Lists, Stacks, and Queues

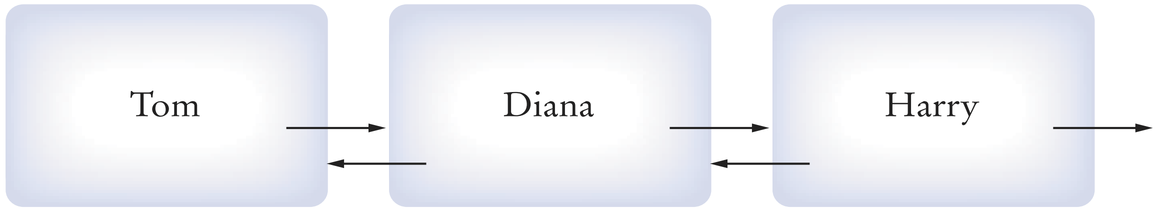
Topics

1. Linked Lists
2. Stacks and Queues
3. Selecting a Collection

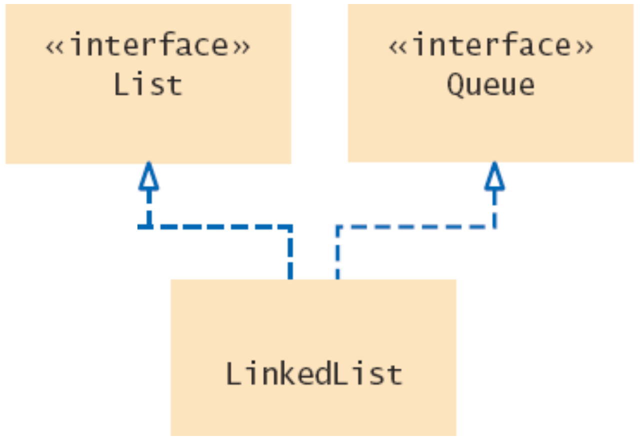
Linked List

15.2 Linked Lists

- Linked lists use references to maintain an ordered lists of 'nodes'
  - The 'head' of the list references the first node
  - Each node has a value and a reference to the next node



- They can be used to implement
  - A List Interface
  - A Queue Interface

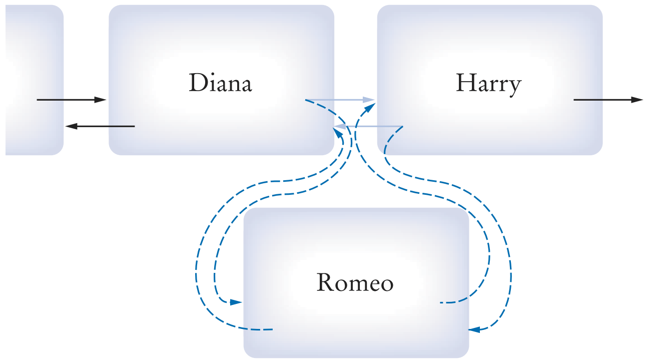


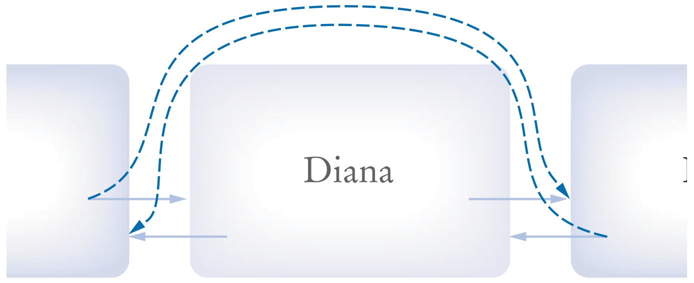
Node

```
class LinkedList
{
   private class Node
   {
       Object data;
        Node next;
        public Node(Object data, Node next)
        {
           this.data = data;
           this.next = next;
         }
  }
  Node head = null;
  // methods of LinkedList
}
```

Linked Lists Operations

- Efficient Operations
  - Insertion of a node
    - Find the elements it goes between
    - Remap the references
  - Removal of a node
    - Find the element to remove
    - Remap neighbor's references
  - Visiting all elements in order
- Inefficient Operations
  - Random access





Each instance variable is declared just like other variables we have used.

LinkedList: Important Methods

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<String>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 676).

Generic Linked Lists

- The Collection Framework uses Generics
  - Each list is declared with a type field in < > angle brackets

```
LinkedList<String> employeeNames = . . .;
```

```
LinkedList<String>
LinkedList<Employee>
```

List Iterators

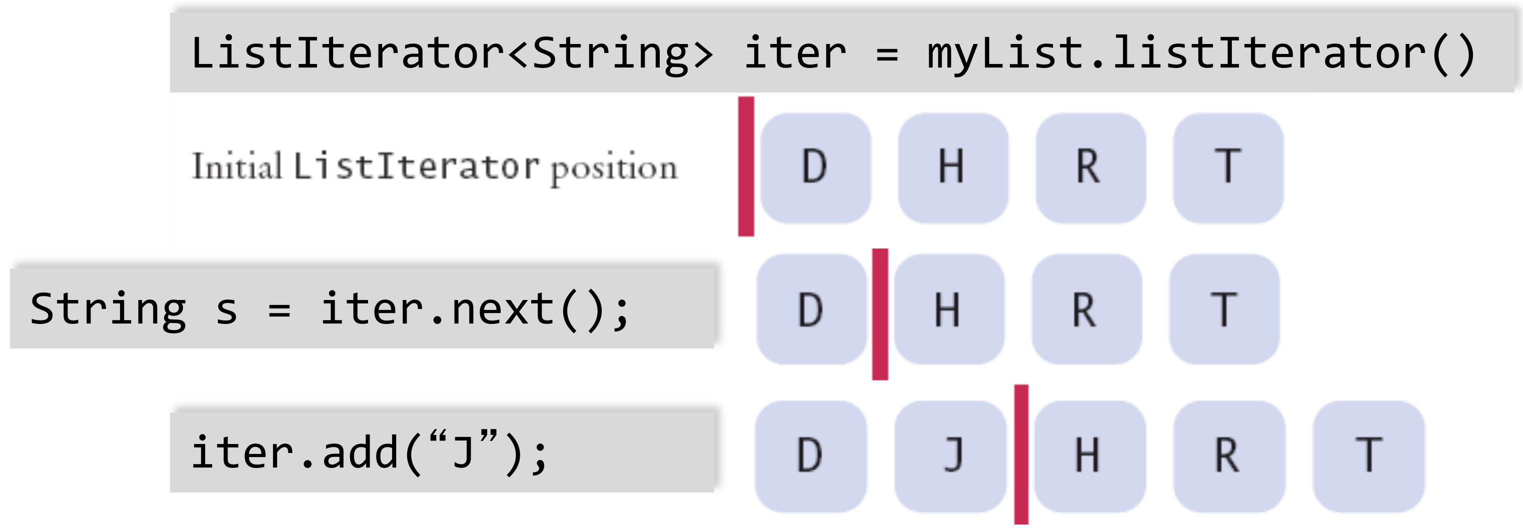
- When traversing a `LinkedList`, use a `ListIterator`
  - Keeps track of where you are in the list.

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iter = employeeNames.listIterator()
```

- Use an iterator to:
  - Access elements inside a linked list
  - Visit other than the first and the last nodes

Using Iterators

- Think of an iterator as pointing **between** two elements



```
ListIterator<String> iter = myList.listIterator()
```

```
String s = iter.next();
```

```
iter.add("J");
```

- Note that the generic type for the `listIterator` must match the generic type of the `LinkedList`

Iterator and ListIterator Methods

- Iterators allow you to move through a list easily
  - Similar to an index variable for an array

Table 3 Methods of the Iterator and ListIterator Interfaces

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.

`iter.previous();`
`iter.set("Juliet");`
	The `set` method updates the last element returned by `next` or `previous`. The list is now `[Juliet]`.

`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`   s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list. `previous` and `hasPrevious` are `ListIterator` methods.

`iter.add("Diana");`	Adds an element before the iterator position (`ListIterator` only). The list is now `[Diana, Juliet]`.

`iter.next();`
`iter.remove();`
	`remove` removes the last element returned by `next` or `previous`. The list is now `[Diana]`.

Adding and Removing with Iterators

```
iterator.add("Juliet");
```

- Adding
  - A new node is added AFTER the Iterator
  - The Iterator is moved past the new node
- Removing
  - Removes the object that was returned with the last call to `next` or `previous`
  - It can be called only once after `next` or `previous`
  - You cannot call it immediately after a call to `add`.

If you call the `remove` method improperly, it throws an `IllegalStateException`.

```
while (iterator.hasNext())
{
  String name = iterator.next();
  if (condition is true for name)
  {
    iterator.remove();
  }
}
```

ListDemo.java (1)

- Illustrates adding, removing and printing a list

```java
import java.util.LinkedList;
import java.util.ListIterator;

/**
   This program demonstrates the LinkedList class.
*/
public class ListDemo
{
   public static void main(String[] args)
   {
      LinkedList<String> staff = new LinkedList<String>();
      staff.addLast("Diana");
      staff.addLast("Harry");
      staff.addLast("Romeo");
      staff.addLast("Tom");

      // | in the comments indicates the iterator position

      ListIterator<String> iterator = staff.listIterator(); // |DHRT
      iterator.next(); // D|HRT
      iterator.next(); // DH|RT

```

ListDemo.java (2)

```
        // Add more elements after second element

        iterator.add("Juliet"); // DHJ|RT
        iterator.add("Nina"); // DHJN|RT

        iterator.next(); // DHJNR|T

        // Remove last traversed element

        iterator.remove(); // DHJN|T

        // Print all elements

        System.out.println(staff);
        System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
    }
}
```

Program Run

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```

Stacks and Queues

15.5 Stacks, Queues and Priority Queues

- Queues and Stacks are specialized lists
  - Only allow adding and removing from the ends

	Insert At	Remove At	Operation
Stack	Start(top)	Start(top)	Last in, first out (LIFO)
Queue	End (tail)	Start (head)	First in, first out (FIFO)
Priority Queue	By Priority	Highest Priority (Lowest #)	Prioritized list of tasks

Stacks, Queues and Priority Queues

- Stacks are used for undo features (most recent first)

- Queues are like lines at the bank or store

- Priority Queues remove lowest number first

Working with Stacks

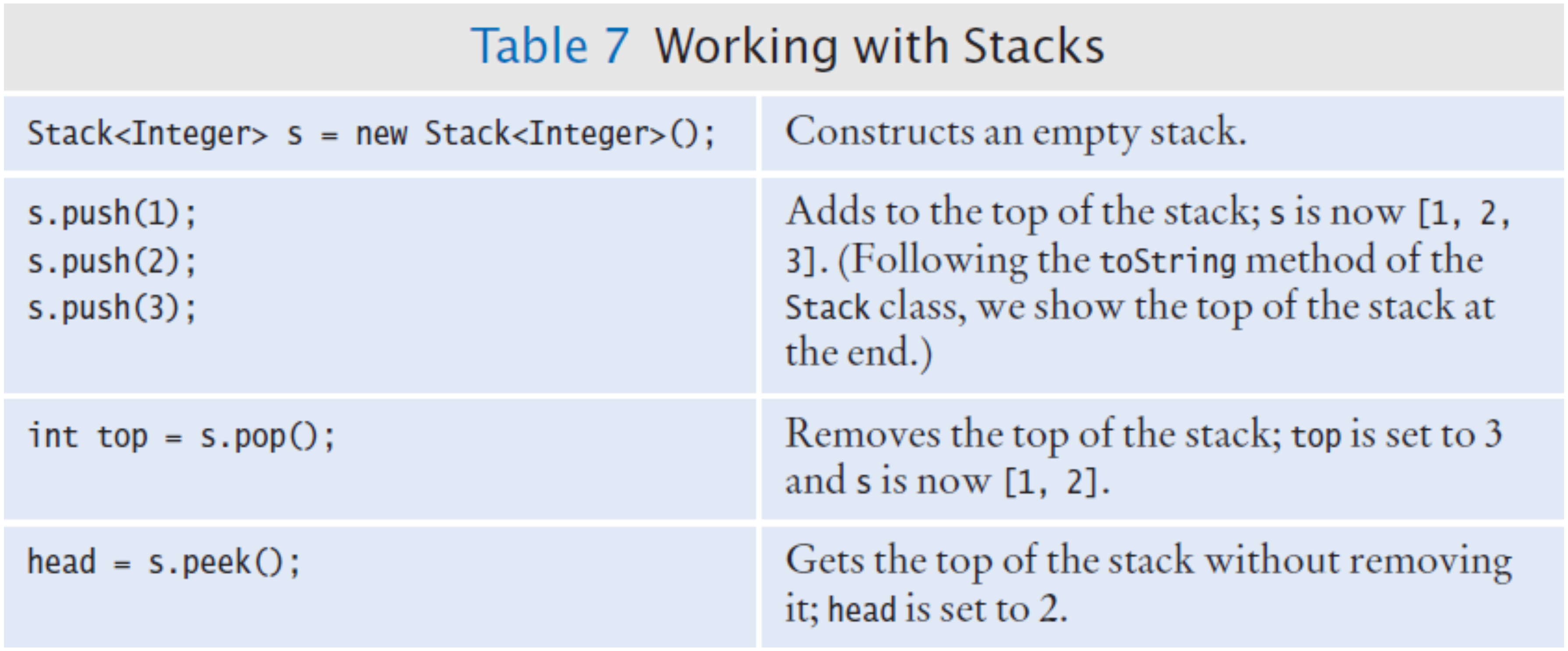


Table 7 Working with Stacks

`Stack<Integer> s = new Stack<Integer>();`	Constructs an empty stack.

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; `s` is now `[1, 2, 3]`. (Following the `toString` method of the `Stack` class, we show the top of the stack at the end.)

`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

Stack Example

- The Java library provides a Stack class that implements the abstract stack type's push and pop operations.
  - The Stack is not technically part of the Collections framework, but uses generic type parameters

```
Stack<String> s = new Stack<String>();
s.push("A");
s.push("B");
s.push("C");
// The following loop prints C, B, and A
while (s.size() > 0)
{
  System.out.println(s.pop());
}
```

The stack class provides a size method

Queues and Priority Queues

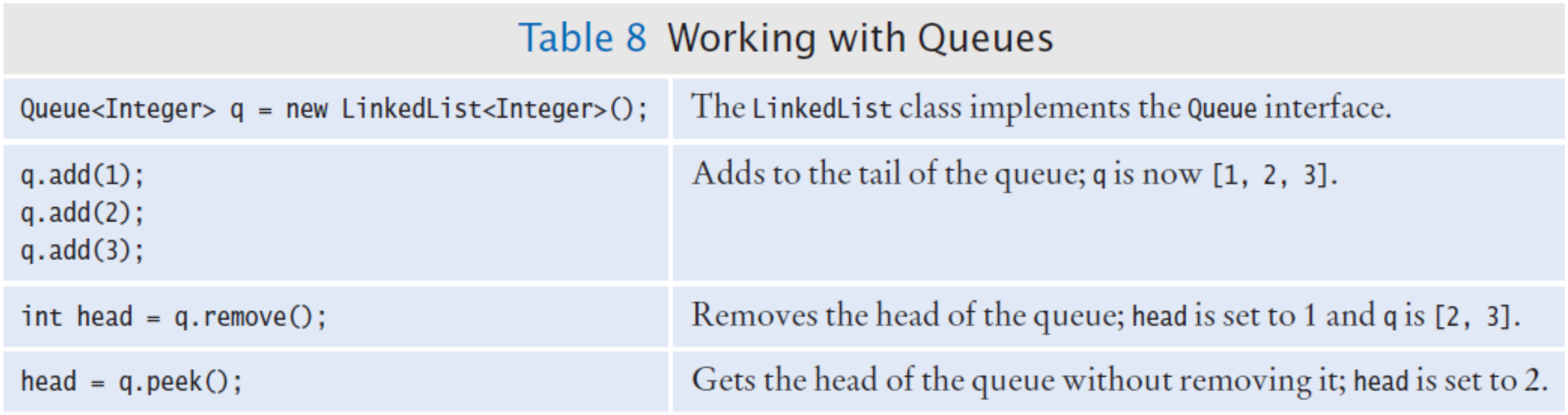


Table 8 Working with Queues

`Queue<Integer> q = new LinkedList<Integer>();`	The `LinkedList` class implements the `Queue` interface.

`q.add(1);`
`q.add(2);`
`q.add(3);`
	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.

`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.

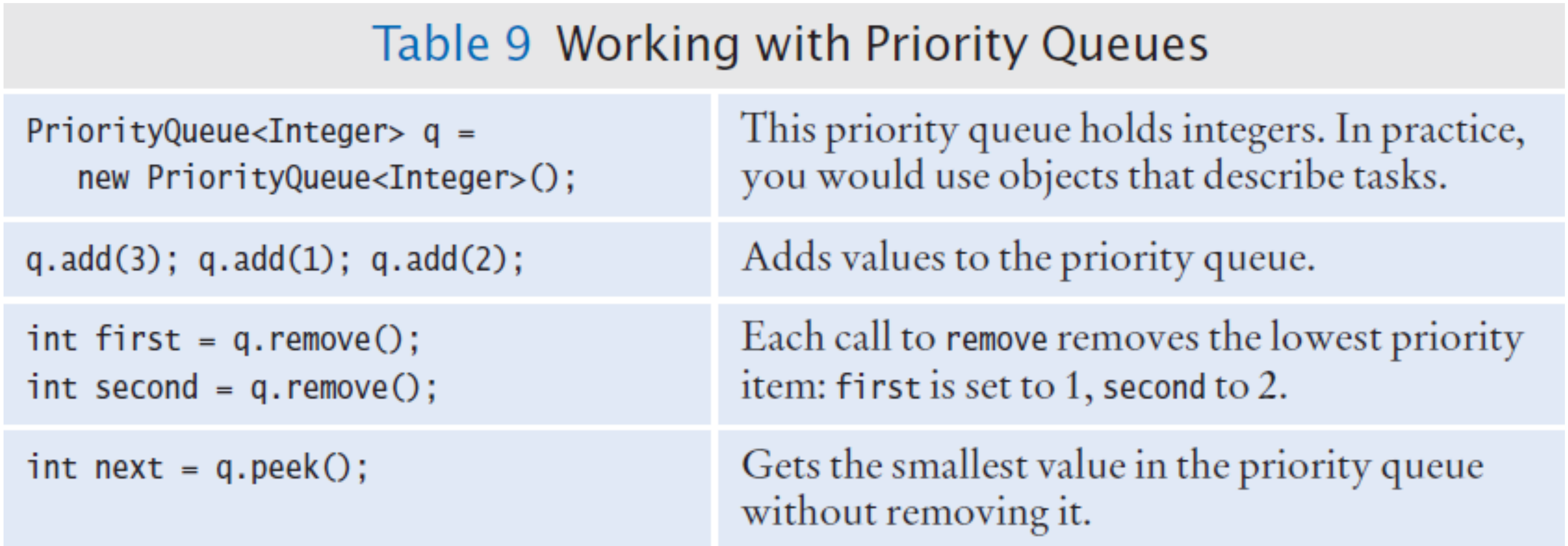


Table 9 Working with Priority Queues

`PriorityQueue<Integer> q =`
`   new PriorityQueue<Integer>();`
	This priority queue holds integers. In practice, you would use objects that describe tasks.

`q.add(3); q.add(1); q.add(2);`	Adds values to the priority queue.

`int first = q.remove();`
`int second = q.remove();`
	Each call to `remove` removes the lowest priority item: `first` is set to 1, `second` to 2.

`int next = q.peek();`	Gets the smallest value in the priority queue without removing it.

Priority Queues

- A priority Queue collects elements, each of which has a priority
  - Example: a collection of work requests, some of which may be more urgent than others
  - It is NOT a FIFO Queue

```
PriorityQueue<WorkOrder> q = new PriorityQueue<WorkOrder>();
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix broken sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

- Lowest value priority (1) will be removed first

```
WorkOrder next = q.remove();  // removes “Fix broken sink”
```

15.6 Stack and Queue Applications

- Balancing Parenthesis
  - Section 2.5, showed how to balance parenthesis by adding 1 for each left (and subtracting for each right)

```
-(b * b - (4 * a * c ) ) / (2 * a)
 1        2            1 0   1    0
```

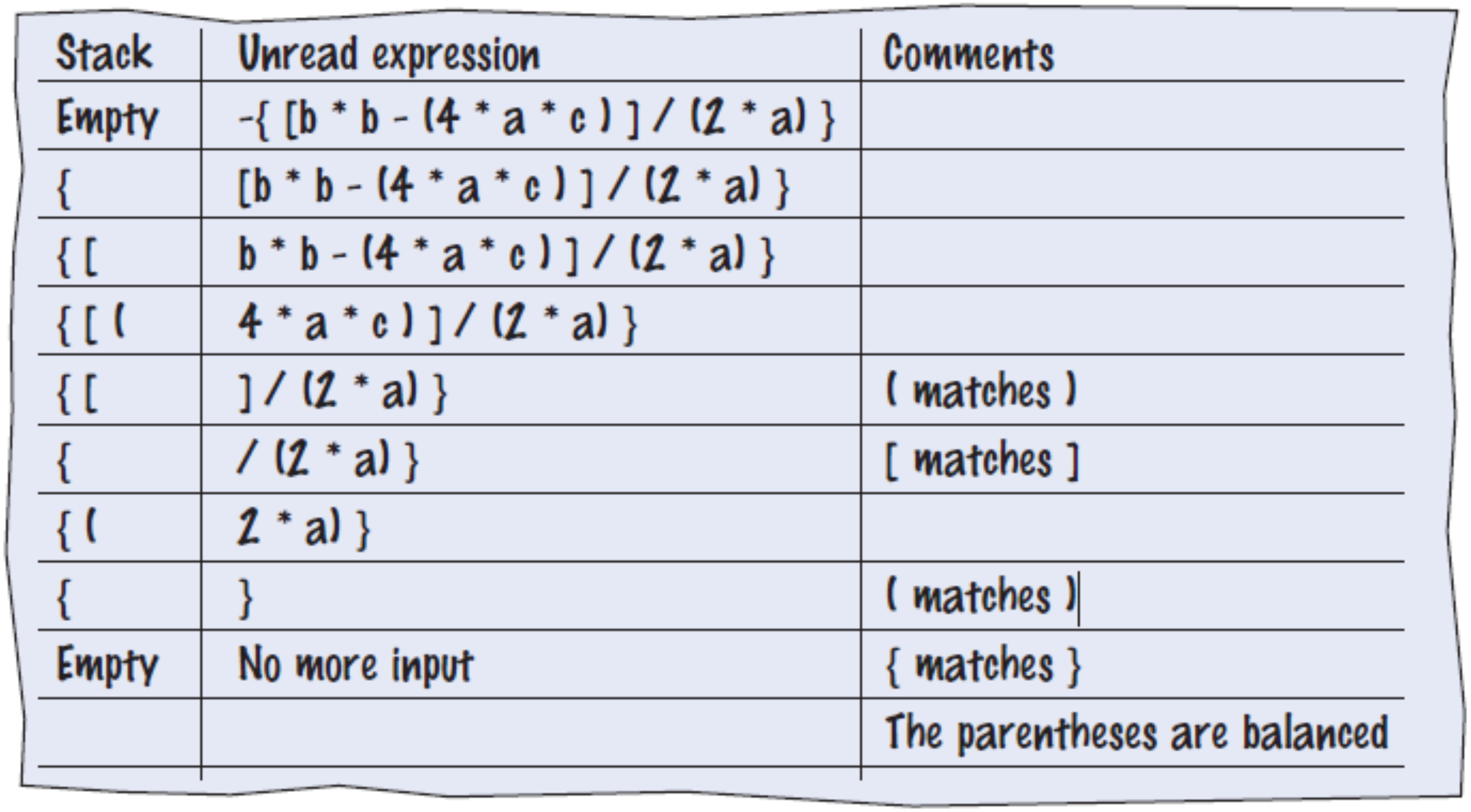
  - A stack can be used to keep track of 'depth':

```
When you see an opening parenthesis, push it on the stack.
When you see a closing parenthesis, pop the stack.
If the opening and closing parentheses don't match
    The parentheses are unbalanced. Exit.
    If at the end the stack is empty
    The parentheses are balanced.
Else
    The parentheses are not balanced.
```

Using a Stack (Example)

- Here is a walkthrough of the sample expression
  - We will use the mathematical version (three types of parenthesis)

$$-\{[b \cdot b - (4 \cdot a \cdot c)]/(2 \cdot a)\}$$



Stack	Unread expression	Comments
Empty	-{ [b * b - (4 * a * c)] / (2 * a) }	
{	[b * b - (4 * a * c)] / (2 * a) }	
{ [	b * b - (4 * a * c)] / (2 * a) }	
{ [(	4 * a * c)] / (2 * a) }	
{ [	] / (2 * a) }	(matches)
{	/ (2 * a) }	[matches]
{ (	2 * a) }	
{	}	(matches)
Empty	No more input	{ matches }
		The parentheses are balanced

Reverse Polish Expressions

- The first handheld calculator used a notation that was easily implemented with a stack: Reverse Polish
  - No parenthesis required if you…
    - Input both operands first, then the operator:

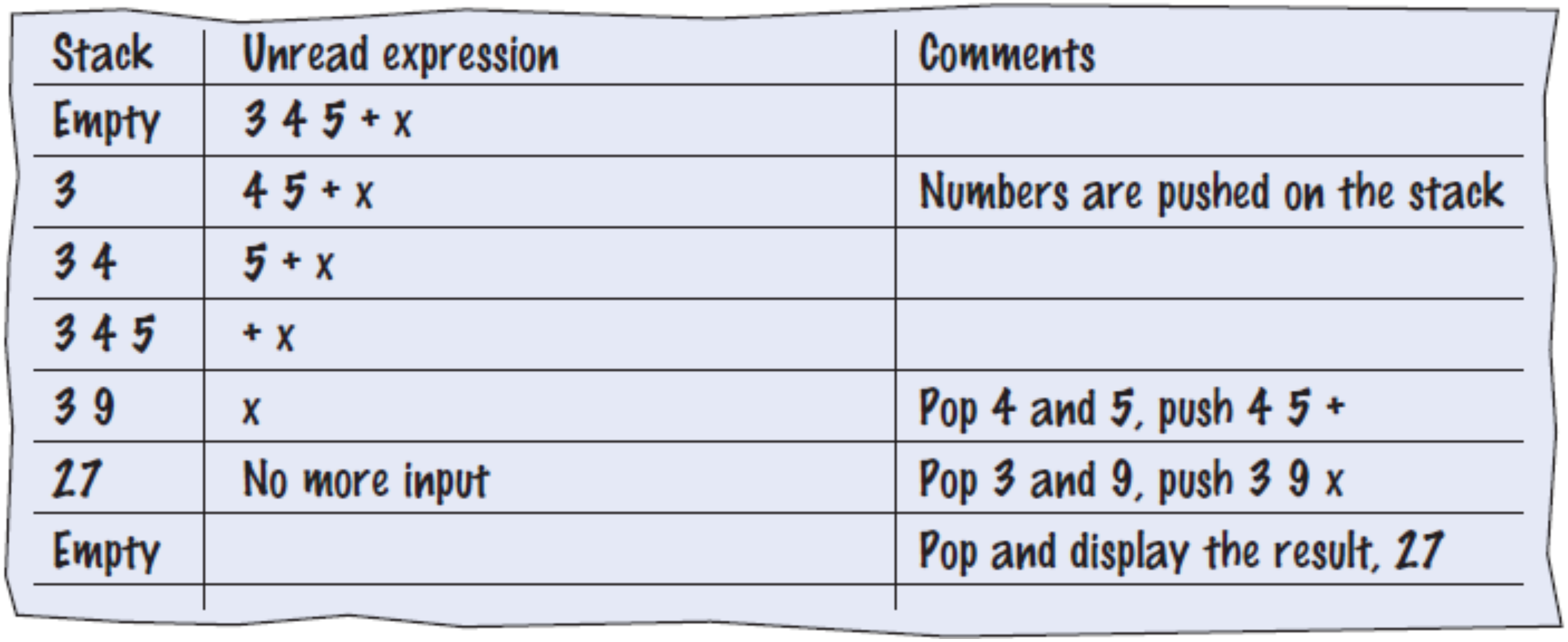
Algebra	Reverse Polish
(3 + 4) x 5	3 4 + 5 x
(3 + 4) x (5 + 6)	3 4 + 5 6 + x

Reverse Polish Expressions

```
If you read a number
    Push it on the stack.
Else if you read an operand
    Pop two values off the stack.
    Combine the values with the operand.
    Push the result back onto the stack.
Else if there is no more input
    Pop and display the result.
```

Reverse Polish Calculator

- Walkthrough with 3 4 5 + x



Stack	Unread expression	Comments
Empty	3 4 5 + x	
3	4 5 + x	Numbers are pushed on the stack
3 4	5 + x	
3 4 5	+ x	
3 9	x	Pop 4 and 5, push 4 5 +
27	No more input	Pop 3 and 9, push 3 9 x
Empty		Pop and display the result, 27

Calculator.java (1)

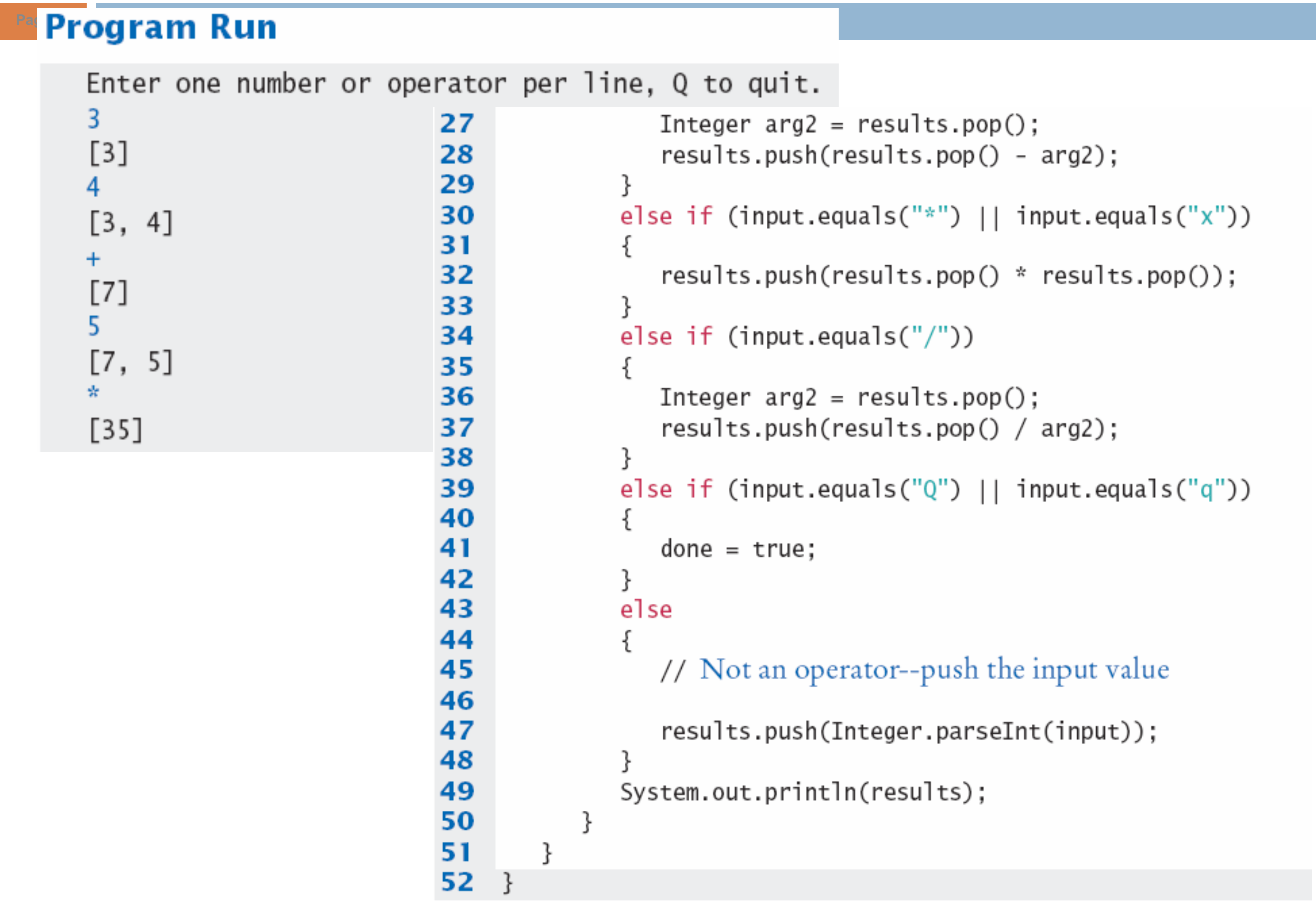
```java
import java.util.Scanner;
import java.util.Stack;

/**
   This calculator uses the reverse polish notation.
*/
public class Calculator
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      Stack<Integer> results = new Stack<Integer>();
      System.out.println("Enter one number or operator per line, Q to quit. ");
      boolean done = false;
      while (!done)
      {
         String input = in.nextLine();

         // If the command is an operator, pop the arguments and push the result

         if (input.equals("+"))
         {
            results.push(results.pop() + results.pop());
         }
         else if (input.equals("-"))
         {
```

Calculator.java (2)



Program Run

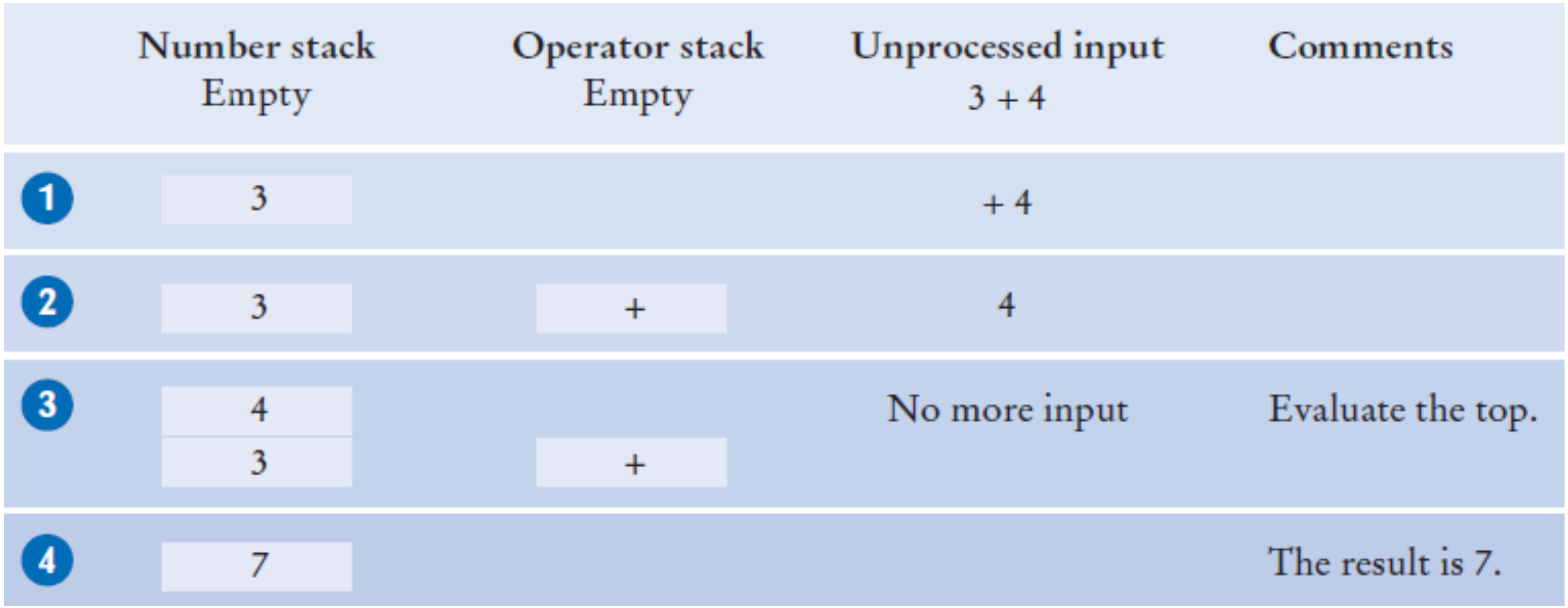
```
Enter one number or operator per line, Q to quit.
3
[3]
4
[3, 4]
+
[7]
5
[7, 5]
*
[35]
```

```java
            Integer arg2 = results.pop();
            results.push(results.pop() - arg2);
         }
         else if (input.equals("*") || input.equals("x"))
         {
            results.push(results.pop() * results.pop());
         }
         else if (input.equals("/"))
         {
            Integer arg2 = results.pop();
            results.push(results.pop() / arg2);
         }
         else if (input.equals("Q") || input.equals("q"))
         {
            done = true;
         }
         else
         {
            // Not an operator--push the input value

            results.push(Integer.parseInt(input));
         }
         System.out.println(results);
      }
   }
}
```

Evaluating Algebraic Expressions

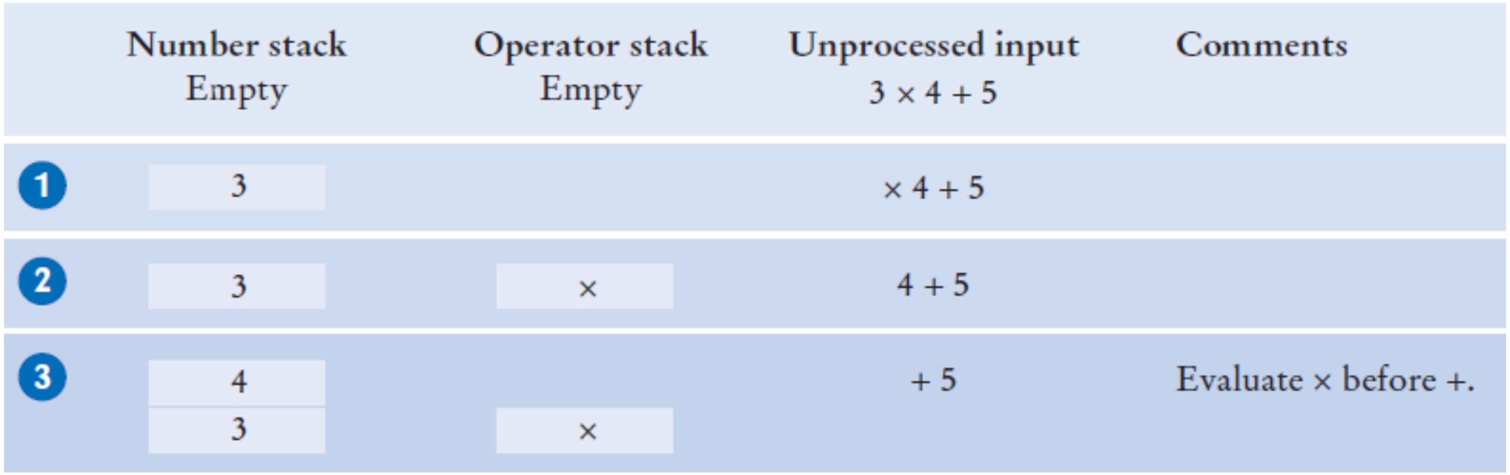
- Can be done with two stacks:
  1) Numbers
  2) Operators



	Number stack	Operator stack	Unprocessed input	Comments
	Empty	Empty	3 + 4	
1	3		+ 4	
2	3	+	4	
3	4 3	+	No more input	Evaluate the top.
4	7			The result is 7.

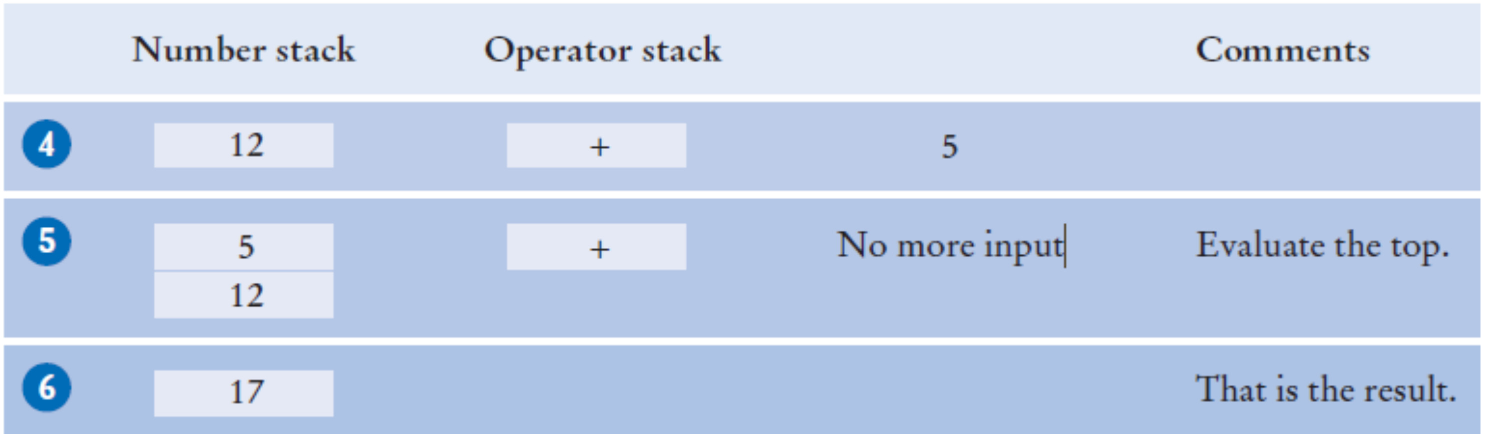
Expression Example 2

- Second Example: 3 x 4 + 5



	Number stack	Operator stack	Unprocessed input	Comments
	Empty	Empty	3 × 4 + 5	
1	3		× 4 + 5	
2	3	×	4 + 5	
3	4		+ 5	Evaluate × before +.
	3	×		

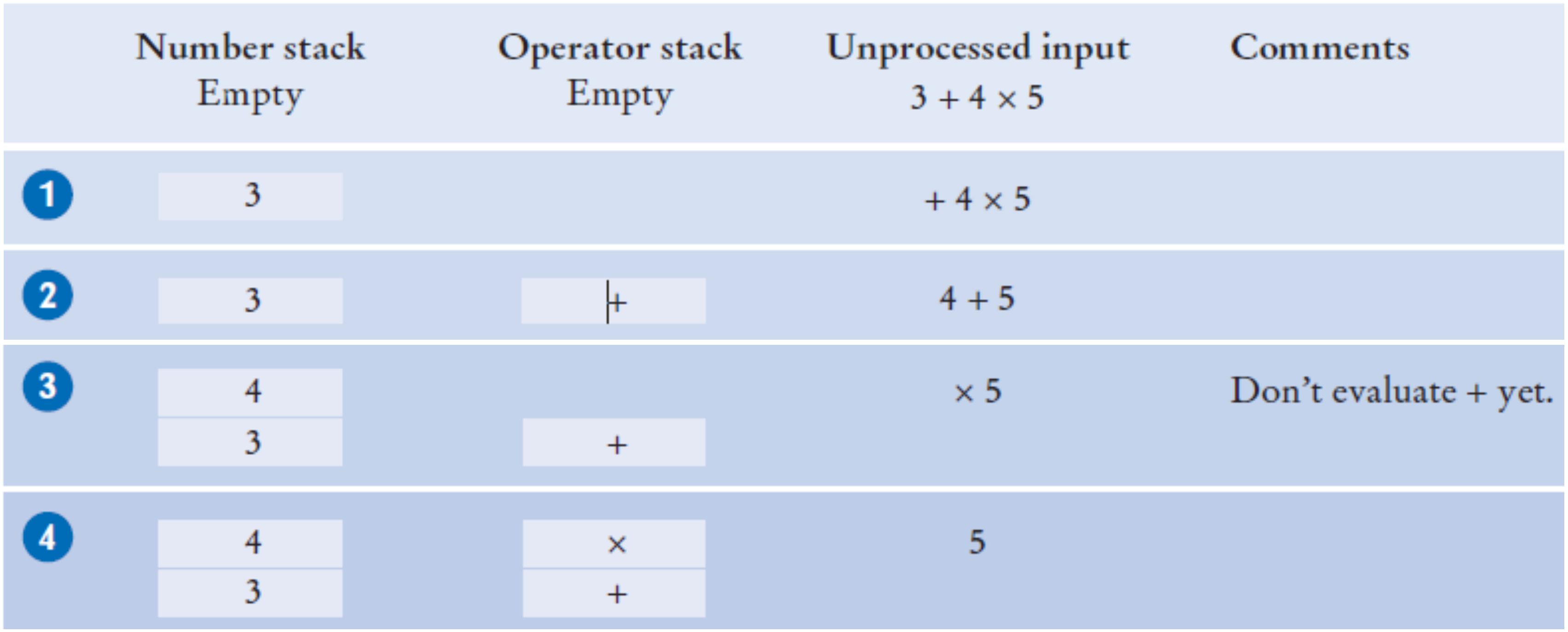
- Must use precedence (multiply before adding)



	Number stack	Operator stack		Comments
4	12	+	5	
5	5	+	No more input	Evaluate the top.
	12			
6	17			That is the result.

Precedence and Expressions (1)

- Third Example: 3 + 4 x 5

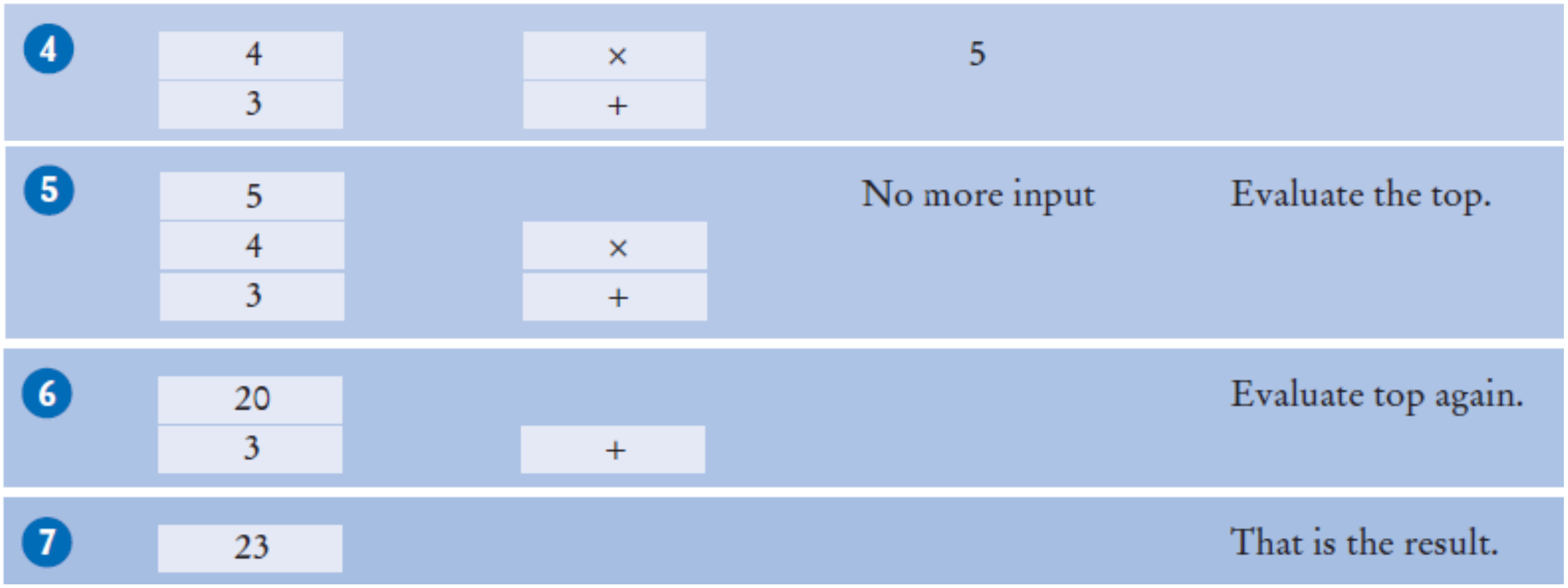


	Number stack	Operator stack	Unprocessed input	Comments
	Empty	Empty	3 + 4 × 5	
1	3		+ 4 × 5	
2	3	+	4 + 5	
3	4		× 5	Don't evaluate + yet.
	3	+		
4	4	×	5	
	3	+		

- Keep operators on the stack until they are ready to be evaluated

Precedence and Expressions (2)

- Third Example: 3 + 4 x 5

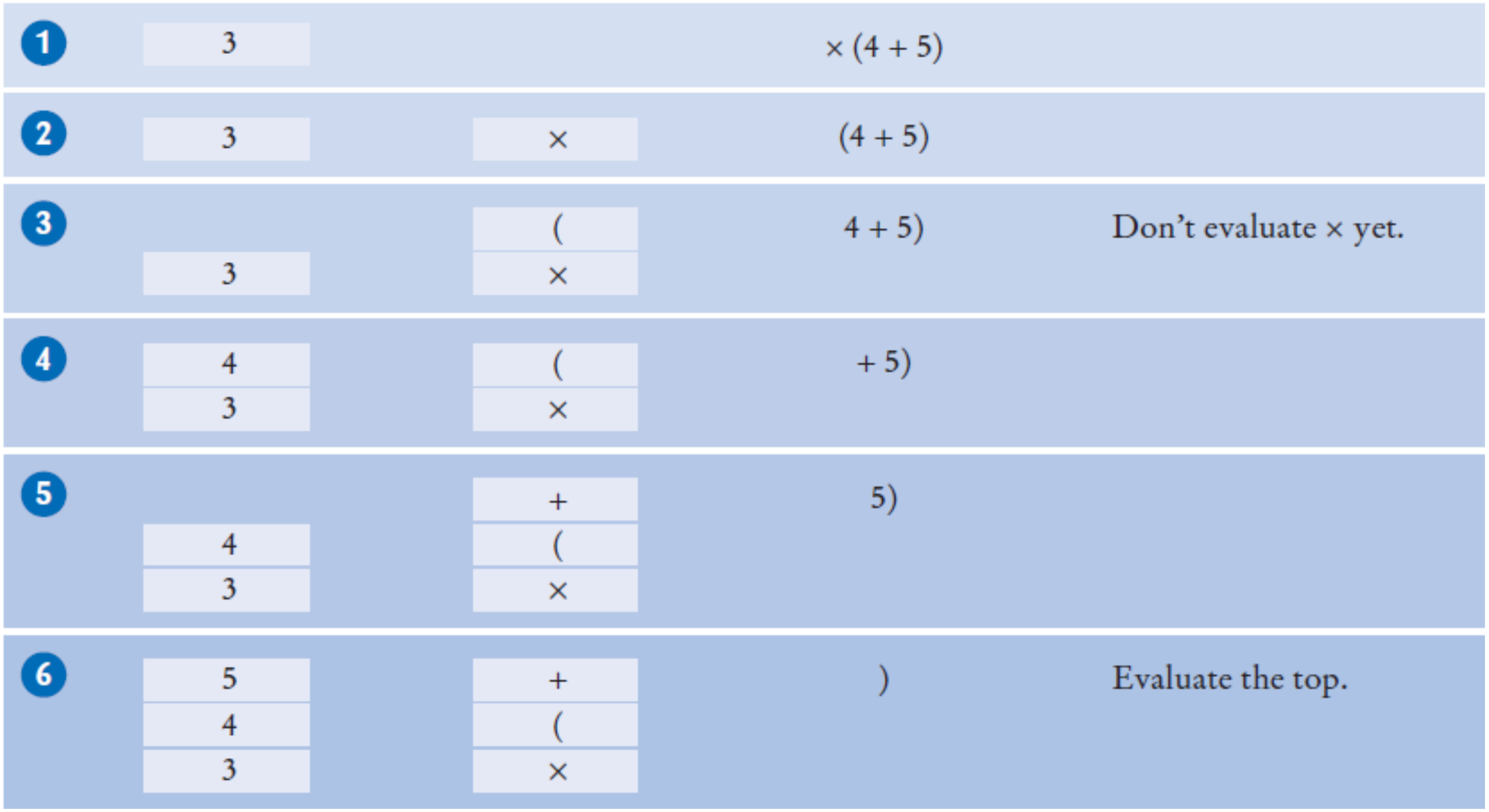


4	4 / 3	× / +	5	
5	5 / 4 / 3	× / +	No more input	Evaluate the top.
6	20 / 3	+		Evaluate top again.
7	23			That is the result.

  - Evaluate top 2 numbers with top operator
  - Store result on top of stack
  - Evaluate until operator stack is empty

Expressions with Parenthesis (1)

- Fourth Example: 3 x (4 + 5)

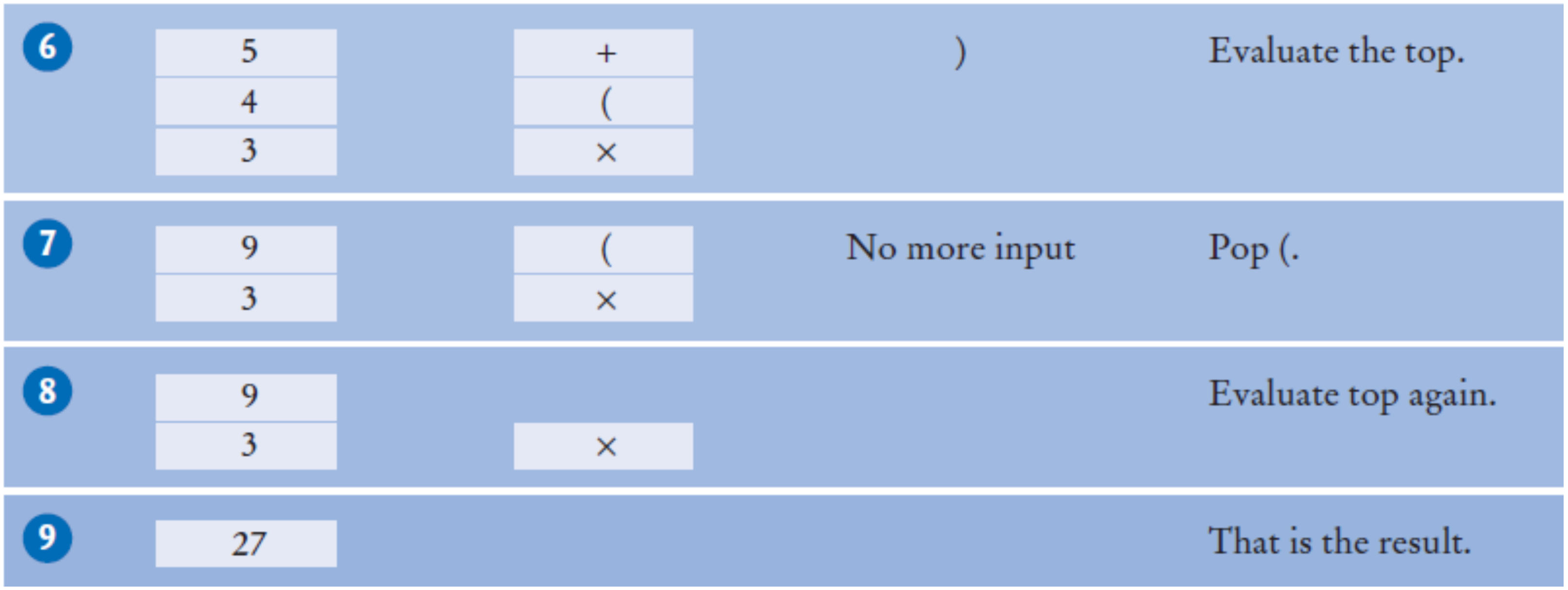


1	3		× (4 + 5)	
2	3	×	(4 + 5)	
3	3	(×	4 + 5)	Don't evaluate × yet.
4	4 3	(×	+ 5)	
5	4 3	+ (×	5)	
6	5 4 3	+ (×	)	Evaluate the top.

- If (, don't evaluate. If), evaluate

Expressions with Parenthesis (2)

- Fourth Example: 3 x (4 + 5)



6	5 4 3	+ (×	)	Evaluate the top.
7	9 3	(×	No more input	Pop (.
8	9 3	×		Evaluate top again.
9	27			That is the result.

- Don't put) on stack – evaluate now
- Ignore (

Precedence Algorithm

```
If you read a number
    Push it on the number stack.
Else if you read a (
    Push it on the operator stack.
Else if you read an operator op
    While the top of the stack has a higher precedence than op
        Evaluate the top.
    Push op on the operator stack.
Else if you read a )
    While the top of the stack is not a (
        Evaluate the top.
    Pop the ).
Else if there is no more input
    While the operator stack is not empty
        Evaluate the top.
```

<u>Evaluate the Top:</u>

Pop two numbers off the number stack.

Pop an operator off the operator stack.

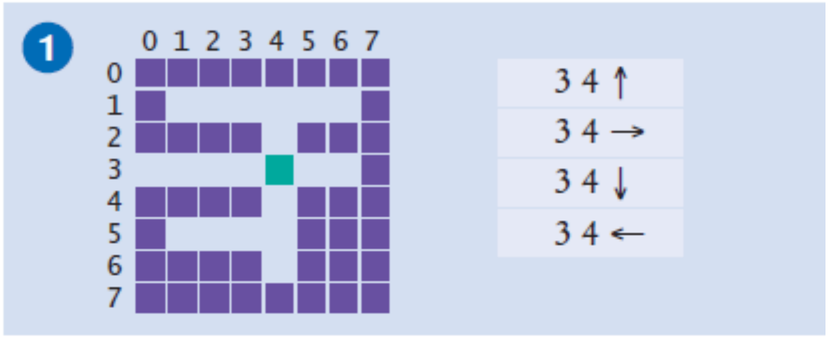
Combine the numbers with that operator.

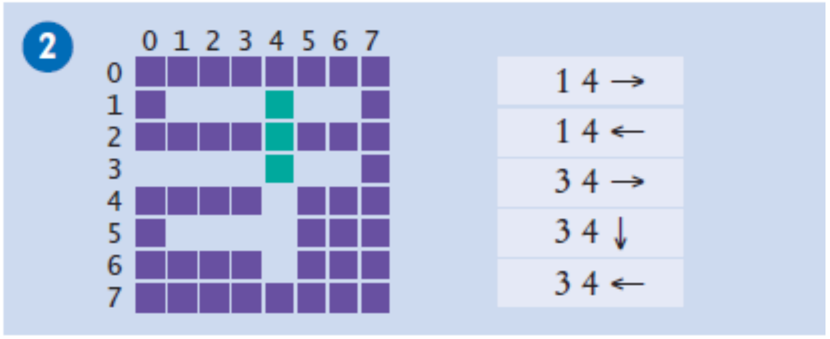
Push the result on the number stack.

Backtracking (1)

- Uses a stack to solve a maze
- Stack current location, arrow for each possible path

1. We pop off the topmost one, traveling north from (3, 4). Following this path leads to position (1, 4).
2. We now push two choices on the stack, going west or east.





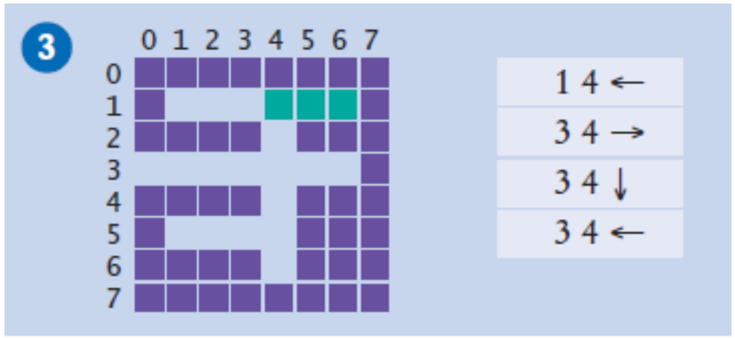
Backtracking (2)

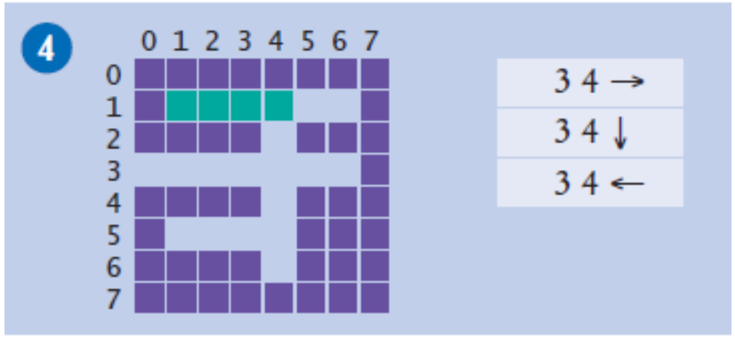
3. Pop off (1, 4) east. It is a dead end.

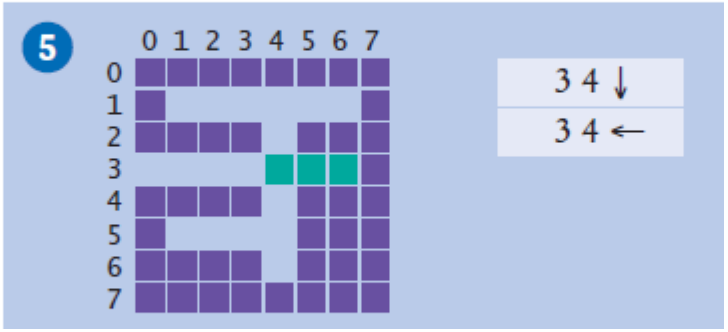
4. Pop off (1, 4) west. It is a dead end.

5. Now we pop off the path from (3, 4) going east. That too is a dead end.

This leaves us with (3, 4) south and (3, 4) west to try

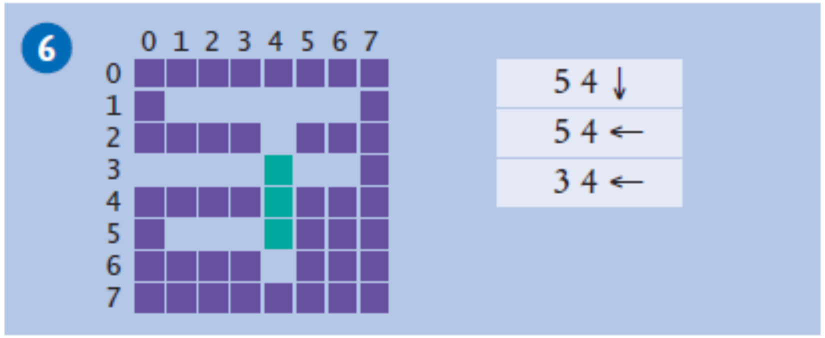


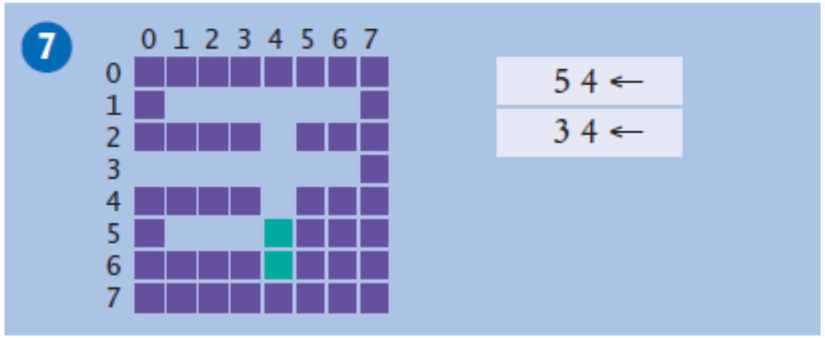


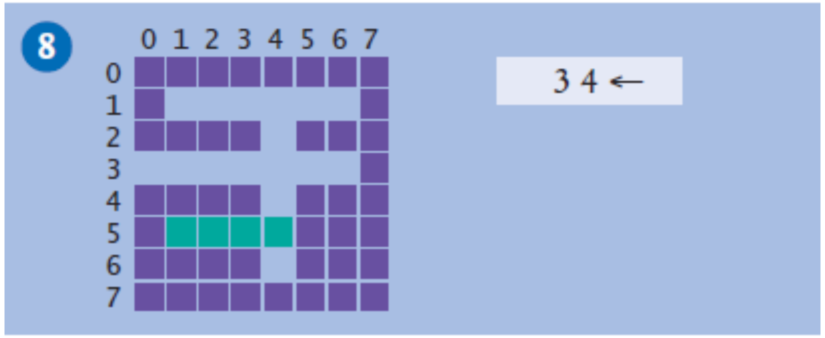


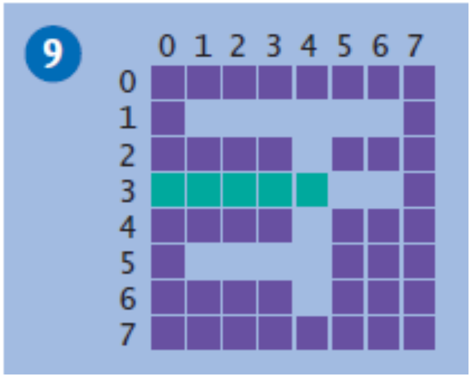
Backtracking (3)

6. Next is the path from (3, 4) going south. At (5, 4), it comes to an intersection. Both choices are pushed on the stack.
7. (5, 4) south is a dead end.
8. (5, 4) west is a dead end.
9. Finally, the path from (3, 4) going west leads to an exit









Maze Solving Pseudocode

```
Push all paths from the point on which you are standing
on a stack.
While the stack is not empty
    Pop a path from the stack.
    Follow the path until you reach an exit, intersection,
    or dead end.
    If you found an exit
        Congratulations!
    Else if you found an intersection
        Push all paths meeting at the intersection, except
        the current one, onto the stack.
```

Collections Summary

Steps to Choosing a Collection

1) Determine how you access values

- Values are accessed by an integer position. Use an `ArrayList`
  - Go to Step 2, then stop
- Values are accessed by a key that is not a part of the object
  - Use a `Map`.
- It doesn't matter. Values are always accessed "in bulk", by traversing the collection and doing something with each value

2) Determine the element types or key/value types

- For a `List` or `Set`, a single type
- For a `Map`, the key type and the value type

Steps to Choosing a Collection

3) Determine whether element or key order matters

- Elements or keys must be sorted
  - Use a `TreeSet` or `TreeMap`. Go to Step 6
- Elements must be in the same order in which they were inserted
  - Your choice is now narrowed down to a `LinkedList` or an `ArrayList`
- It doesn't matter
  - If you chose a map in Step 1, use a `HashMap` and go to Step 5

Steps to Choosing a Collection

4) For a collection, determine which operations must be fast

- Finding elements must be fast
  - Use a `HashSet` and go to Step 5
- Adding and removing elements at the beginning or the middle must be fast
  - Use a `LinkedList`
- You only insert at the end, or you collect so few elements that you aren't concerned about speed
  - Use an `ArrayList`.

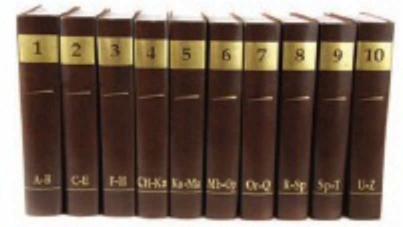
Steps to Choosing a Collection

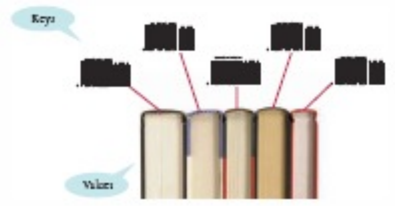
5) For hash sets and maps, decide if you need to implement the `equals` and `hashCode` methods
   - If your elements do not support them, you must implement them yourself.

6) If you use a tree, decide whether to supply a comparator
   - If your element class does not provide it, implement the Comparable interface for your element class

Summary: Collections

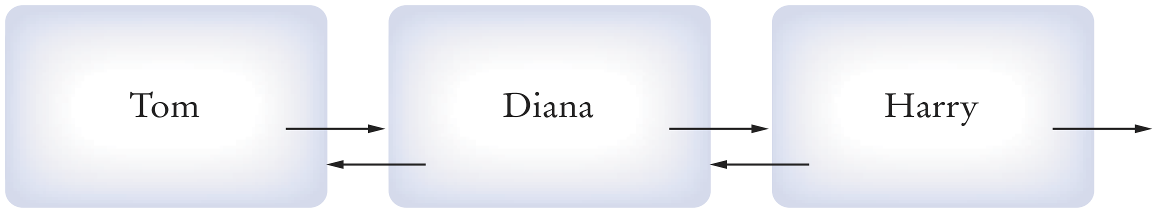
- A collection groups together elements and allows them to be retrieved later
  - A **list** is a collection that remembers the order of its elements
  - A **set** is an unordered collection of unique elements
  - A **map** keeps associations between key and value objects





Summary: Linked Lists

- A linked list consists of a number of nodes, each of which has a reference to the next node
  - Adding and removing elements in the middle of a linked list is efficient
  - Visiting the elements of a linked list in sequential order is efficient, but random access is not
  - You use a list iterator to access elements of a linked list



Summary: Choosing a Set

- The `HashSet` and `TreeSet` classes both implement the `Set` interface.
- Set implementations arrange the elements so that they can locate them quickly.
- You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.
- You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.
- Sets don't have duplicates. Adding a duplicate of an element that is already present is silently ignored.
- A set iterator visits the elements in the order in which the set implementation keeps them.
- You cannot add an element to a set at an iterator position.

Summary: Maps

- Maps associate keys with values
  - The `HashMap` and `TreeMap` classes both implement the `Map` interface
  - To find all keys and values in a `Map`, iterate through the key set and find the values that correspond to the keys
  - A hash function computes an integer value from an object.
  - A good hash function minimizes **collisions**—identical hash codes for different objects.
  - Override `hashCode` methods in your own classes by combining the hash codes for the instance variables.
  - A class's `hashCode` method must be compatible with its `equals` method.

Summary: Stacks and Queues

- A stack is a collection of elements with "last-in, first-out" retrieval.

- A queue is a collection of elements with "first-in, first-out" retrieval.
- When removing an element from a priority queue, the element with the most urgent priority is retrieved.

Summary: Problem Solving

- A stack can be used to check whether parentheses in an expression are balanced.
- Use a stack to evaluate expressions in reverse Polish notation.
- Using two stacks, you can evaluate expressions in standard algebraic notation.
- Use a stack to remember choices you haven't yet made so that you can backtrack to them.

Topics

1. Linked Lists
2. Stacks and Queues
3. Selecting a Collection