last time will display `4`.
3. Design and implement your own GUI for this problem. Write a short explanation about why your interface is better than the one shown in the textbook.

Figure 3.16 is a UML diagram showing the "uses" relationships among the stack implementation class, the postfix expression evaluation class, and the two main driver classes (one console-based and one GUI-based).

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Figure 3.16 *UML diagram for postfix program*

Summary

We have defined a stack at the logical level as an abstract data type, used a stack in two applications, and presented three implementations (array-based, link-based, and `ArrayList`-based). We have also seen how to use Java's generics to enable our stack implementations to work with different kinds of objects.

Although our logical picture of a stack is a linear collection of data elements with the newest element (the top) at one end and the oldest element at the other end, the physical representation of the stack class does not have to re-create our mental image. The implementation of the stack class must always support the last in, first out (LIFO) property; how this property is supported, however, is another matter.

Usually more than one functionally correct design is possible for the same data structure. When multiple correct solutions exist, the requirements and specifications of the application may determine which solution represents the best design.

We have seen how a hierarchy of interfaces can be used to represent the essential features of a data structure ADT and then extend it with different properties, such as being bounded or unbounded in size. We have also seen three different approaches to dealing with exceptional situations that are encountered within an ADT.

In this chapter we developed algorithms for two important applications of stacks in computer science. We can now check whether the grouping symbols in a string are balanced, and we can evaluate a postfix arithmetic expression.

Figure 3.17 is a UML diagram showing the stack-related interfaces and classes developed in this chapter, along with a few other supporting classes, and their relationships.

Exercises

3.1 Stacks

1. True or False?
   a. A stack is a first in, first out structure.
   b. The item that has been in a stack the longest is at the "bottom" of the stack.
   c. If you `push` five items onto an empty stack and then `pop` the stack five times, the stack will be empty again.
   d. If you `push` five items onto an empty stack and then perform the `top` operation five times, the stack will be empty again.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

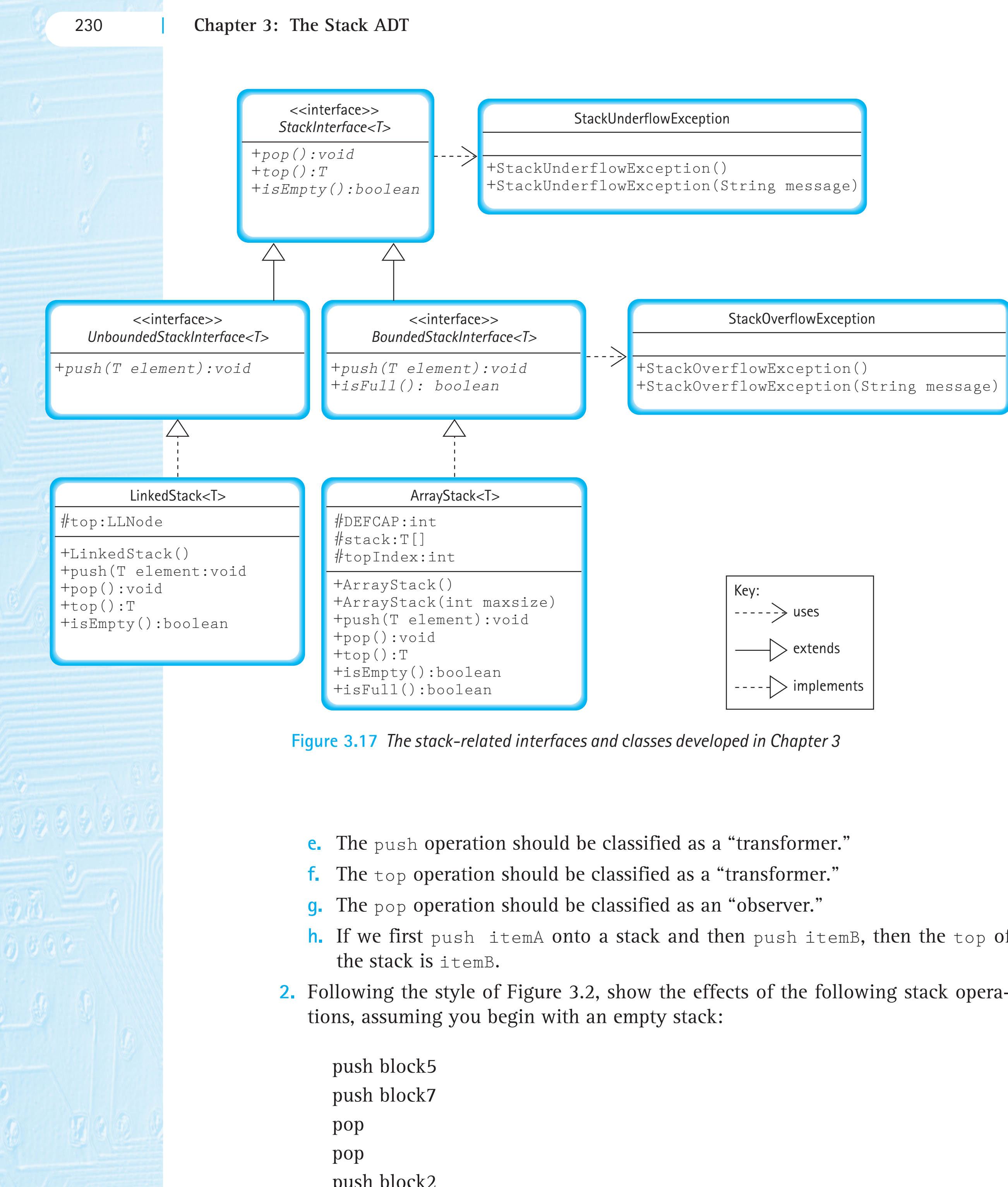


Figure 3.17 *The stack-related interfaces and classes developed in Chapter 3*

e. The `push` operation should be classified as a "transformer."

f. The `top` operation should be classified as a "transformer."

g. The `pop` operation should be classified as an "observer."

h. If we first `push itemA` onto a stack and then `push itemB`, then the `top` of the stack is `itemB`.

2. Following the style of Figure 3.2, show the effects of the following stack operations, assuming you begin with an empty stack:

push block5
push block7
pop
pop
push block2
push block1

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

pop
push block8

3.2 Collection Elements

3. In Section 3.2 we looked at four approaches to defining the types of elements we can hold in a collection ADT. Briefly describe each of the four approaches.

4. For each of the following programs that involve casting, predict the result of compiling and running the program. Potential answers include "there is a syntax error because . . . ," "there is a run-time error because . . . ," and "the output of the program would be . . . ."

   a.
```
public class test1
{
  public static void main(String[] args)
  {
    String s1, s2;
    Object o1;
    s2 = "E. E. Cummings";
    o1 = s2;
    s1 = o1;
    System.out.println(s1.toLowerCase());
  }
}
```

   b.
```
public class test2
{
  public static void main(String[] args)
  {
    String s1, s2;
    Object o1;
    s2 = "E. E. Cummings";
    o1 = s2;
    s1 = (String) o1;
    System.out.println(s1.toLowerCase());
  }
}
```

5. In Chapter 2 we developed a `StringLog` ADT. It represents a "log" that holds objects of class `String`. Suppose that instead of restricting ourselves to strings, we decided to create a "log" that holds objects of type `Object`. Describe the changes you would have to make to each of the following classes to implement such a change.

   a. Change `StringLogInterface` to `ObjectLogInterface`.

   b. Change `ArrayStringLog` to `ArrayObjectLog`.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

c. Change `ITDArrayStringLog` to `ITDArrayObjectLog`.

d. Change `LLStringNode` to `LLObjectNode`.

e. Change `LinkedStringLog` to `LinkedObjectLog`.

6. Create a generic array-based log class that features all the functionality of the `StringLog` class, as specified in Section 2.2. Create a driver application that demonstrates that your implementation works correctly.

7. Create a generic link-based `bag` class that features all the functionality of the `Bag` class specified in Exercise 29 of Chapter 2. Create a driver application that demonstrates that your implementation works correctly.

3.3 Exceptional Situations

8. Explain the difference between a programmer-defined exception that extends the Java `Exception` class and one that extends the Java `RunTimeException` class.

9. Expand your solution to Exercise 29 of Chapter 1, where you implemented the `Date` class, to include the appropriate throwing of the `DateOutOfBoundsException` by the class constructor, as described in this chapter. Don't forget to check for legal days, in addition to months and years.

10. Let's assume you have correctly implemented the `Date` class, as requested in Exercise 9. Recall that the `IncDate` class extends `Date`, adding a method called `increment` that adds one day to the date represented by the `Date` object. Consider exceptional situations that might be related to the `increment` method.

    a. Should the `increment` method test the current date values to make sure they are legal before incrementing the date?

    b. How about after incrementing the date? Would it be a good idea for the `increment` method to test the new date to make sure it is legal, perhaps raising the `DateOutOfBoundsException` if it is not?

11. Describe three ways to "handle" error situations within our ADT specification/implementation. For each approach, include a brief description of when it is most appropriate to use it.

12. What is wrong with the following method, based on our conventions for handling error situations?

```
public void method10(int number)
// Precondition:  number is > 0.
// Throws NotPositiveException if number is not > 0,
// otherwise ...
```

13. There are three parts to this exercise:

    a. Create a "standard" exception class called `ThirteenException`.

    b. Write a program that repeatedly prompts the user to enter a string. After each string is entered the program outputs the length of the string, unless the

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

length of the string is 13, in which case the `ThirteenException` is thrown with the message "Use thirteen letter words and stainless steel to protect yourself!" Your `main` method should simply throw the `ThirteenException` exception out to the run-time environment. A sample run of the program might be:

```
Input a string > Villanova University
That string has length 20.
Input a string > Triscadecaphobia
That string has length 16.
Input a string > misprogrammed
```

At this point the program bombs and the system provides some information, including the "Use thirteen letter words and stainless steel to protect yourself!" message.

c. Create another program similar to the one you created for part b, except this time, within your code, include a try-catch clause so that you catch the exception when it is thrown. If it is thrown, then catch it, print its message, and end the program "normally."

14. Write a class `Array` that encapsulates an array and provides bounds-checked access. The private instance variables should be `int index` and `int array[10]`. The public members should be a default constructor and methods (signatures shown below) to provide read and write access to the array.

```
void insert(int location, int value);
int retrieve(int location);
```

If the `location` is within the correct range for the array, the `insert` method should set that location of the array to the value. Likewise, if the `location` is within the correct range for the array, the `retrieve` method should return the value at that location—the approach taken by the library before Java 5.0. In either case, if `location` is not within the correct range, the method should throw an exception of class `ArrayOutOfBoundsException`. Write an application that helps you test your implementation. Your application should assign values to the array by using the `insert` method, then use the `retrieve` method to read these values back from the array. It should also try calling both methods with illegal location values. Catch any exceptions thrown by placing the "illegal" calls in a `try` block with an appropriate `catch`.

3.4 Formal Specification

15. Based on our Stack ADT specification, an application programmer has two ways to check for an empty stack. Describe them and discuss when one approach might be preferable to the other approach.

16. Show what is written by the following segments of code, given that `item1`, `item2`, and `item3` are `int` variables, and `stack` is an object that fits the

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

abstract description of a stack as given in the section. Assume that you can store and retrieve variables of type `int` on `stack`.

a.
```
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);
}
```

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

17. Your friend Bill says, “The `push` and `pop` stack operations are inverses of each other. Therefore performing a `push` followed by a `pop` is always equivalent to performing a `pop` followed by a `push`. You get the same result!” How would you respond to that? Do you agree?

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

18. The following code segment is a count-controlled loop going from 1 through 5. At each iteration, the loop counter is either printed or put on a stack depending on the `boolean` result returned by the method `random`. (Assume that `random` randomly returns either `true` or `false`.) At the end of the loop, the items on the stack are removed and printed. Because of the logical properties of a stack, this code segment cannot print certain sequences of the values of the loop counter. You are given an output and asked to determine whether the code segment could generate the output.

```
for (count = 1; count <= 5; count++)
{
  if (random())
    System.out.println(count);
  else
    stack.push(count);
}
while (!stack.isEmpty())
{
  number = stack.top();
  stack.pop();
  System.out.println(number);
}
```

a. The following output is possible: 1 3 5 2 4
   i. True ii. False iii. Not enough information

b. The following output is possible: 1 3 5 4 2
   i. True ii. False iii. Not enough information

c. The following output is possible: 1 2 3 4 5
   i. True ii. False iii. Not enough information

d. The following output is possible: 5 4 3 2 1
   i. True ii. False iii. Not enough information

19. In compiler construction, we need an observer method to examine stack elements based on their location in the stack (the top of the stack is considered location 1, the second element from the top is location 2, and so on). This is sometimes called (colloquially) a “glass stack” or (more formally) a “traversable stack.” The definition of the stack is exactly as we specify in this chapter, except we add a public method named `inspector` that accepts an `int` argument indicating the location to be returned. The method should return `null` if the argument indicates an unused location. Describe explicitly what you would add to the `StackInterface` interface to include this method.

20. In compiler construction, we need to be able to pop more than one element at a time, discarding the items popped. To do so, we provide an `int` parameter `count` for a `popSome` method that removes the top `count` items from the stack. The new method should throw `StackUnderflowException` as needed.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Write the `popSome` method at the application level, using operations from `StackInterface`.

21. In each plastic container of Pez candy, the colors are stored in random order. Your little brother Phil likes only the yellow ones, so he painstakingly takes out all the candies one by one, eats the yellow ones, and keeps the others in order, so that he can return them to the container in exactly the same order as before—minus the yellow candies, of course. Write the algorithm to simulate this process. (You may use any of the stack operations defined in the Stack ADT, but may not assume any knowledge of how the stack is implemented.)
22. Describe inheritance of interfaces and explain why it was used in Section 3.4.

Exercises 23–26 require "outside" research.

23. Describe the major differences between the Java library's `Vector` and `ArrayList` classes.
24. Explain how the iterators in the Java Collections Framework are used.
25. What is the defining feature of the Java library's `Set` class?
26. Which classes of the Java library implement the `Collection` interface?

3.5 Array-Based Implementations

27. Explain why an array is a good implementation structure for a bounded stack.
28. Describe the effects each of the following changes would have on the `ArrayStack` class.
    a. Remove the `final` attribute from the `DEFCAP` instance variable.
    b. Change the value assigned to `DEFCAP` to 10.
    c. Change the value assigned to `DEFCAP` to −10.
    d. In the first constructor change the statement to `stack = (T[]) new Object[100];`
    e. In `isEmpty`, change "`topIndex == -1`" to "`topIndex < 0`".
    f. Reverse the order of the two statements in the `if` clause of the `push` method.
    g. Reverse the order of the two statements in the `if` clause of the `pop` method.
    h. In the `throws` statement of the `top` method change the argument string from "Top attempted on an empty stack" to "Pop attempted on an empty stack."
29. Create a `toString` method for the `ArrayStack` class. This method should create and return a string that correctly represents the current stack. Such a method could prove useful for testing and debugging the `ArrayStack` class and for testing and debugging applications that use the `ArrayStack` class.
30. Write a segment of code (application level) to perform each of the following operations. Assume `myStack` is an object of the class `ArrayStack`. You may call any of the public methods of `ArrayStack`. You may declare additional stack objects.
    a. Set `secondElement` to the second element from the top of `myStack`, leaving `myStack` without its original top two elements.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

b. Set `bottom` equal to the bottom element in `myStack`, leaving `myStack` empty.

c. Set `bottom` equal to the bottom element in `myStack`, leaving `myStack` unchanged.

d. Print out the contents of `myStack`, leaving `myStack` unchanged.

31. Explain the differences between arrays and array lists.

32. Explain why we use a comparison to −1 for the `isEmpty` method of `ArrayStack`, yet in the `isEmpty` method for `ArrayListStack` we use a comparison to 0.

33. Exercise 19 described an `inspector` method for a stack.

a. Implement `inspector` for the `ArrayStack` class.

b. Implement `inspector` for the `ArrayListStack` class.

34. Exercise 20 described a `popSome` method for a stack.

a. Implement `popSome` for the `ArrayStack` class.

b. Implement `popSome` for the `ArrayListStack` class.

35. Two stacks of positive integers are needed, both containing integers with values less than or equal to 1000. One stack contains even integers; the other contains odd integers. The total number of elements in the combined stacks is never more than 200 at any time, but we cannot predict how many are in each stack. (All of the elements could be in one stack, they could be evenly divided, both stacks could be empty, and so on.) Can you think of a way to implement both stacks in one array?

a. Draw a diagram of how the stacks might look.

b. Write the definitions for such a double-stack structure.

c. Implement the `push` operation; it should store the new item into the correct stack according to its value (even or odd).

3.6 Application: Well-Formed Expressions

36. For each of the following programs that involve casting and Autoboxing, predict the result of compiling and running the program. Potential answers include "there is a syntax error because . . . ," "there is a run-time error because . . . ," and "the output of the program would be . . . ."

a.
```
public class test3
{
  public static void main(String[] args)
  {
    String s1;
    int i1;
    Object o1;
    i1 = 35;
    o1 = i1;
    s1 = (String) o1;
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION


```
      System.out.println(s1.toLowerCase());
    }
  }
```

b.
```
public class test4
{
  public static void main(String[] args)
  {
    Integer I1;
    int i1;
    Object o1;
    i1 = 35;
    o1 = i1;
    I1 = (Integer) o1;
    System.out.println(I1);
  }
}
```

37. Answer the following questions about the `Balanced` class:

a. Is there any functional difference between the class being instantiated in the following two ways?

```
Balanced bal = new Balanced ("abc", "xyz");
Balanced bal = new Balanced ("cab", "zxy");
```

b. Is there any functional difference between the class being instantiated in the following two ways?

```
Balanced bal = new Balanced ("abc", "xyz");
Balanced bal = new Balanced ("abc", "zxy");
```

c. Is there any functional difference between the class being instantiated in the following two ways?

```
Balanced bal = new Balanced ("abc", "xyz");
Balanced bal = new Balanced ("xyz", "abc");
```

d. Which type is pushed onto the `stack`? A `char`? An `int`? An `Integer`? Explain.

e. Under which circumstances is the first operation performed on the `stack` (not counting the `new` operation) the `top` operation?

f. What happens if the string `s`, which is passed to the `test` method, is an empty string?

38. Suppose we want to change our application so that it reports more information about an unbalanced string—namely, the location and value of the first unbalanced character. To report character locations to the user, we number the char-

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

acters starting with 1. For example, if the user enters the string "(xxx[x}]xx)x" the output would be "Unbalanced symbol } at location 7."

a. Describe how you would change the classes to implement this change.

b. Make the changes to the application and test the result.

3.7 Link-Based Implementation

39. What are the main differences, in terms of memory allocation, between using an array-based stack and using a reference-based stack?

40. Consider the code for the `push` method of the `LinkedStack` class. What would be the effect of the following changes to that code?

    a. Switch the first and second lines.

    b. Switch the second and third lines.

41. Draw a sequence of diagrams, of the style used in Section 3.7, to depict what happens from the inside view with the dynamic allocation of space for the references to the stack elements. Assume A, B, and C represent objects of class `String`.

    a.
```
UnboundedStackInterface<String> myStack;
myStack = new LinkedStack<String>();
myStack.push(A);
myStack.pop();
myStack.push(B);
myStack.push(C);
```

    b.
```
UnboundedStackInterface<String> myStack;
myStack = new LinkedStack<String>();
myStack.push(A);
myStack.push(B);
myStack.push(A);
```

    c.
```
UnboundedStackInterface<String> myStack;
myStack = new LinkedStack<String>();
myStack.push(A);
myStack.push(C);
myStack.push(B);
myStack.pop();
```

42. Create a `toString` method for the `LinkedStack` class. This method should create and return a string that correctly represents the current stack. Such a method could prove useful for testing and debugging the `LinkedStack` class and for testing and debugging applications that use the `LinkedStack` class.

43. Exercise 19 described an `inspector` method for a stack. Implement `inspector` for the `LinkedStack` class.

44. Exercise 20 described a `popSome` method for a stack. Implement `popSome` for the `LinkedStack` class.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

45. We decide to add a new operation to our Stack ADT called `popTop`. We add the following code to our `StackInterface` interface:

```
public T popTop() throws StackUnderflowException;
// Throws StackUnderflowException if this stack is empty,
// otherwise removes and returns top element from this stack.
```

An operation like this is often included for stacks. Implement the `popTop` method for the `LinkedStack` class.

46. Suppose we decide to add a new operation to our Stack ADT called `sizeIs`, which returns a value of primitive type `int` equal to the number of items on the stack. The method signature for `sizeIs` is

```
public int sizeIs()
```

a. Write the code for `sizeIs` for the `ArrayStack` class.

b. Write the code for `sizeIs` for the `LinkedStack` class (do not add any instance variables to the class; each time `sizeIs` is called you must "walk" through the stack and count the nodes).

c. Suppose you decide to augment the `LinkedStack` class with an instance variable `size` that always holds the current size of the stack. Now you can implement the `sizeIs` operation by just returning the value of `size`. Identify all of the methods of `LinkedStack` that you need to modify to maintain the correct value in the `size` variable and describe how you would change them.

d. Analyze the methods created/changed in parts a, b, and c in terms of Big-O efficiency.

47. Use the `LinkedStack` class to support an application that tracks the status of an online auction. Bidding begins at 1 (dollars, pounds, euros, or whatever) and proceeds in increments of at least 1. If a bid arrives that is less than the current bid, it is discarded. If a bid arrives that is more than the current bid, but less than the maximum bid by the current high bidder, then the current bid for the current high bidder is increased to match it and the new bid is discarded. If a bid arrives that is more than the maximum bid for the current high bidder, then the new bidder becomes the current high bidder, at a bid of one more than the previous high bidder's maximum. When the auction is over (the end of the input is reached), a history of the actual bids (the ones not discarded), from high bid to low bid, should be displayed. For example:

New Bid	Result	High Bidder	High Bid	Maximum Bid
7 John	New high bidder	John	1	7
5 Hank	High bid increased	John	5	7
10 Jill	New high bidder	Jill	8	10
8 Thad	No change	Jill	8	10
15 Joey	New high bidder	Joey	11	15

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

The bid history for this auction would be

Joey	11
Jill	8
John	5
John	1

Input/output details can be determined by you or your instructor. In any case, as input proceeds the current status of the auction should be displayed. The final output should include the bid history as described above.

3.8 Case Study: Postfix Expression Evaluator

48. Evaluate the following postfix expressions.
    a. 5 7 8 * +
    b. 5 7 8 + *
    c. 5 7 + 8 *
    d. 1 2 + 3 4 + 5 6 * 2 *
49. Evaluate the following postfix expressions. Some of them may be ill-formed expressions—in that case, identify the appropriate error message (e.g., too many operands, too few operands).
    a. 1 2 3 4 5 + + +
    b. 1 2 + + 5
    c. 1 2 * 5 6 *
    d. / 23 * 87
    e. 4567 234 / 45372 231 * + 34526 342 / + 0 *
50. Revise and test the postfix expression evaluator program as specified here.
    a. Use the `ArrayListStack` class instead of the `ArrayStack` class—do not worry about stack overflow.
    b. Catch and handle the divide-by-zero situation that was assumed not to happen. For example, if the input expression is 5 3 3 - /, the result would be the message "illegal divide by zero."
    c. Support a new operation indicated by "^" that returns the larger of its operands. For example, 5 7 ^ = 7.
    d. Keep track of statistics about the numbers pushed onto the stack during the evaluation of an expression. The program should output the largest and smallest numbers pushed, the total numbers pushed, and the average value of pushed numbers.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION