
2.1 Introduction

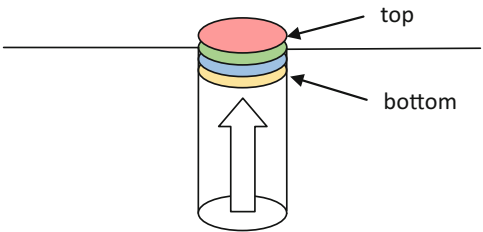
One of the easier data structures to learn first is the stack. Although it is simpler than other data structures, it still is very important and is used for a variety of purposes. For example, it is very useful in low-level programming such as assembly language, and many processor instruction sets have built-in instructions for stack manipulation [1]. The reason for its use is that it is sometimes faster to use a stack than to transfer data to a labeled memory location. Regardless of the language used, a stack can be used to swap memory locations, reverse data in an array, and evaluate arithmetic expressions. Further, as might have been discussed in a previous course or text, stacks are also used to implement recursion [2].

If a stack is so useful, how does it work? A simple explanation is to look at stacks in the real world. For example, if one is eating lunch at cafeteria, one might grab a plate from a stack of dishes. Or when coming back from a class, one might place a textbook on a stack of books on a desk. In both examples, items are only placed on or removed from the top of the stack. Further, it should be noticed that the last item put onto the stack is the first one taken off the stack. The property is known as last-in, first-out or *LIFO* and defines a *stack*. Of course, one might insert or remove an item from the middle or bottom of the stack, but then the structure is no longer has the properties of a stack nor would it be considered a stack. The act of putting something on a stack is known as a *push* operation and the act of removing an item from a stack is known as a *pop* operation.

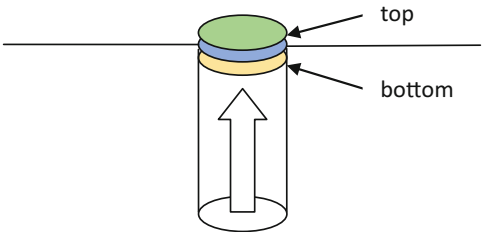
2.2 Analysis and Design

How could a stack be implemented in software? The simplest approach is to use an array. (An alternative approach using references will be explored in Chap. 7.) In addition to the stack itself, one needs to determine how to represent the top of the

stack. Looking at a real-world example can help in gaining some insight on how to implement a stack. Using the previous cafeteria plate example where the plates are stored below the surface of the table, consider the following diagram where the horizontal line represent the counter top and the large arrow represents a spring pushing the plates up.



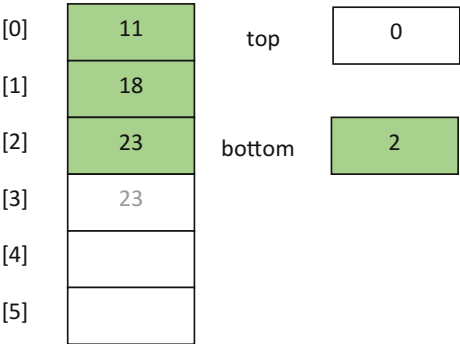
As plates are removed from the top of the stack, the spring below causes the next plate to move to the top and all the plates below move up one position as shown below:



Where there were previously four plates on the stack, now there are only three plates on the stack. Also note that the pointer to the top plate has not moved, but rather only the pointer to the bottom has moved. Although this model looks like a good possibility, how might it translate into an array? Consider the following six-element array containing four integers.

[0]	7	top	0
[1]	11		
[2]	18	bottom	3
[3]	23		
[4]			
[5]			

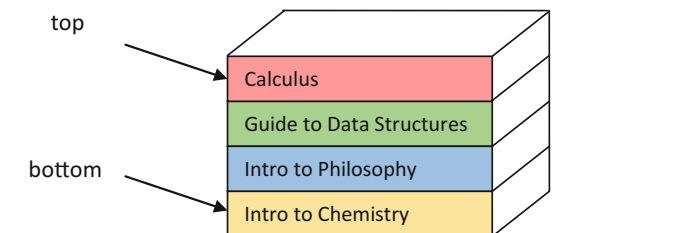
First, recall that a six-element array would have its elements numbered from 0 to 5. Further, do not forget the difference between the index of an array element and the contents of an array, where the zeroth array element above contains a 7. Notice that instead of a pointer, the variables `top` and `bottom` contain the index of the top and bottom elements, respectively. Now if the top element was removed, the integer 11 would have to be copied from the first position to the zeroth position, then the integer 18 could be copied from the second position to the first position, and so on. Lastly, the variable `bottom` would need to change from 3 to 2. All the changes are shown in green as follows:



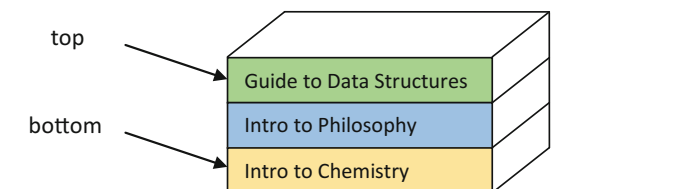
Although shown in light gray in the above figure, in actuality a copy of the integer 23 would still be in position 3 of the array since it was merely copied from one element to another. However, it is the variable `bottom` now at 2 which indicates the bottom of the stack, not the number of items in the array. Note that in removing all the items from the stack, `bottom` would be a `-1` and `top` would remain at a `0`. This is an important point concerning the distinction between an array and a stack as a data structure as discussed in Chap. 1. The array merely implements the data structure, but it is a combination of all the variables and the algorithm that makes up a data structure.

Although the above representation of a stack seems to work, are there any disadvantages to it? To answer this question, consider if instead of just four items in the stack, what if it was a 100 element array with 100 items in the stack? To remove just one item in the stack, it would need to copy 99 items. What if it was a 1000 element array with a 1000 items in the stack? Then to remove just one item, 999 elements would need to be copied. To extrapolate, if there were n items in the stack, then $n - 1$ items would need to be moved. As discussed in Sect. 1.4, this process would be $O(n)$. Clearly as the number of items increases, the efficiency of this design is lacking. Even though the above seems to work fairly well for a small number of items, one of the things that must be considered when designing a data structure is how it works with a large number of items.

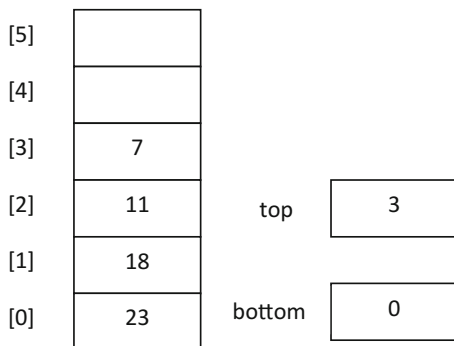
Instead of using the plate example as a model, consider the other example involving a stack of books on a desk in the following diagram where the horizontal line represents the desktop:



In this case when a book is removed from the stack, the rest of the stack does not move as it does in the plate example. As shown below, top moves down one place as opposed to the bottom moving up.

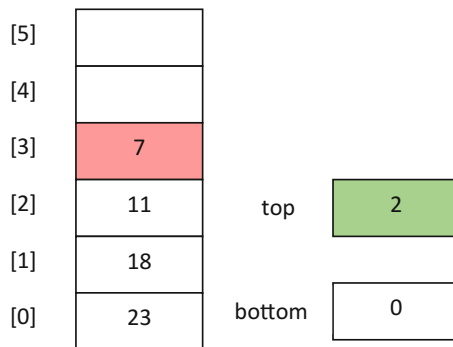


How might this model be represented using an array? Using the same numbers as before, examine the following diagram:



First, note that typically an array would be drawn with the zeroth element on the top and subsequent elements following below as was done in the previous example, but the above array has the zeroth element at the bottom instead. The reason for this is that when using an array to represent a stack it is usually more convenient to have the zeroth element drawn at the bottom, so that when items are added, they are added to the next higher number index and it looks more like a physical stack. Does this affect processing on the computer? No, since how it is drawn has no effect on the processing. Drawing the array this way is only more convenient for viewing by the reader. (Note that this text will typically continue to draw arrays with the zeroth element at the top, except when an array is representing a stack as above.) Also, notice that instead of having top at 0 and bottom at 3, these numbers are also

reversed in this example. When the top number, 7, is removed from the stack, the value of `top` will change from 3 to 2 as shown below.



As shown in red to indicate it has been popped off the stack, the integer 7 is still in element 3. The item was not technically removed from the stack, but rather only `top` was decremented. Since `top` is now at 2, that is what defines the top of the stack data structure, not what is in the array. Further, the advantage of this model is that whether there are four, one hundred, or one thousand items in the stack, none of the items needs to be copied. So, if there were n items in the stack, the process would not be of $O(n)$, but rather a constant. As discussed in Sect. 1.4, if the amount of time is constant, regardless what that number is, it is represented as $O(1)$.

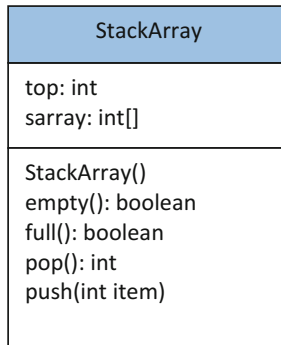
2.3 Stack Class: Data Members and Methods

The idea of a stack lends itself well to be implemented as an object. As with any object, one of the considerations is which instance variables should be used. In looking at the variables in the previous section one can see that both the array and the variable `top` are necessary. But what about the variable `bottom`—is it needed? Since it does not appear to change, then it probably is not necessary. Would a count of how many items in the stack be needed? Since the variable `top` seems to indicate how many items are in the stack, a count is not necessary either.

What sort of operations should be allowed on a stack? As mentioned in Sect. 2.1, a stack is a LIFO structure, meaning that the last item pushed onto the stack is the first item popped off the stack and that items are put onto only one end of the stack, the top. In addition to `push` and `pop` methods, since an array is of finite length and it is possible when pushing an item onto the stack that there might not be any room on the stack, it is a good idea to have a method to determine whether the stack is full. Conversely, it is also possible when popping an item off the stack that there might not be any items on the stack, so it is also a good idea to include a method to determine whether the stack is empty. Two other methods that might be useful are a `peek` method, which allows one to see what is on the top of

the stack without removing it, and a `size` method, which indicates how many items are on the stack. Although these two methods might be useful in some situations, they are not used here and are left as exercises at the end of this chapter.

So initially it seems that there should be four methods plus a constructor. The `push` method should put an item on the stack, so it should have a parameter to pass the item to the stack and since it will not be returning a value, it should be a `void` method. The `pop` method will be removing and returning an item from the stack, so it should be a value returning method. Both `empty` and `full` should return a value indicating whether the stack is empty or full respectively, so they too should be value-returning methods that return a `boolean` value of `true` or `false`. Although later in this chapter the concept of generics which allow many different types to be used will be discussed, for now it will be assumed that only integers will be stored on the stack. The result is that the following UML diagram would represent the stack using an array called `sarray`.



When pushing an item onto an empty stack, where should the first item be placed in the array? Looking back at the last figure in the last section, it should be clear that it ought to go in location 0 which could be indicated by the variable `top`. However, should `top` start off at 0 and then be incremented in preparation for the next item to be pushed onto the stack, or should `top` begin at -1 and be incremented by 1 prior to putting the item at location 0? Either technique works well, but this text will use the former as shown below and leave the latter as an exercise at the end of the chapter.

```
sarray[top] = item;
top++;
```

What if `sarray` were full? One solution is to create a larger array and then copy all the items from the old array into the new array. This is a good solution, would allow processing to continue, and is also left as an exercise at the end of the chapter. However, it would take processing time to create a new array and copy the contents, so a simpler solution is used here. Instead, if the stack is full, an error message is output to the user and the item is not placed on the stack. There is yet another solution using exceptions which will also be presented shortly. Lastly, should `push`

be `public` or `private`? Since it will be called from another program, it should be `public`. The complete method for `push` is shown below:

```
public void push(int item) {
    if(full())
        System.out.println("Stack is full");
    else {
        sarray[top] = item;
        top++;
    }
}
```

Should one be concerned that the `full` method has not been written yet and is this a problem? It is if it is never written, but right now it is not a problem. That is the beauty of modular design and UML diagrams. It has already been determined what the method will do, so it does not have to be written immediately. In a larger programming project, the various methods could be written by other team members and as long as the other methods are written according to the specifications, work can progress on all the modules in parallel.

Continuing on to the `pop` method, it should be noticed that `top` is now indicating where the next item will be pushed, so before an item can be popped off the stack, `top` will need to be decremented. The basic code for the `pop` routine is as follows:

```
top--;
item = sarray[top];
```

Notice that it is essentially the same as the `push` routine except in reverse order. As with the `push` method which is concerned with whether the stack is full, the `pop` method should be concerned with whether the stack is empty, so an `if` statement is again used. As with `push`, the `pop` method is declared as `public`.

```
// Caution: Poorly Implemented Code
public int pop() {
    if(empty()) {
        System.out.println("Stack is empty");
        return -1;
    }
    else {
        top--;
        return sarray[top];
    }
}
```

Assuming that the values on the stack will be non-negative, the then section of the `if` statement returns a `-1` so that a value will be returned should the stack be empty, otherwise the else section returns the item from the top of the stack. Although this code segment would “work”, it is not the best solution. Note that using two `return` statements is considered unstructured programming and a method should have only one entry point and one exit point. Although this method has one entry point at the `if` statement, there are two ways it can leave via the two `return` statements. Though not much of a problem here since it is such a small code segment, it is not a good habit to start because in larger code segments it can be a potential source of logic errors and can be very difficult to debug. Alternatively, the following code segment uses only one `return` statement.

```
public int pop() {
    int item = -1;
    if (empty())
        System.out.println("Stack is empty");
    else {
        top--;
        item = sarray[top];
    }
    return item;
}
```

Notice that a local variable `item` is used to hold the `-1` or the value copied from the stack and there is now only one `return` statement. Although this variable takes up one additional memory location, the amount of memory used is negligible and the more structured code is worth the minimal cost. Also, note as mentioned in the previous section that the value returned is still in the array since it was merely copied into `item`. Is this a problem? No, because if another item is pushed onto the array, the prior value will just be overwritten and if another item is popped off the stack, the value of `top` will first be decremented so that the next value in the stack will be returned.

As might have been discussed in a previous course or text, Java provides an extensive set of classes to support exception handling when an execution error occurs. How can one take advantage of these classes during the development of the `StackArray` class? Instead of using a `println` statement, an exception can be thrown. The types of exceptions that may arise during the `push` or `pop` operations are not checked during compile time, but they are detected during runtime and called runtime exceptions. Therefore, an object of `RuntimeException` can be thrown. The result is that the `push` method can be rewritten as shown below:


```
public void push(int item){
    if(full())
        throw new RuntimeException("Stack is full");
    sarray[top] = item;
    top++;
}
```

Note that an `else` statement is no longer necessary because if an exception is thrown, the program will stop and the subsequent code will not be executed. Similarly the `pop` method can be rewritten as follows:

```
public int pop(){
    if(empty())
        throw new RuntimeException("Stack is empty");
    top--;
    return sarray[top];
}
```

In addition to the `else`, note that the variable `item` is also deleted from the code. Again, if an exception is thrown the subsequent code will not be executed and a value is not returned. What about the `full` and `empty` methods? Looking at the `empty` method first, how does one know the stack is empty? As mentioned previously, it was decided that `top` would initially be 0, so that would indicate an empty stack. The code for `empty` could be written as follows:

```
// Caution: Poorly Implemented Code
if(top == 0)
    return true;
else
    return false;
```

Although this code segment would “work”, it suffers from the same problem as the first version of the `pop` method in that it is using two `return` statements and is unstructured. As before, a possible solution is to declare a local variable, assign it a boolean value, and then return the contents of the variable as follows:

```
//Caution: Poorly Implemented Code
boolean flag;
if(top == 0)
    flag = true;
else
    flag = false;
return flag;
```

Though there is one extra variable, the code segment is now structured and has only one exit point. However, there is an even simpler and more efficient solution to this problem. Since the value of `top == 0` is either `true` or `false`, why assign the boolean value to a variable? Why not just return the results of the relation `top == 0` as shown below?

```
return top == 0;
```

If `top` equals 0, the results of `top == 0` will be `true` and this is the boolean value that will be returned. Likewise, if `top` does not equal zero, the results of `top == 0` will be `false` and this is the value that will be returned. The result is that there is no need for an `if` statement and thus no need for an extra variable and no possibility of having two `return` statements.

Lastly, should `empty` be `public` or `private`? Since as will be seen, `empty` is used in the main program, it should be declared as `public`. The complete method is shown below:

```
public boolean empty() {  
    return top == 0;  
}
```

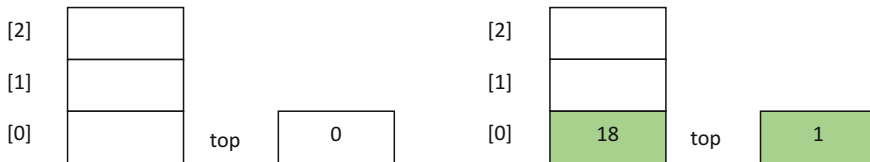
Is there a possibility that `top` will ever be equal to a negative value? If all the modules are written correctly, the answer is no. However, what if the code in `pop` is accidentally modified so that the value in `top` can skip over the value 0 and continue on towards the negative numbers? Of course if access to a negative element of the array is attempted, an execution error would occur. To help ensure that does not happen, the above code segment could be rewritten as:

```
private boolean empty() {  
    return top <= 0;  
}
```

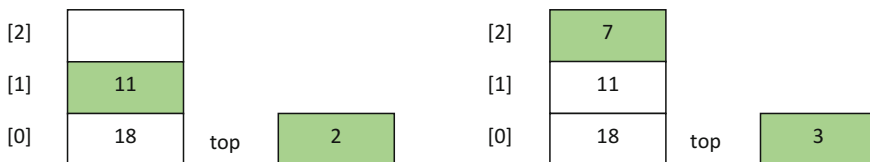
Which one of the two is the better way to code the method? The second one has the advantage that if a negative value is encountered, the code could continue to execute. However, the disadvantage is that it might mask a potential error in the code. If a negative index were to be encountered, then maybe an execution error should occur so that the programmer will be aware of the error and the logic can be corrected. For now, the second approach will be taken, but if one's instructor, place of employment, or an independent reader prefers the first approach, then it should be used.

The `full` method will be very similar to the `empty` method, except it will check the upper bound of the variable `top`. If there is a 3 element array, what is the largest value that `top` will have? Since the elements are numbered 0–2, one might be tempted

to say 2, but although that would work, it is not quite correct. When writing a new code segment it is not the time for guessing, but rather careful analysis of the situation. Instead, drawing a diagram and walking through a small sample size will result in the correct solution. A little time at this stage can save a lot of debugging time later and is well worth the effort. Assuming that the stack is initially empty as shown below to the left below and the numbers 18, 11, and 7 need to be pushed on to the stack, the following diagram to the right indicates the resulting value in `top` after the 18 is pushed onto the stack and `top` is incremented by 1.



Going from left to right in the diagram below, the second number 11 is placed into position 1 and `top` is incremented to 2. Lastly, the number 7 is put into position 2 and `top` is incremented to 3.



As can be seen, the final value of `top` when the stack is full is 3. So, for an `n`-element stack, when the stack is full the value of `top` would be `n`. Given this information, the `full` method would be as follows:

```
public boolean full() {
    return top >= sarray.length;
}
```

Notice the relation `>=` is used, but as mentioned above `==` could alternatively be used if preferred. Also, instead of a user-defined constant indicating the size of the array, the `.length` constant could be used as shown above. The only method remaining is the constructor, where the stack should be set to empty by setting `top` to 0 and the space for the array `sarray` should be allocated using a constant as shown below along with the other data members:

```
public final int N = 3;
private int top, sarray[];

public StackArray() {
    top = 0;
    sarray = new int[N];
}
```

However, what if one wanted to change the size of the array? As a programmer, the constant could be changed prior to compilation, but wouldn't it be more convenient to be able to change it during execution time? This can be accomplished by adding a parameter to the above method and replacing the constant `N` with the parameter `n`. Further, another constructor is created without a parameter and using `this(N)` to call the original constructor as shown below:

```
private static final int N = 3;
private int top, sarray[];

//constructors
public StackArray() {
    this(N);
}
public StackArray(int n) {
    top = 0;
    sarray = new int[n];
}
```

Note that the constant `N` has now been made a class constant using the word `static`, so that it can be used as an argument in the first constructor. Now the users of the class have the choice of either using the default stack of size `N` or they can set the size of the stack through an argument. Putting all the methods together along with the data members, the class definition for the `StackArray` class appears in Fig. 2.1.

Notice that the constructor is first, followed by the value-returning methods, followed by the `void` method, and within each group the methods are in alphabetical order. Alternatively, should one's instructor or place of employment prefer otherwise, all methods regardless of whether they are value-returning or not can be placed in alphabetical order.

```
public class StackArray {
    private static final int N = 3;
    private int top, sarray[];

    // constructors
    public StackArray() {
        this(N);
    }
    public StackArray(int n) {
        top = 0;
        sarray = new int[n];
    }

    // value returning methods
    public boolean empty() {
        return top <= 0;
    }
    public boolean full() {
        return top >= sarray.length;
    }
    public int pop() {
        if(empty())
            throw new RuntimeException("Stack is empty");
        top--;
        return sarray[top];
    }

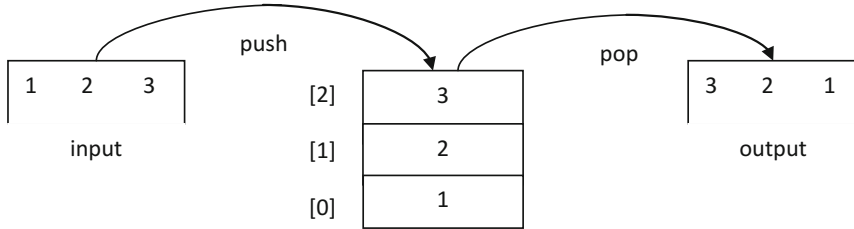
    // void method
    public void push(int item) {
        if(full())
            throw new RuntimeException("Stack is full");
        sarray[top]=item;
        top++;
    }
}
```

Fig. 2.1 StackArray class

2.4 Reversing Integers

Having created a `StackArray` class, how can it be used to solve various problems? As mentioned in the introduction to this chapter, one of the ways in which a stack can be used is to reverse items. As might have been demonstrated in a previous course or text, an array can be used for this task, but a stack is also a convenient way to reverse an array.

Recall that as each item is pushed onto a stack, when an item is popped off the stack, the last one in will be the first one out. For example, first the number 1 could be pushed onto the stack, then the number 2, and lastly the number 3. Then when the numbers are popped off the stack they will be in reverse order. First the number 3 would be popped off, then the number 2, and finally the number 1. If they are output as they are popped off, the output will be in reverse order as shown in the following diagram:



Of course, the first task is to prompt for and input the values to be pushed onto the stack. Although an array and a second loop could be used to push the integers onto the stack, this would be inefficient in terms of space and time, so instead as the values are input, they can be pushed onto the stack. What sort of loop should be used? If there will always be a fixed number, then a fixed iteration `for` loop should be used. However, if there could be a variable number of integers input, then a `while` loop is the best choice. How will the user indicate that all the integers have been input? This would be a good use of a sentinel controlled loop, where assuming only non-negative integers will be entered as data, a negative integer could serve as the sentinel value. In addition to checking for a `-1`, the input loop could also check to see if the stack is full. Then instead of having an exception thrown, it could allow processing to continue. The resulting loop can be seen below:

```
System.out.print("Enter an integer or a -1 to stop: ");
number = scanner.nextInt();
while(number >= 0 && !stack.full()) {
    stack.push(number);
    System.out.print("Enter an integer or a -1 to stop: ");
    number = scanner.nextInt();
}
```

After all the items have been pushed onto the stack, a second loop could be used to pop the items off the stack and output them. What sort of loop would be best in this circumstance? If a count was kept about the number of integers entered, then a fixed iteration loop would be a good choice. However, another alternative is to not use a count and just keep popping integers off the stack until the stack is empty. In this case a `while` loop would be the best choice. The resulting loop can be seen below:

```
System.out.print("The reverse integers are: ");
while(!stack.empty())
    System.out.print(stack.pop() + " ");
System.out.println();
```

Putting all the code together and using the `StackArray` class from Fig. 2.1, the complete program is shown below:

```
import java.util.*;
class Reverse {
    public static void main(String[] args) {

        int number;
        StackArray stack;
        stack = new StackArray();

        Scanner scanner = new Scanner(System.in);

        System.out.println();
        System.out.print("Enter an integer or a -1 to stop: ");
        number = scanner.nextInt();
        while(number >= 0 && !stack.full()) {
            stack.push(number);
            System.out.print("Enter an integer or a -1 to stop: ");
            number = scanner.nextInt();
        }
        System.out.println();
        System.out.print("The reverse integers are: ");
        while(!stack.empty())
            System.out.print(stack.pop() + " ");
        System.out.println();
        System.out.println();
    }
}
```

2.5 Generic Types

Given the stack which stores integers and the application that reverses the sequence of integers implemented in the previous section, suppose items of different types were to be reversed. The `StackArray` class would need to be written for the particular type. The new class would be essentially the same as `StackArray` for integers and the only modification that needs to be made would be to change the type of the element stored on the stack. Is there a better way to do this rather than copying and making small changes for different types? It would be nice if the definition of the stack could be written once and used for any type of data. If a parameter could be used to represent the type of an item at the time the abstract data structure is written, then it could be specified when the application is implemented later. The following section describes how to accomplish the above idea using *generic types*.

First, the `StackArray` class presented in Fig. 2.1 could be copied and modified, where the name of the class could be changed to `StackArrayGeneric`, so that if the original class is still needed, there will not be any confusion. In order to define a generic class a *type parameter* in angle brackets is placed at the end of the class name as shown below:

```
public class StackArrayGeneric<T> {
```

Generally single capital letters such as `T` and `E` are used as type parameters in Java. Inside a generic class, the type parameter is used for declarations as if it was a regular type. As can be seen below, `int` is replaced by `T` in the declaration of `sarray`.

```
private T[] sarray;
```

Further, the return type of the `pop` method `int` is also changed to `T`.

```
public T pop() {
```

Note that in `pop`, if the local variable `item` is needed, it could be defined as type `T` and initialized to `null` instead of `-1`. Also, the type of the formal parameter, `item`, in the `push` method needs to be changed to `T`.

```
public void push(T item) {
```

Lastly, one more place the type `T` is needed is in the constructor. In it an array is instantiated, specifically an array of a generic type. At first one might think the following statement could be used to create an array of type `T`:


```
//Caution: Incorrectly Implemented Code  
sarray = new T[n];
```

However, the above code causes a compilation error. In Java an array of a generic type cannot be instantiated. The solution is the use of typecasting as shown below.

```
sarray = (T[]) new Object[n];
```

First, an array of the `Object` class is created and then it is cast into the array of the generic type. The class `Object` is the root of the class hierarchy in Java. In other words, the `Object` class is directly or indirectly a superclass of all the user defined and predefined classes. The above statement will be compiled successfully, but it will produce a warning message during compilation. The message can be ignored and one can proceed to the execution of the program. However, remember that running code which generates a warning is not encouraged unless the meaning of the warning is understood fully and it is not avoidable as it is in this case. Figure 2.2 is the complete `StackArrayGeneric` class definition.

To use a generic class in the main program, a type must be specified for the type parameter. It should be noted that the type has to be a reference type. In other words, it cannot be a primitive data type, such as `int`, `double`, and `char`. To store items of a primitive data type in generic data structures, programmers have to either define a class in order to instantiate an object which contains a field of the primitive data type or they can use a predefined *wrapper class* in the Java API. A wrapper class defines the object which wraps primitive data types and there are eight of them available in Java. They also provide many useful methods dealing with their values. The eight primitive types and their corresponding wrapper classes are shown in Table 2.1.

Before declaring and creating an object of a generic class, recall from the previous section how an object of the `StackArray` class, which was specifically written for the `int` type, was declared and created as shown below:

```
StackArray stack;  
stack = new StackArray();
```

For the generic class, since `int` cannot replace the type parameter, to declare and create a stack which holds integers, the corresponding wrapper class, `Integer`, is used as follows,

```
StackArrayGeneric<Integer> stack;  
stack = new StackArrayGeneric <Integer> ();
```

```

public class StackArrayGeneric<T> {
    private static final int N = 3;
    private int top;
    private T[] sarray;

    // constructors
    public StackArrayGeneric() {
        this(N);
    }

    public StackArrayGeneric(int n) {
        top = 0;
        sarray = (T[]) new Object[n];
    }

    // value returning methods
    public boolean empty() {
        return top <= 0;
    }

    private boolean full() {
        return top >= sarray.length;
    }

    public T pop() {
        if(empty())
            throw new RuntimeException("Stack is empty");
        top--;
        return sarray[top];
    }

    // void method
    public void push(T item) {
        if(full())
            throw new RuntimeException("Stack is full");
        sarray[top]=item;
        top++;
    }
}

```

Fig. 2.2 StackArrayGeneric<T> class

Instead of using <T> as in the heading of the class definition, <Integer> is used here in the declaration of the variable, `stack`. Once declared, one way to push an `int` onto the stack is by using the following statement to push a 2 onto the stack using the wrapper class `Integer`. First an object of the `Integer` class containing the integer 2 is created by calling the constructor, `new Integer(2)`. Then the reference of the newly created object is used as an argument for the `push` method.

```
stack.push(new Integer(2));
```

Table 2.1 Wrapper classes

Primitive type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

The statement below outputs the integer value that is popped off the stack. The `pop` method returns a reference to an `Integer` object, and in order to display the contents of the object, the `intValue` method defined in the `Integer` class is used to convert `Integer` type to `int` type so that it can be output by the `println`.

```
System.out.println(stack.pop().intValue());
```

Combining the above, a simple main program using `StackArrayGeneric` class is implemented below:

```
public class TestStackArrayGeneric {
    public static void main(String[] args) {
        StackArrayGeneric<Integer> stack;
        stack = new StackArrayGeneric <Integer> ();
        stack.push(new Integer(2));
        stack.push(new Integer(3));
        stack.push(new Integer(4));
        while(!stack.empty())
            System.out.println(stack.pop().intValue());
    }
}
```

The wrapper class is necessary to create a stack containing integers when dealing with the generic class. However, it is cumbersome to create an `Integer` object using the constructor in order to push an item onto the stack and extract an `int` value from the `Integer` object using the `intValue` method when popping an item off the stack and outputting it. Although this method works, there is an easier way.

Instead of using a wrapper class explicitly, one can take an advantage of the Java features *autoboxing* and *unboxing*. Autoboxing is an automatic conversion that the Java compiler makes from the primitive types to their corresponding wrapper class, and unboxing is a conversion from the wrapper class to their corresponding primitive types.

Java uses autoboxing if a primitive data type such as `int` is placed where a reference type such as `Integer` is required. Java automatically calls the constructor of the `Integer` class to create an object containing an `int` value. To push a 2 onto the stack, the following statement is legal because Java automatically boxes the `int` value of 2.

```
stack.push(2);
```

Unboxing occurs when, for example, the reference type `Integer` is used in a place where the primitive data type `int` is required. With a stack of type `Integer` and since the return type of the `pop` method is `Integer`, Java would unbox the value returned from `stack.pop()` automatically by calling the `intValue` method so that the content of the `Integer` object is displayed when the following line is executed:

```
System.out.println(stack.pop());
```

Now, relying on Java's autoboxing and unboxing features, the `TestStackArrayGeneric` class can be rewritten as follows:

```
public class TestStackArrayGeneric {
    public static void main(String[] args) {
        StackArrayGeneric<Integer> stack;
        stack = new StackArrayGeneric <Integer> ();

        stack.push(2);
        stack.push(3);
        stack.push(4);

        while(!stack.empty())
            System.out.println(stack.pop());
    }
}
```

The above code is cleaner and expresses the program intentions better without cluttering the code. Both autoboxing and unboxing can occur for all the eight primitive types and their corresponding wrapper classes pairs.

2.6 Prefix and Postfix Expressions

Typically, an arithmetic expression is in a form known as an infix expression. As a simple example, $1 + 2$ is an *infix* expression because the arithmetic operator is in-between the two operands, 1 and 2. Alternatively, the arithmetic operator can be placed prior to the operands as in $+ 1 2$ which is known as a *prefix* expression. This was originally proposed by the Polish mathematician Jan Lukasiewicz in the early 1800s and as a result is sometimes called Polish notation. Further, the arithmetic operator can also be placed after the operands as in $1 2 +$ and this is known as *postfix* notation or reverse Polish notation (*RPN*).

At first these two alternative forms of expressions might seem a little awkward because one is more familiar with infix notation, but there are various advantages to using prefix or postfix notation and they are a good application for stacks as will be seen shortly. Further, they are very helpful in explaining tree traversals later in Chap. 8.

One important advantage of prefix and postfix notation is their ability to indicate precedence without the use of parentheses. As one should remember from an introductory course in computer science or from mathematics, multiplication and division operators have precedence over addition and subtraction. Further, in infix notation one can always use parentheses to override the precedence of the operators. If there are more than one set of parentheses, then the innermost nested ones are evaluated first. If the parentheses are not nested, then they are evaluated from left to right. In fact, if there is a tie of any sort, such as a subtraction symbol and an addition symbol or a division symbol and a multiplication symbol, the order is also from left to right.

For example, how could the infix expression $a + b * c$ be converted to prefix? First, one writes down all of the operands in the same order that they appear in the infix expression.

$a \ b \ c$

Note that although one might be able to rearrange the operands and achieve similar results, it is not recommended because it will cause difficulties later. Next, since the multiplication has precedence over addition, it is placed prior to the operands b and c .

$a * b \ c$

Then the addition operator is placed prior to the a and the product of $* b c$ as follows:

$$+ a * b c$$

However, what if one wanted to have the addition performed first? In infix notation, parentheses would be used as in $(a + b) * c$. It should be noted that, in prefix and postfix notation, parentheses are not required to indicate precedence, nor should they be used. Instead the precedence is indicated by the location of the operators. In order to convert the above infix expression to prefix, the process outline above is again followed. First the operands are written in the same order that they appear in the infix expression.

$$a b c$$

Since the addition is done first, the plus sign is placed prior to the a and b as follows:

$$+ a b c$$

Next the multiplication is placed prior to $+ a b$ and the c as shown below:

$$* + a b c$$

Note that this equation does not contain parentheses and is different than the previous prefix expression indicating the difference in precedence. The same rules follow for postfix except the operators are placed after the operands so that for the infix expression $a + b * c$, the multiplication symbol is placed after the b and c , and then the plus sign is after the a and the product $b c *$ as follows:

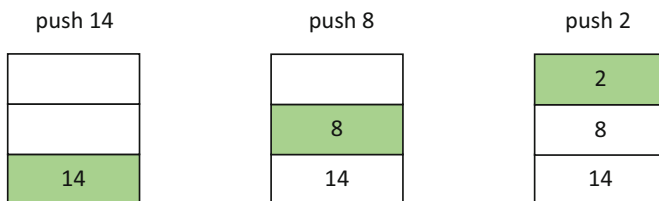
$$a b c * +$$

The second infix expression $(a + b) * c$ would have the plus sign placed after the a and b , and the multiplication symbol would be placed after the sum $a b +$ and the c as shown below:

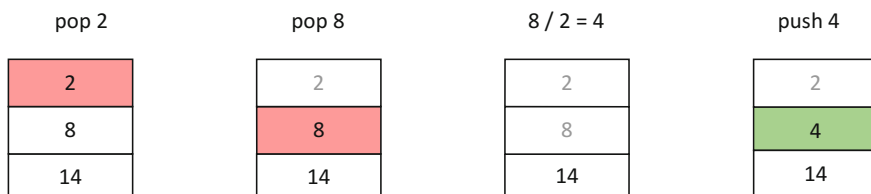
$$a b + c *$$

Again note that parentheses are not used and this postfix expression is different from the proceeding one. One of the advantages of prefix and postfix expressions is that they can be evaluated using a stack. Using integers instead of characters, what would the result be for following infix expression $14 - 8 / 2$? An answer of 3 would be incorrect. Recall the order of precedence and note that the correct answer is 10. As demonstrated previously, the corresponding postfix expression would be $14 8 2 / -$ which can be evaluated using a stack. The rule for evaluating a postfix expression is that when scanning from left to right, every time an operand is encountered it should be pushed onto the stack. Then whenever an operator is encountered, the last two operands should be popped from the stack, the operation performed and the results pushed back onto the stack. Assuming a three-element stack, this is illustrated below

by first pushing the three operands onto the stack with the green indicating the most recent item pushed onto the stack:

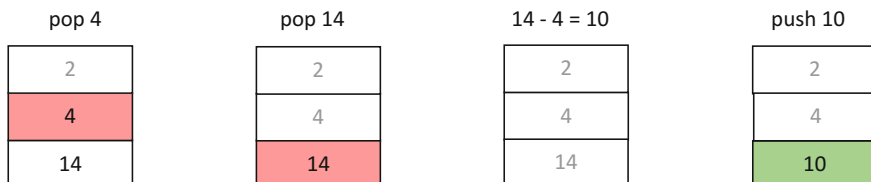


Then when the division sign is encountered, the 2 is popped from the stack, the 8 is popped from the stack, the division is performed, and the result of 4 is pushed back onto the stack as shown below:

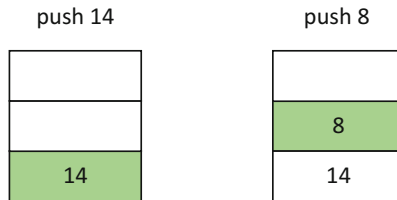


First, notice the order of the operands in the equation as they are popped off the stack when the division is performed. Although not really a concern with addition or multiplication because these operations are commutative, the same does not hold true for division and subtraction. With division, the first operand 2 is used as the divisor and the second operand 8 is used as the dividend. Further, notice that the positions for the 2 and 8 are in red as they are popped off the stack and then the numbers are shown in light gray to indicate that although they are no longer part of the stack, they are still in the array as done previously. Also note that the 8 is overwritten when the 4 is pushed onto the stack.

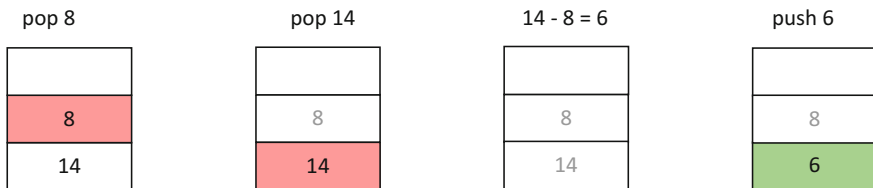
Next when the subtraction is encountered, the 4 and the 14 are popped off the stack, and the 4 is the subtrahend and the 14 is the minuend. The difference is calculated and the result of 10 is then pushed back onto the stack as shown below. If the answer needs to be output or used in another calculation, it can simply be popped off the stack.



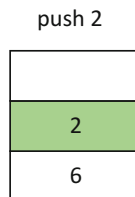
If subtraction is to be performed first, the infix expression would be $(14 - 8)/2$ and the answer this time would be 3. The corresponding postfix expression is $14\ 8\ -\ 2\ /$ and its evaluation using a stack is to first



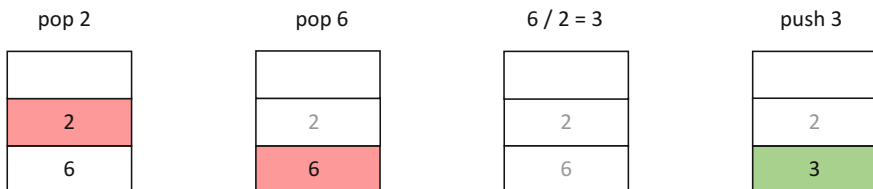
When the minus sign is encountered, the two operands are popped off the stack, the operation is performed, and the results are pushed back onto the stack as follows:



Then the 2 is pushed onto the stack:



When the division symbol is encountered, the 2 and the 6 are popped off the stack, the division is performed and the quotient is pushed back onto the stack. Again, should the answer be needed, it is merely popped off the stack.



As can be seen, both postfix expressions are evaluated correctly using a stack and without the need for parentheses. This is also a good way to check to see if the conversion from infix to postfix has been performed properly. If the operators are not put in the proper order, then the answer might be incorrect. An even worse problem is if an expression is not created properly such as $a + b c *$ and a program execution error could occur. In this example when the plus sign is evaluated there

would be only one operand on the stack and when the second operand is attempted to be popped from the stack an empty stack exception would be thrown.

Note that prefix expressions can be evaluated similarly but instead of evaluating the expression from left to right, they would need to be evaluated from right to left. The implementation of a program to evaluate postfix expressions is left as an exercise at the end of the chapter, as well the related problem to evaluate prefix expressions.

2.7 Complete Program: Checking for Palindromes in Strings

This section designs a program that checks if the given word is a palindrome. What is a palindrome? A palindrome is a sequence of symbols such as a word, phrase, verse, or sentence, and that reads the same way from either direction, forward or backward. Examples of single word palindromes include “pop,” “level,” and “radar.” In order to check if the word is a palindrome, the first and the last characters are compared, then the second and the second last characters are compared and so on until all the characters have been compared. During the process, as soon as two symbols do not match, then the word is not a palindrome.

One of the ways to write this application is to use a stack and a stack can be implemented with arrays discussed in this chapter or with objects linked together described in Chap. 7. Therefore, here is the good opportunity to use data abstraction to hide the implementation of these data structures from the person who will be implementing the palindrome application using a stack. As discussed in Sect. 1.3 an interface is used, and generic types and exceptions are incorporated. The `StackGeneric <T>` interface will have headings of all the methods except constructors in the `StackArrayGeneric<T>` class presented in Fig. 2.2 without their code as shown in Fig. 2.3.

One small change needs to be made in the `StackArrayGeneric<T>` class in Fig. 2.2 where the first line is replaced by `public class StackArrayGeneric<T> implements StackGeneric <T>`. Figure 2.4 contains the complete `StackArrayGeneric<T>` class.

In Chap. 7, another class called `StackRefGeneric <T>` class that implements `StackGeneric <T>` interface will be presented. Further, the UML diagram in Fig. 2.5 shows the `StackGeneric <T>` interface and two classes which contains four methods. Note that a dashed line from the class to the interface is used to indicate the relationship between the interface and the classes that implement it.

Now that a stack is implemented using an interface, one can concentrate on how to use a stack to write the palindrome application. First, while scanning a word from left to right, each of the characters are pushed onto a stack. As characters are popped from the stack, they are checked whether they are equal to the characters in the original word. For example, if the characters of the word “level” are pushed onto the stack, the first character popped is “l” and the first character in the original

```
public interface StackGeneric<T> {  
    public boolean empty();  
  
    public boolean full();  
  
    public T pop();  
  
    public void push(T item);  
}
```

Fig. 2.3 StackGeneric<T> interface

word is “l”. The second character popped and the second character in the original word are both “e”. The third character popped and the third character in the original word are also the same. When the stack is empty and all the characters have been matched, then the word “level” is shown to be a palindrome. Now, if the characters for the word “label” are pushed onto the stack, the first character popped and the first character in the original word are the same. However, the second character popped and the second character in the original word are different, “e” and “a” respectively. Therefore, the word “label” is not a palindrome.

The following program implements the above description. In general it is a good idea to declare variables for an object using interface types rather than class types as discussed in Sect. 1.4. Therefore, the variable `stack` is declared as of type `StackGeneric<Character>`. First, a user will enter a word. Notice the stack of type `StackArrayGeneric<Character>` is created meaning the content of the stack is an object of type `Character`. Also, notice that it is using autoboxing and unboxing. If a `char` type is used where a `Character` object is required, Java will automatically create a `Character` object which contains the `char` value. When a `Character` object is used in a place of where a `char` is required, the `char` value will be extracted from the `Character` object.

```
public class PalindromeVersion1 {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        int i, j;  
        String word;  
        boolean isPalindrome;  
        StackGeneric<Character> stack;  
  
        System.out.print("Enter a word: ");  
        word = scanner.next();  
        stack = new StackArrayGeneric<Character> (word.length());
```

```

public class StackArrayGeneric<T> implements StackGeneric<T> {
    private static final int N = 3;
    private int top;
    private T[] sarray;

    // constructors
    public StackArrayGeneric() {
        this(N);
    }

    public StackArrayGeneric(int n) {
        top = 0;
        sarray = (T[]) new Object[n];
    }

    // value returning methods
    public boolean empty() {
        return top <= 0;
    }

    public boolean full() {
        return top >= sarray.length;
    }

    public T pop() {
        if(empty())
            throw new RuntimeException("Stack is empty");
        top--;
        return sarray[top];
    }

    // void method
    public void push(T item) {
        if(full())
            throw new RuntimeException("Stack is full");
        sarray[top]=item;
        top++;
    }
}

```

Fig. 2.4 StackArrayGeneric<T> class that implements the StackGeneric<T> interface

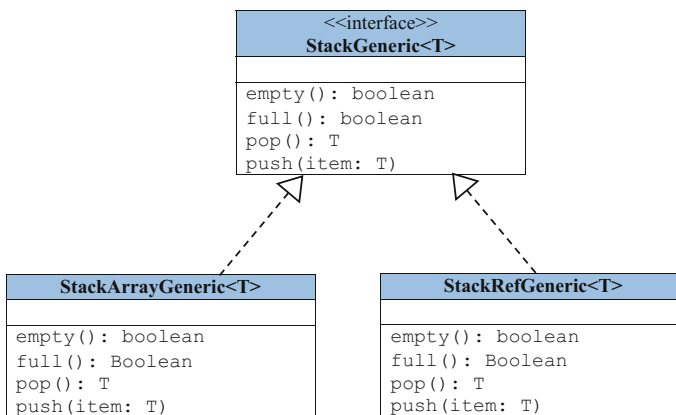


Fig. 2.5 UML Diagram for StackGeneric<T> interface

```
for(i = 0; i <word.length(); i++)
    stack.push(word.charAt(i));
j = 0;
isPalindrome = true;
while(!stack.empty() && isPalindrome) {
    if(stack.pop() != word.charAt(j))
        isPalindrome = false;
    j++;
}
if(isPalindrome)
    System.out.println("The word entered is a palindrome.");
else
    System.out.println("The word entered is NOT a palindrome.");
}
```

Examine the above program carefully and see if there is an improvement that could be made to the code in order to increase efficiency. Instead of storing entire word, what if a half of the word is kept in the stack and characters are checked for equality starting from the middle? The first half of the word from the stack and the last half of the word are compared. The following program is the second version of the palindrome program:

```
import java.util.*;
public class PalindromeVersion2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int i, j;
        String word;
        boolean isPalindrome;
        StackGeneric <Character> stack;
        System.out.print("Enter a word: ");
        word = scanner.next();
        stack = new StackArrayGeneric<Character> (word.length()/2);
```

```
for(i = 0; i < word.length() / 2; i++)
    stack.push(word.charAt(i));
j = i;
if(word.length()%2 != 0)
    j++;
isPalindrome = true;
while(!stack.empty() && isPalindrome) {
    if(stack.pop() != word.charAt(j))
        isPalindrome = false;
    j++;
}

if(isPalindrome)
    System.out.println("The word entered is a palindrome.");
else
    System.out.println("The word entered is NOT a palindrome.");
}
```

When the above program is compiled and executed using the sample input of “level” and “label,” the output of the program looks as given below:

```
Enter a word: level
The word entered is a palindrome.

Enter a word: label
The word entered is NOT a palindrome.
```

2.8 Summary

- A stack is a LIFO structure meaning Last-In, First-Out.
- Putting an item on a stack is a push operation, whereas removing an item from a stack is a pop operation.
- Care must be taken to indicate when an array based stack is empty depending on whether `top` is at 0 or -1.
- Depending on the value of `top` used to indicate an empty stack will affect the value used to indicate a full stack.
- Also, care must be taken when creating `push` and `pop` routines to determine whether the increments and decrements occur before or after inserting an item into or removing an item from an array based stack.

- Exceptions are often a convenient way to inform programmers and users when an execution error has occurred.
- Generic types and the wrapper class make data structures reusable with different primitive data types.
- Autoboxing is an automatic conversion that Java makes from the primitive types to their corresponding wrapper classes and unboxing is an automatic conversion that Java makes from wrapper classes to their corresponding primitive types.
- Prefix and postfix expressions should not use parentheses to indicate precedence.

2.9 Exercises (Items Marked with an * Have Solutions in Appendix B)

- *1. Assuming the stack `intStack` of type `StackArray` in Fig. 2.1 was initially empty, draw a stack after the following operations are completed:

```
int x, y, z;
intStack.push(41);
intStack.push(6);
x = intStack.pop();
y = intStack.pop();
intStack.push(78);
intStack.push(5);
intStack.push(79);
z = intStack.pop();
intStack.push(5);
```

2. Assuming the stack `characterStack` of type `StackArrayGeneric<Character>` which is an object of the generic class `StackArrayGeneric<T>` in Fig. 2.2 is initially empty, draw a stack after the following operations are completed:

```
Character x, y, z;
characterStack.push('U');
characterStack.push('V');
x = characterStack.pop();
characterStack.push('W');
y = characterStack.pop();
```

```
characterStack.push('P');  
characterStack.push('Q');  
z = characterStack.pop();  
characterStack.push('C');
```

- *3. Write a method `peek` that can be added to the `StackArray` class in Fig. 2.1 which returns the item at the top of the stack without removing it.
4. Add the heading of the `peek` method described in the previous question to the `StackGeneric <T>` interface in Fig. 2.3 and implement it in the `StackArrayGeneric<T>` class in Fig. 2.4.
5. Write a method `size` that can be added to the `StackArrayGeneric<T>` class in Fig. 2.2 which returns the number of items in the stack.
6. In Sect. 2.3, the `push` method was implemented by having the variable `top` be 0 initially and incremented by 1 after pushing an item on to the stack. Rewrite the `push` method in Fig. 2.1 where the `top` is initialized to -1 and incremented prior to pushing an item at the location. Also, rewrite the `pop` method so that it works correctly too.
7. Section 2.3 mentioned that there was a possibility an item could not be pushed onto the stack because the stack is full. One solution is to create a larger array and then copy all the items from the old array into the new array. Write a method `expandArray` that creates a new array that is twice the size of the previous one and copies the item over to the new array. The method should be added to the `StackArray` class in Fig. 2.1 and be called from the `push` method.
8. Find a reference on how to convert a decimal number to a binary number such as *Guide to Assembly Language* [1] or other similar text and then write a method to perform the conversion using a stack.
9. What are the time complexities of the methods `full` and `empty` in Fig. 2.1 and also `expandArray` from the question 7? Give your answer in Big O notation.
- *10. What are the time complexities of the two complete programs in Sect. 2.7? Is there a difference between the two? Give your answer in Big O notation.
11. Write a program to evaluate a postfix expression using an algorithm discussed in Sect. 2.6. Assume that the input contains only single digit numbers as operands and that a valid postfix expression is entered as input.
12. Write a program to evaluate a prefix expression using a stack. A hint for the implementation is given in Sect. 2.6. Assume that the input contains only single digit numbers as operands and that a valid prefix expression is entered as input.

Guide to Data Structures

A Concise Introduction Using Java

Streib, J.T.; Soma, T.

2017, XIII, 376 p. 43 illus., 9 illus. in color., Softcover

ISBN: 978-3-319-70083-0