Chapter15 – Collections

Goals

- To learn how to use the collection classes supplied in the Java library
- To use iterators to traverse collections
- To choose appropriate collections for solving programming problems
- To study applications of stacks and queues

An Overview of the Collections Framework

- A collection groups together elements and allows them to be retrieved later.
- Java collections framework: a hierarchy of interface types and classes for collecting objects.
  - Each interface type is implemented by one or more classes

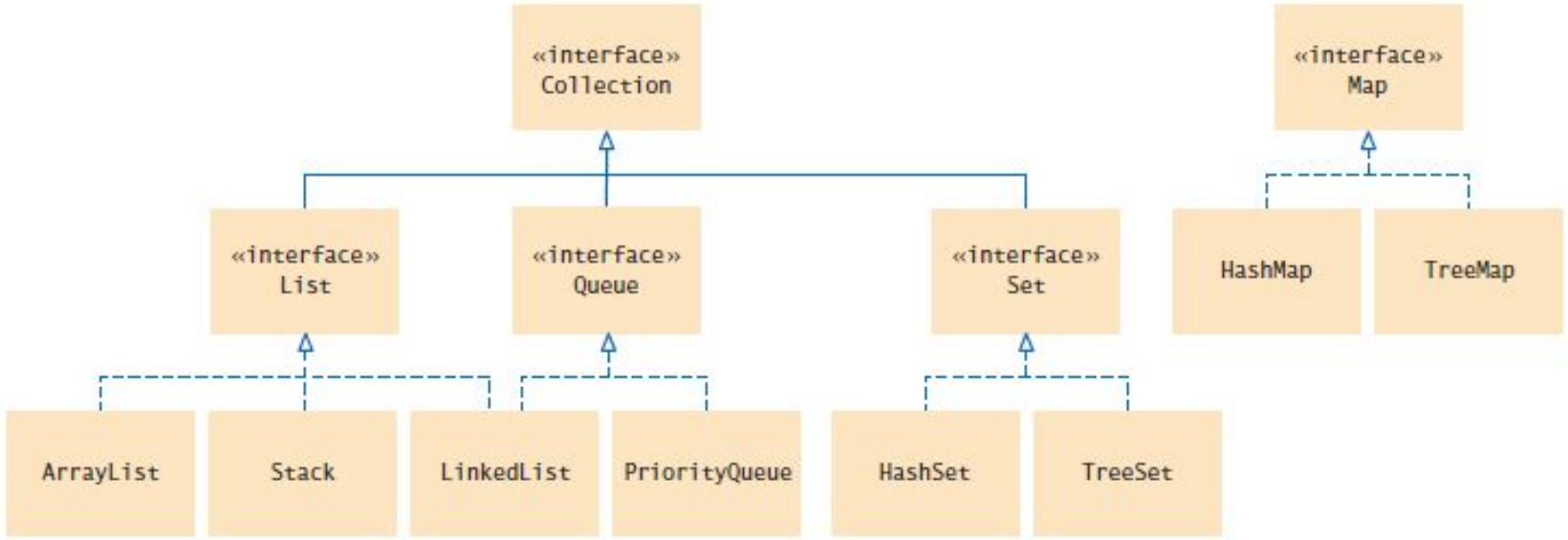


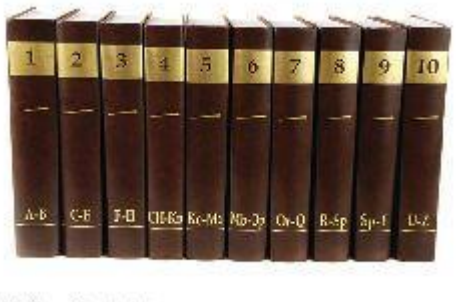
Figure 1 Interfaces and Classes in the Java Collections Framework

An Overview of the Collections Framework

- The Collection interface is at the root
  - All Collection classes implement this interface
  - So all have a common set of methods

An Overview of the Collections Framework

- List interface
- A list is a collection that remembers the order of its elements.
- Two implementing classes
  - ArrayList
  - LinkedList



Books

An Overview of the Collections Framework

- Set interface
- A set is an **unordered** collection of **unique** elements.
- Arranges its elements so that finding, adding, and removing elements is more efficient.
- Two mechanisms to do this
  - hash tables
  - binary search trees

Figure 3 A Set of Books

Difference between HashMap & Hashtable

	HashMap	Hashtable
Synchronized	No	Yes
Thread-Safe	No	Yes
Null Keys and Null values	One null key ,Any null values	Not permit null keys and values
Iterator type	Fail fast iterator	Fail safe iterator
Performance	Fast	Slow in comparison
Superclass and Legacy	AbstractMap , No	Dictionary , Yes

An Overview of the Collections Framework

- Stack
  - Remembers the order of elements
  - But you can only add and remove at the top

Figure 4 A Stack of Books

An Overview of the Collections Framework

- Queue
  - Add items to one end (the tail) and remove them from the other end (the head)
- A queue of People

- A priority queue
  - an unordered collection
  - has an efficient operation for removing the element with the highest priority

An Overview of the Collections Framework

- Map
  - Keeps associations between key and value objects.
  - Every key in the map has an associated value.
  - The map stores the keys, values, and the associations between them.

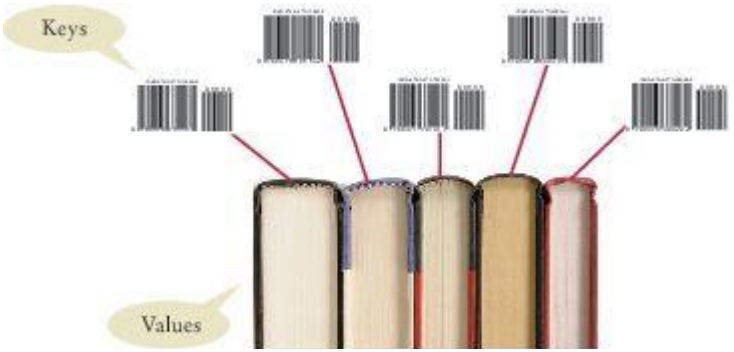


Figure 5 A Map from Bar Codes to Books

An Overview of the Collections Framework

- Every class that implements the Collection interface has these methods.

Table 1 The Methods of the Collection Interface

`Collection<String> coll = new ArrayList<String>();`	The `ArrayList` class implements the `Collection` interface.
`coll = new TreeSet<String>();`	The `TreeSet` class (Section 15.3) also implements the `Collection` interface.
`int n = coll.size();`	Gets the size of the collection. `n` is now `0`.
`coll.add("Harry");` `coll.add("Sally");`	Adds elements to the collection.
`String s = coll.toString();`	Returns a string with all elements in the collection. `s` is now `[Harry, Sally]`.
`System.out.println(coll);`	Invokes the `toString` method and prints `[Harry, Sally]`.
`coll.remove("Harry");` `boolean b = coll.remove("Tom");`	Removes an element from the collection, returning `false` if the element is not present. `b` is false.
`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

`for (String s : coll)`
`{`
`  System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

`Iterator<String> iter = coll.iterator();`	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

Self Check 15.1

A teacher's gradebook program stores a collection of quizzes. Should it use a list or a set?

Answer: A list is a better choice because the application will want to retain the order in which the quizzes were given.

Self Check 15.2

A student information system stores a collection of student records for a university. Should it use a list or a set?

Answer: A set is a better choice. There is no intrinsically useful ordering for the students. For example, the registrar's office has little use for a list of all students by their GPA. By storing them in a set, adding, removing, and finding students can be efficient.

Self Check 15.3

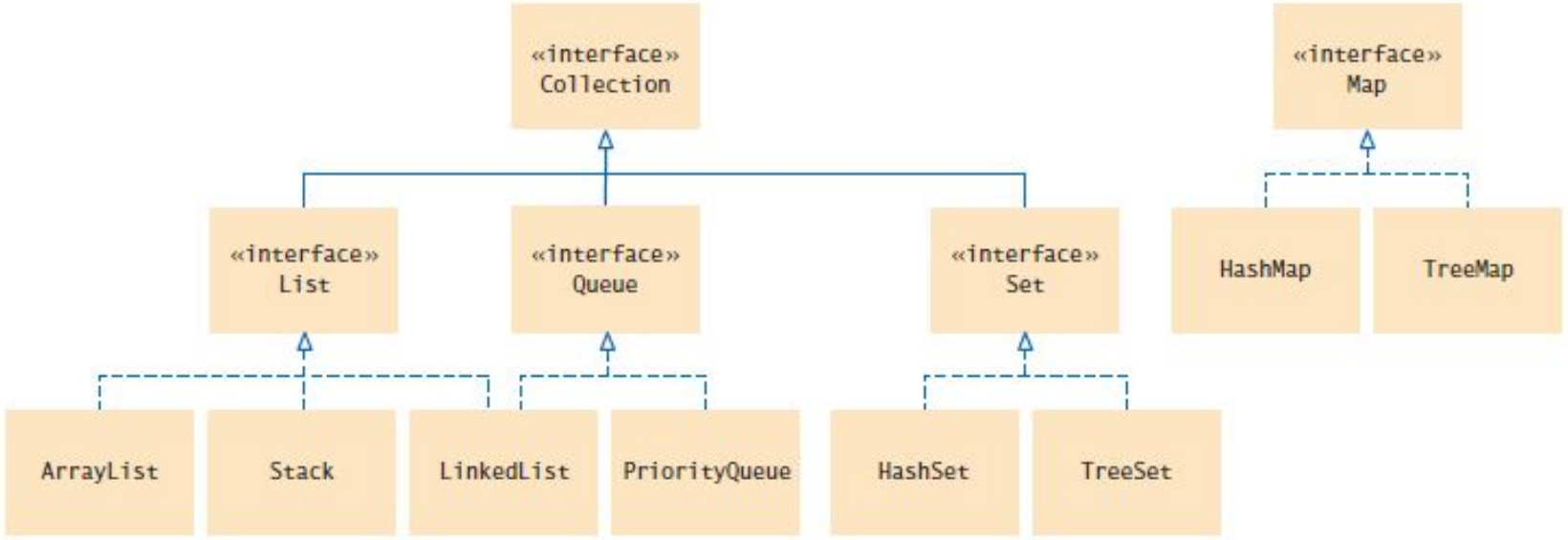
Why is a queue of books a better choice than a stack for organizing your required reading?

Answer: With a stack, you would always read the latest required reading, and you might never get to the oldest readings.

Self Check 15.4

As you can see from Figure 1 below, the Java collections framework does not consider a map a collection. Give a reason for this decision.

- **Answer:** A collection stores elements, but a map stores *associations* between elements.



Linked Lists

- A data structure used for collecting a sequence of objects:
  - Allows efficient addition and removal of elements in the middle of the sequence.
- A linked list consists of a number of nodes;
  - Each node has a reference to the next node.
- A node is an object that stores an element and references to the neighboring nodes.
- Each node in a linked list is connected to the neighboring nodes.

Linked Lists

- 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.
- Random access is **not** efficient.

Figure 6 Example of a linked list

Linked Lists

- When inserting or removing a node:
  - Only the neighboring node references need to be updated

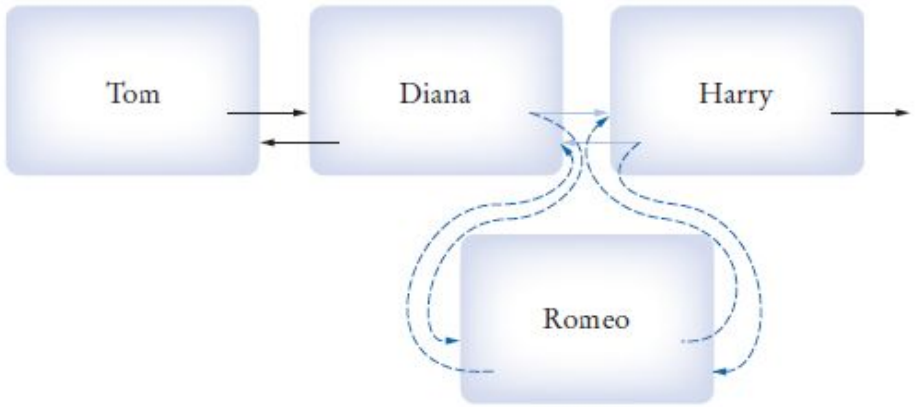


Figure 7 Inserting a Node into a Linked List

Linked Lists

Figure 8 Removing a Node From A Linked List

- Visiting the elements of a linked list in sequential order is efficient.
- Random access is not efficient.
- When to use a linked list:
  - You are concerned about the efficiency of inserting or removing elements
  - You rarely need element access in random order

The LinkedList Class of the Java Collections Framework

- Generic class
  - Specify type of elements in angle brackets: LinkedList<Product>
- Package: java.util
- LinkedList has the methods of the Collection interface.

The LinkedList Class of the Java Collections Framework

- Some additional LinkedList 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 678).

List Iterator

- Use a list iterator to access elements inside a linked list.
- Encapsulates a position anywhere inside the linked list.
- Think of an iterator as pointing between two elements:
  - Analogy: like the cursor in a word processor points between two characters
- To get a list iterator, use the listIterator method of the LinkedList class.

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

- Also a generic type.

List Iterator

- Initially points before the first element.
- Move the position with next method:

```
if (iterator.hasNext())
{
   iterator.next();
}
```

- The next method returns the element that the iterator is passing.
- The return type of the next method matches the list iterator's type parameter.

List Iterator

- To traverse all elements in a linked list of strings:

```
while (iterator.hasNext())
{
   String name = iterator.next();
   Do something with name
}
```

- To use the “for each” loop:

```
for (String name : employeeNames)
{
   Do something with name
}
```

List Iterator

- The nodes of the LinkedList class store two links:
  - One to the next element
  - One to the previous one
  - Called a doubly-linked list
- To move the list position backwards, use:
  - hasPrevious
  - previous

List Iterator

- The add method adds an object after the iterator.
  - Then moves the iterator position past the new element.

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

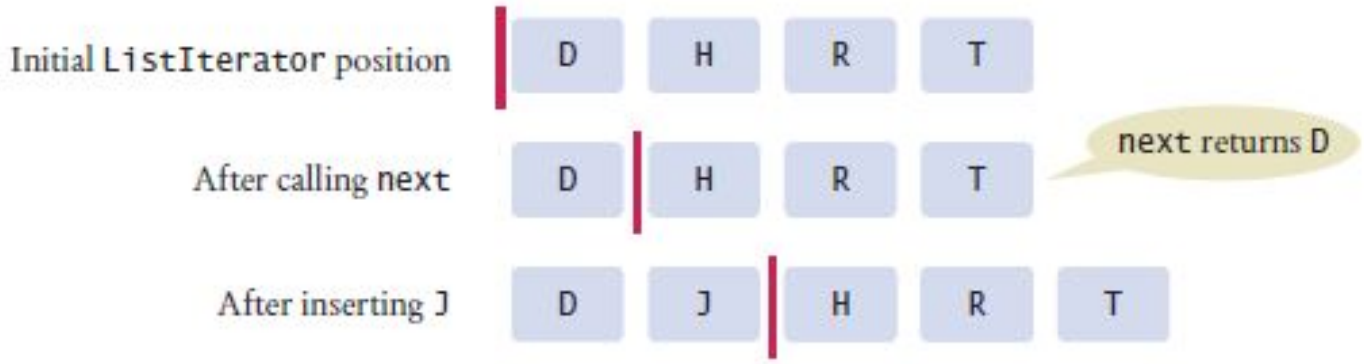


Figure 8 A Conceptual View of the List Iterator

List Iterator

- The remove method:
  - Removes object that was returned by the last call to next or previous
- To remove all names that fulfill a certain condition:

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

- Be careful when calling remove:
  - It can be called only **once** after calling next or previous
  - You cannot call it immediately after a call to add
  - If you call it improperly, it throws an IllegalStateException

List Iterator

- ListIterator interface extends Iterator interface.
- Methods of the Iterator and ListIterator interfaces

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]`.

Sample Program

- ListDemo is a sample program that:
  - Inserts strings into a list
  - Iterates through the list, adding and removing elements
  - Prints the list

section_2/ListDemo.java

```
 1  import java.util.LinkedList;
 2  import java.util.ListIterator;
 3
 4  /**
 5     This program demonstrates the LinkedList class.
 6  */
 7  public class ListDemo
 8  {
 9     public static void main(String[] args)
10     {
11        LinkedList<String> staff = new LinkedList<String>();
12        staff.addLast("Diana");
13        staff.addLast("Harry");
14        staff.addLast("Romeo");
15        staff.addLast("Tom");
16
17        // | in the comments indicates the iterator position
18
19        ListIterator<String> iterator = staff.listIterator(); // |DHRT
20        iterator.next(); // D|HRT
21        iterator.next(); // DH|RT
22
23        // Add more elements after second element
24
25        iterator.add("Juliet"); // DHJ|RT
26        iterator.add("Nina"); // DHJN|RT
27
28        iterator.next(); // DHJNR|T
29
30        // Remove last traversed element
31
```

Continued

section_2/ListDemo.java

```
32          iterator.remove(); // DHJN|T
33
34          // Print all elements
35
36          System.out.println(staff);
37          System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38      }
39  }
```

Program Run:

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

Self Check 15.5

Do linked lists take more storage space than arrays of the same size?

Answer: Yes, for two reasons. A linked list needs to store the neighboring node references, which are not needed in an array. Moreover, there is some overhead for storing an object. In a linked list, each node is a separate object that incurs this overhead, whereas an array is a single object.

Self Check 15.6

Why don't we need iterators with arrays?

Answer: We can simply access each array element with an integer index.

Self Check 15.7

Suppose the list letters contains elements "A", "B", "C", and "D". Draw the contents of the list and the iterator position for the following operations:

```
ListIterator<String> iter = letters.iterator();
iter.next();
iter.next();
iter.remove();
iter.next();
iter.add("E");
iter.next();
iter.add("F");
```

Answer:

|ABCD
A|BCD
AB|CD
A|CD
AC|D
ACE|D
ACED|
ACEDF|

Self Check 15.8

Write a loop that removes all strings with length less than four from a linked list of strings called words.

Answer:

```
ListIterator<String> iter = words.iterator();
while (iter.hasNext())
{
   String str = iter.next();
   if (str.length() < 4) { iter.remove(); }
}
```

Self Check 15.9

Write a loop that prints every second element of a linked list of strings called words.

Answer:

```
ListIterator<String> iter = words.iterator();
while (iter.hasNext())
{
   System.out.println(iter.next());
   if (iter.hasNext())
   {
      iter.next(); // Skip the next element
   }
}
```

Sets

- A set organizes its values in an order that is optimized for efficiency.
- May not be the order in which you add elements.
- Inserting and removing elements is more efficient with a set than with a list.

Sets

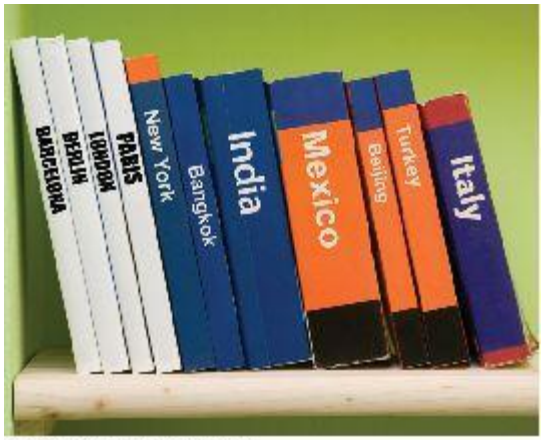
- The Set interface has the same methods as the Collection interface.
- A set does not admit duplicates.
- Two implementing classes
  - HashSet
    - based on hash table
  - TreeSet
    - based on binary search tree
- A Set implementation arranges the elements so that it can locate them quickly.

Sets

- In a hash table
  - Set elements are grouped into smaller collections of elements that share the same characteristic.
    - Grouped by an integer hash code
    - Computed from the element
- Elements in a hash table must implement the method hashCode. Must have a properly defined equals method.
- You can form hash sets holding objects of type String, Integer, Double, Point, Rectangle, or Color.
  - HashSet<String>, HashSet<Rectangle>, or a HashSet<HashSet<Integer>>

Sets

- On this shelf, books of the same color are grouped together. Similarly, in a hash table, objects with the same hash code are placed in the same group



Sets

- In a TreeSet
  - Elements are kept in sorted order

- Elements are stored in nodes.
- The nodes are arranged in a tree shape,
  - Not in a linear sequence
- You can form tree sets for any class that implements the Comparable interface:
  - Example: String or Integer.

Sets

- Use a TreeSet if you want to visit the set's elements in sorted order.
  - Otherwise choose a HashSet
    - It is a bit more efficient — if the hash function is well chosen

Sets

- Store the reference to a TreeSet or HashSet in a Set<String> variable:

```
Set<String> names = new HashSet<String>();
```

Or

```
Set<String> names = new TreeSet<String>();
```

- After constructing the collection object:
  - The implementation no longer matters
  - Only the interface is important

Working with Sets

- Adding and removing elements:

```
names.add("Romeo");
names.remove("Juliet");
```

- Sets don't have duplicates.
  - Adding a duplicate is ignored.
- Attempting to remove an element that isn't in the set is ignored.
- The contains method tests whether an element is contained in the set:

```
if (names.contains("Juliet")) . . .
```

  - The contains method uses the equals method of the element type

Working with Sets

- To process all elements in the set, get an iterator.
- A set iterator visits the elements in the order in which the set implementation keeps them.

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}
```

- You can also use the “for each” loop

```
for (String name : names)
{
   Do something with name
}
```

Working with Sets

- You cannot add an element to a set at an iterator position - A set is unordered.
- You can remove an element at an iterator position.
- The Iterator interface as no previous method.

Working with Sets

Table 4 Working with Sets	
`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<String>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.
`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.

`for (String name : names)`
`{`
`. . .`
`}`
	Use this loop to visit all elements of a set.

`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

SpellCheck Example Program

- Read all the correctly spelled words from a dictionary file
  - Put them in a set
- Reads all words from a document
  - Put them in a second set
- Print all the words in the second set that are not in the dictionary set.
- Potential misspellings

section_3/SpellCheck.java

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3  import java.util.Set;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6
7  /**
8     This program checks which words in a file are not present in a dictionary.
9  */
10 public class SpellCheck
11 {
12    public static void main(String[] args)
13       throws FileNotFoundException
14    {
15       // Read the dictionary and the document
16
17       Set<String> dictionaryWords = readWords("words");
18       Set<String> documentWords = readWords("alice30.txt");
19
20       // Print all words that are in the document but not the dictionary
21
22       for (String word : documentWords)
23       {
24          if (!dictionaryWords.contains(word))
25          {
26             System.out.println(word);
27          }
28       }
29    }
30
```

Continued

section_3/SpellCheck.java

```
31      /**
32         Reads all words from a file.
33         @param filename the name of the file
34         @return a set with all lowercased words in the file. Here, a
35         word is a sequence of upper- and lowercase letters.
36      */
37      public static Set<String> readWords(String filename)
38         throws FileNotFoundException
39      {
40         Set<String> words = new HashSet<String>();
41         Scanner in = new Scanner(new File(filename));
42         // Use any characters other than a-z or A-Z as delimiters
43         in.useDelimiter("[^a-zA-Z]+");
44         while (in.hasNext())
45         {
46            words.add(in.next().toLowerCase());
47         }
48         return words;
49      }
50   }
```

Program Run:

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
```

Self Check 15.10

Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?

Answer: Adding and removing elements as well as testing for membership is more efficient with sets.

Self Check 15.11

Why are set iterators different from list iterators?

Answer: Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backward.

Self Check 15.12

What is wrong with the following test to check whether the Set<String> s contains the elements "Tom", "Diana", and "Harry"?

```
if (s.toString().equals("[Tom, Diana, Harry]")) . . .
```

Answer: You do not know in which order the set keeps the elements.

Self Check 15.13

How can you correctly implement the test of Self Check 12?

Answer: Here is one possibility:

```
if (s.size() == 3 && s.contains("Tom")
   && s.contains("Diana")
   && s.contains("Harry")) . . .
```

Self Check 15.14

Write a loop that prints all elements that are in both

Set<String> s and Set<String> t.

Answer:

```
for (String str : s)
{
   if (t.contains(str))
   {
      System.out.println(str);
   }
}
```

Self Check 15.15

Suppose you changed line 40 of the SpellCheck program to use a TreeSet instead of a HashSet. How would the output change?

Answer: The words would be listed in sorted order.

Maps

- A map allows you to associate elements from a **key set** with elements from a **value collection**.
- Use a map when you want to look up objects by using a key.

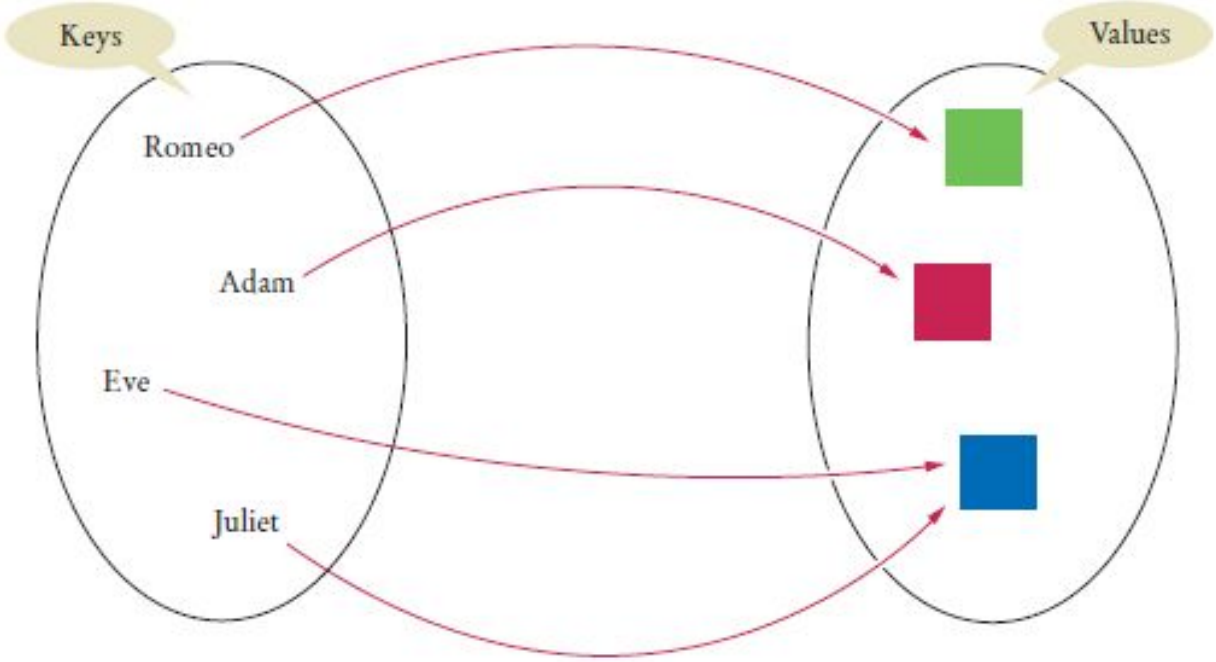


Figure 10 A Map

Maps

- Two implementations of the Map interface:
  - HashMap
  - TreeMap
- Store the reference to the map object in a Map reference:

```
Map<String, Color> favoriteColors =
   new HashMap<String, Color>();
```

Maps

- Use the put method to add an association:

```
favoriteColors.put("Juliet", Color.RED);
```

- You can change the value of an existing association by calling put again:

```
favoriteColors.put("Juliet", Color.BLUE);
```

- The get method returns the value associated with a key:

```
Color favorite = favorite.get("Juliet");
```

- If you ask for a key that isn't associated with any values, the get method returns null.
- To remove an association, call the remove method with the key:

```
favoriteColors.remove("Juliet");
```

Working with Maps

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<String, Integer>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

`for (String key : scores.keySet())`
`{`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Maps

- Sometimes you want to enumerate all keys in a map.
- The keySet method yields the set of keys.
- Ask the key set for an iterator and get all keys.
- For each key, you can find the associated value with the get method.
- To print all key/value pairs in a map m:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

section_4/MapDemo.java

```
 1  import java.awt.Color;
 2  import java.util.HashMap;
 3  import java.util.Map;
 4  import java.util.Set;
 5
 6  /**
 7     This program demonstrates a map that maps names to colors.
 8  */
 9  public class MapDemo
10  {
11     public static void main(String[] args)
12     {
13        Map<String, Color> favoriteColors = new HashMap<String, Color>();
14        favoriteColors.put("Juliet", Color.BLUE);
15        favoriteColors.put("Romeo", Color.GREEN);
16        favoriteColors.put("Adam", Color.RED);
17        favoriteColors.put("Eve", Color.BLUE);
18
19        // Print all keys and values in the map
20
21        Set<String> keySet = favoriteColors.keySet();
22        for (String key : keySet)
23        {
24           Color value = favoriteColors.get(key);
25           System.out.println(key + " : " + value);
26        }
27     }
28  }
```

Continued

section_4/MapDemo.java

Program Run:

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

Self Check 15.16

What is the difference between a set and a map?

Answer: A set stores elements. A map stores associations between keys and values.

Self Check 15.17

Why is the collection of the keys of a map a set and not a list?

Answer: The ordering does not matter, and you cannot have duplicates

Self Check 15.18

Why is the collection of the values of a map not a set?

Answer: Because it might have duplicates.

Self Check 15.19

Suppose you want to track how many times each word occurs in a document. Declare a suitable map variable.

Answer:

```
Map<String, Integer> wordFrequency;
```

Self Check 15.20

What is a Map<String, HashSet<String>>? Give a possible use for such a structure.

Answer: It associates strings with sets of strings. One application would be a thesaurus that lists synonyms for a given word. For example, the key "improve" might have as its value the set ["ameliorate", "better", "enhance", "enrich", "perfect", "refine"].

Choosing a Collection

- Determine how you access the values.
- Determine the element types or key/value types.
- Determine whether element or key order matters.
- For a collection, determine which operations must be efficient.
- For hash sets and maps, decide whether you need to implement the hashCode and equals methods.
- If you use a tree, decide whether to supply a comparator.

Hash Functions

- You may need to implement a hash function for your own classes.
- **A hash function**: a function that computes an integer value, the hash code, from an object in such a way that different objects are likely to yield different hash codes.
- Object class has a hashCode method
  - you need to override it to use your class in a hash table
- A collision: two or more objects have the same hash code.

Hash Functions

- The method used by the String class to compute the hash code.

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
   h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

- This produces different hash codes for "tea" and "eat".

Hash Functions

© one clear vision/iStockphoto.

A good hash function produces different hash values for each object so that they are scattered about in a hash table.

Hash Functions

- Override hashCode methods in your own classes by combining the hash codes for the instance variables.
- A hash function for our Country class:

```
public class Country
{
   public int hashCode()
   {
      int h1 = name.hashCode();
      int h2 = new Double(area).hashCode();
      final int HASH_MULTIPLIER = 29;
      int h = HASH_MULTIPLIER * h1 + h2;
      return h;
   }
}
```

- A class's hashCode method must be compatible with its equals method.

Stacks

- A stack lets you insert and remove elements only at one end:
  - Called the top of the stack.
  - Removes items in the opposite order than they were added
  - Last-in, first-out or LIFO order
- Add and remove methods are called push and pop.

Stacks

- Example

```
Stack<String> s = new Stack<String>();
s.push("A");
s.push("B");
s.push("C");
while (s.size() > 0)
{
   System.out.print(s.pop() + " "); // Prints C B A
}
```

- The last pancake that has been added to this stack will be the first one that is consumed.

Stacks

- Many applications for stacks in computer science.
- Consider: Undo function of a word processor
  - The issued commands are kept in a stack.
  - When you select “Undo”, the **last** command is popped off the stack and undone

- Run-time stack that a processor or virtual machine:
  - Stores the values of variables in nested methods.
  - When a new method is called, its parameter variables and local variables are pushed onto a stack.
  - When the method exits, they are popped off again.

Stack in the Java Library

- Stack class provides push, pop and peek methods.

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.

Queue

- A queue
  - Lets you add items to one end of the queue (the tail)
  - Remove items from the other end of the queue (the head)
  - Items are removed in the same order in which they were added
  - First-in, first-out or FIFO order
- To visualize a queue, think of people lining up.

Photodisc/Punchstock

- Typical application: a print queue.

Queue

- The Queue interface in the standard Java library has: an add method to add an element to the tail of the queue,
- A remove method to remove the head of the queue, and
- A peek method to get the head element of the queue without removing it.
- The LinkedList class implements the Queue interface. When you need a queue, initialize a Queue variable with a LinkedList object:

```
Queue<String> q = new LinkedList<String>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " "); }
// Prints A B C
```

Queue

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.

Priority Queue

- 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.
- Does not maintain a first-in, first-out discipline.
- Elements are retrieved according to their priority.
- Priority 1 denotes the most urgent priority
  - Each removal extracts the minimum element
- When you retrieve an item from a priority queue, you always get the most urgent one.

Priority Queue

- Example: objects of a class WorkOrder into a priority 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"));
```

- When calling q.remove() for the first time, the work order with priority 1 is removed.
- Elements should belong to a class that implements the Comparable interface.

Priority Queue

Table 9 Working with Priority Queues

`PriorityQueue<Integer> q = new PriorityQueue<Integer>();`	This priority queue holds `Integer` objects. 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 most urgent item: `first` is set to 1, `second` to 2.
`int next = q.peek();`	Gets the smallest value in the priority queue without removing it.

Self Check 15.21

Why would you want to declare a variable as

```
Queue<String> q = new LinkedList<String>();
```

instead of simply declaring it as a linked list?

Answer: This way, we can ensure that only queue operations can be invoked on the `q` object.

Self Check 15.22

Why wouldn't you want to use an array list for implementing a queue?

Answer: Depending on whether you consider the 0 position the head or the tail of the queue, you would either add or remove elements at that position. Both are inefficient operations because all other elements need to be moved.

Self Check 15.23

What does this code print?

```
Queue<String> q = new LinkedList<String>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0)
{
   System.out.print(q.remove() + " ");
}
```

Answer: A B C

Self Check 15.24

Why wouldn't you want to use a stack to manage print jobs?

Answer: Stacks use a “last-in, first-out” discipline. If you are the first one to submit a print job and lots of people add print jobs before the printer has a chance to deal with your job, they get their printouts first, and you have to wait until all other jobs are completed.

Self Check 15.25

In the sample code for a priority queue, we used a WorkOrder class. Could we have used strings instead?

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

Answer: Yes--the smallest string (in lexicographic ordering) is removed first. In the example, that is the string starting with 1, then the string starting with 2, and so on. However, the scheme breaks down if a priority value exceeds 9. For example, a string "10 - Line up braces" comes before "2 - Order cleaning supplies" in lexicographic order.

Sorting and Searching in the Java Library - Sorting

- You do not need to write sorting and searching algorithms
  - Use methods in the Arrays and Collections classes
- The Arrays class contains static sort methods.
- To sort an array of integers:

```
int[] a = . . . ;
Arrays.sort(a);
```

  - That sort method uses the Quicksort algorithm (see Special Topic 14.3).
- To sort an ArrayList, use Collections.sort

```
ArrayList<String> names = . . .;
Collections.sort(names);
```

  - Uses merge sort algorithm

Stack and Queue Applications

- A stack can be used to check whether parentheses in an expression are balanced.

  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.

Stack and Queue Applications

- Walkthrough of the sample expression:

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

Stack and Queue Applications

- Use a stack to evaluate expressions in reverse Polish notation.

```
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.
```

Stack and Queue Applications

- Walkthrough of evaluating the expression 3 4 5 + ×:

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

section_6_2/Calculator.java

```
 1  import java.util.Scanner;
 2  import java.util.Stack;
 3
 4  /**
 5     This calculator uses the reverse Polish notation.
 6  */
 7  public class Calculator
 8  {
 9     public static void main(String[] args)
10     {
11        Scanner in = new Scanner(System.in);
12        Stack<Integer> results = new Stack<Integer>();
13        System.out.println("Enter one number or operator per line, Q to quit. ");
14        boolean done = false;
15        while (!done)
16        {
17           String input = in.nextLine();
18
19           // If the command is an operator, pop the arguments and push the result
20
21           if (input.equals("+"))
22           {
23              results.push(results.pop() + results.pop());
24           }
25           else if (input.equals("-"))
26           {
27              Integer arg2 = results.pop();
28              results.push(results.pop() - arg2);
29           }
```

Continued

section_6_2/Calculator.java

```
30            else if (input.equals("*") || input.equals("x"))
31            {
32               results.push(results.pop() * results.pop());
33            }
34            else if (input.equals("/"))
35            {
36               Integer arg2 = results.pop();
37               results.push(results.pop() / arg2);
38            }
39            else if (input.equals("Q") || input.equals("q"))
40            {
41               done = true;
42            }
43            else
44            {
45               // Not an operator--push the input value
46
47               results.push(Integer.parseInt(input));
48            }
49            System.out.println(results);
50         }
51      }
52   }
```

Evaluating Algebraic Expressions with Two Stacks

- Using two stacks, you can evaluate expressions in standard algebraic notation.
  - One stack for numbers, one for operators

© Jorge Delgado/iStockphoto.

- Evaluating the top: 3 + 4

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

Evaluating Algebraic Expressions with Two Stacks

- Evaluate 3 x 4 + 5
  - Push until you get to the +

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

  - x (top of operator stack) has higher precedence than + , so evaluate the top

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

Evaluating Algebraic Expressions with Two Stacks

- Evaluate 3 + 4 × 5
  - Add x to the operator stack so we can get the next number

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

Evaluating Algebraic Expressions with Two Stacks

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

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

Evaluating Algebraic Expressions with Two Stacks

- Evaluating parentheses: 3 × (4 + 5)
  - Push (on the stack
  - Keep pushing until we reach the)
  - Evaluate until we find the matching (

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × (4 + 5)	Comments
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.
7	9 3	(×	No more input	Pop (.
8	9 3	×		Evaluate top again.
9	27			That is the result.

Evaluating Algebraic Expressions with Two Stacks

- The 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.
```

At the end, the value on the number stack the the value of the expression

Evaluating Algebraic Expressions with Two Stacks

- Helper method to evaluate the top:
  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

- Use a stack to remember choices you haven't yet made so that you can backtrack to them.
- Escaping a maze
  - You want to escape from a maze
  - You come to an intersection. What should you do?
  - Explore one of the paths
  - But remember the other paths.
  - If your chosen path doesn't work, you can
    - go back and try one of the other choices
- Use a stack to remember the paths that still need to be tried.
- The process of returning to a choice point and trying another choice is called **backtracking**.

Backtracking – Maze Example

- Start, at position (3, 4).
- There are four possible paths. We push them all on a stack .
- We pop off the topmost one, traveling north from (3, 4).
- Following this path leads to position (1, 4).
  - We now push two choices on the stack, going west or east .
  - Both of them lead to dead ends .
- Now we pop off the path from (3,4) going east.
  - That too is a dead end .
- Next is the path from (3, 4) going south.
- Comes to an intersection at (5, 4).
  - Both choices are pushed on the stack .
  - They both lead to dead ends .
- Finally, the path from (3, 4) going west leads to an exit .

Backtracking – Maze Example

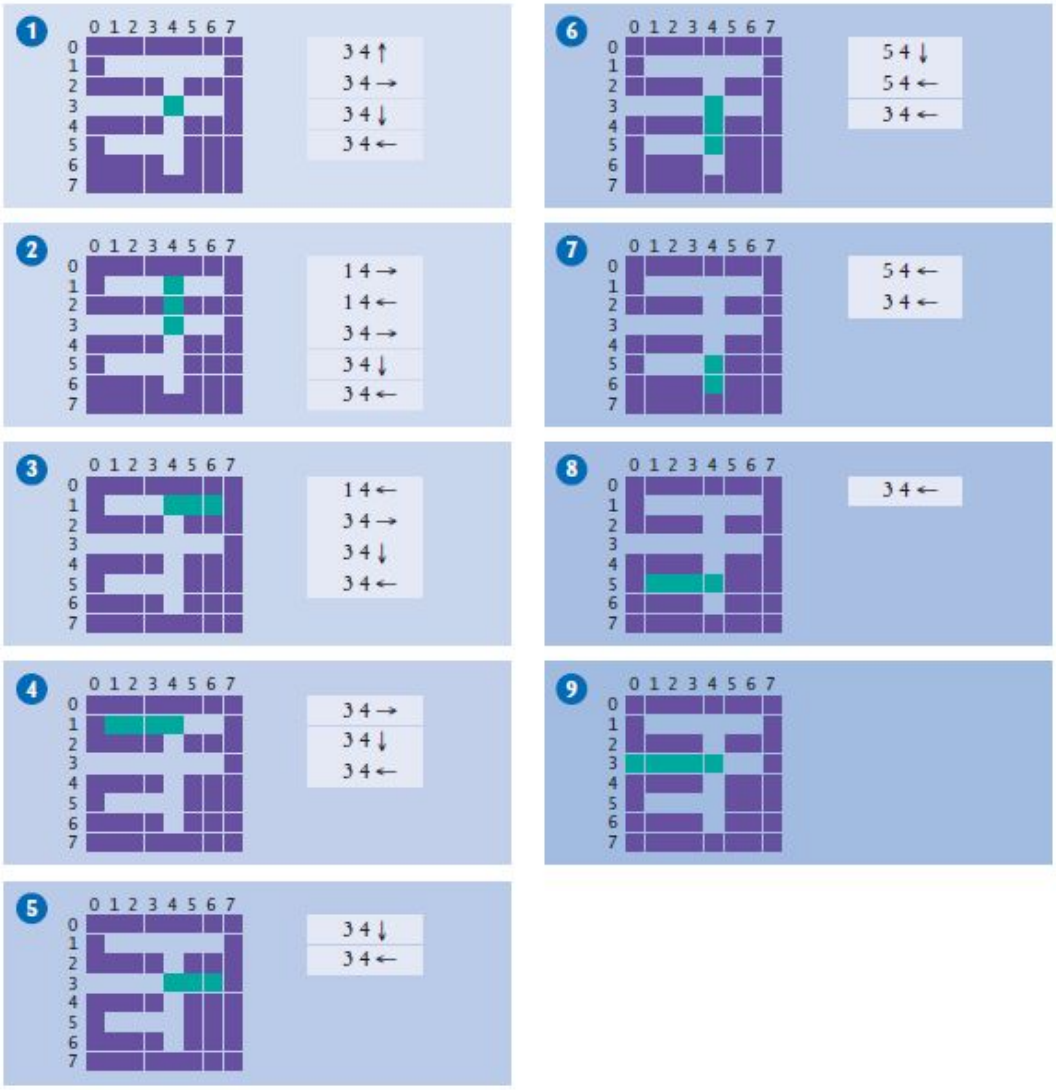


Figure 11 Backtracking Through a Maze

Backtracking – Maze Example

- Algorithm:

  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.

- This works if there are no cycles in the maze.
  - You never circle back to a previously visited intersection
- You could use a queue instead of a stack.

Self Check 15.26

What is the value of the reverse Polish notation expression 2 3 4 + 5 × ×?

Answer: 70

Self Check 15.27

Why does the branch for the subtraction operator in the Calculator program not simply execute

```
results.push(results.pop() - results.pop());
```

Answer: It would then subtract the first argument from the second. Consider the input 5 3 –. The stack contains 5 and 3, with the 3 on the top. Then results.pop() - results.pop() computes 3 – 5.

Self Check 15.28

In the evaluation of the expression 3 – 4 + 5 with the algorithm of Section 15.6.3, which operator gets evaluated first?

Answer: The – gets executed first because + doesn't have a higher precedence.

Self Check 15.29

In the algorithm of Section 15.6.3, are the operators on the operator stack always in increasing precedence?

Answer: No, because there may be parentheses on the stack. The parentheses separate groups of operators, each of which is in increasing precedence.

Self Check 15.30

Consider the following simple maze. Assuming that we start at the marked point and push paths in the order West, South, East, North, in which order are the lettered points visited, using the algorithm of Section 15.6.4?

Answer: A B E F G D C K J N