Java Collections Framework

Ameya Joshi
ameya.joshi_official@outlook.com
+91 9850676160

1

What is Collections Framework

The Java *collections framework standardizes the way in which groups of objects are* handled by your programs.

It is a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a *collection*.

The framework provides a convenient API to many of the abstract data types familiar from computer science data structure curriculum: maps, sets, lists, trees,arrays, hashtables, and other collections.

Because of their object-oriented design, the Java classes in the Collections Framework encapsulate both the data structures and the algorithms associated with these abstractions.

2

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

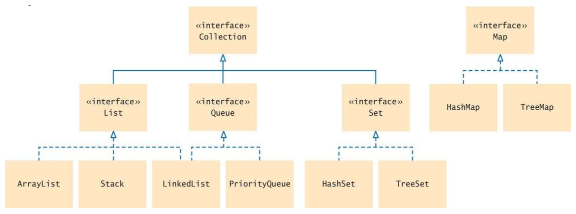


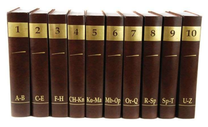
Figure 1 Interfaces and Classes in the Java Collections Framework

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

3

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`

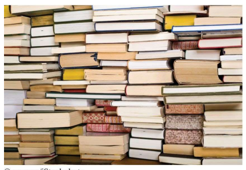


© Filip Fuxa/iStockphoto.

4

An Overview of the Collections Framework

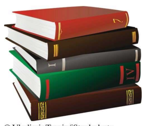
- `Set` interface
- A set is an **unordered** collection of **unique** elements.
- Arranges its elements so that finding, adding, and removing elements is more efficient.
- Two mechanisms to do this
  - hash tables
  - binary search trees


© parema/iStockphoto.

5

An Overview of the Collections Framework

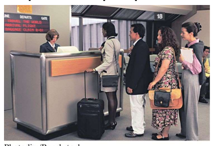
- Stack
  - Remembers the order of elements
  - But you can only add and remove at the top


© Vladimir Trenin/iStockphoto.

6

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



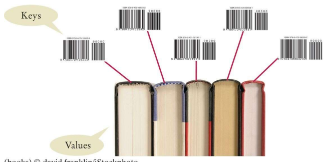
Photodisc/Punchstock.

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

7

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.



(books) © david franklin/iStockphoto.

8

An Overview of the Collections Framework

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

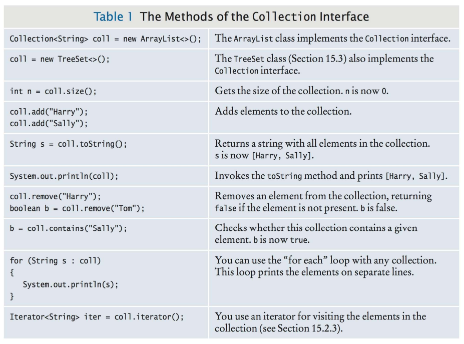


Table 1 The Methods of the `Collection` Interface

`Collection<String> coll = new ArrayList<>();`	The `ArrayList` class implements the `Collection` interface.
`coll = new TreeSet<>();`	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).

9

Self Check

A gradebook application 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.

10

Self Check

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 roll No.. By storing them in a set, adding, removing, and finding students can be efficient.

11

Self Check

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.

12

Self Check

As you can see from Figure 1, 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.

13

The Diamond Syntax

- Convenient syntax enhancement for array lists and other generic classes.
    -
- You can write:

```
ArrayList<String> names = new ArrayList<>();
```

  instead of:

```
ArrayList<String> names = new ArrayList<String>();
```

- This shortcut is called the "diamond syntax" because the empty brackets <> look like a diamond shape.
- This chapter, and the following chapters, will use the diamond syntax for generic classes.

14

Linked Lists

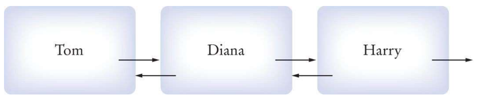
- A data structure used for collecting a sequence of objects:
  - Allows efficient addition and removal of elements in the middle of the sequence.
- A linked list consists of a number of nodes;
  - Each node has a reference to the next node.
- A node is an object that stores an element and references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

© andrea laurita/iStockphoto.

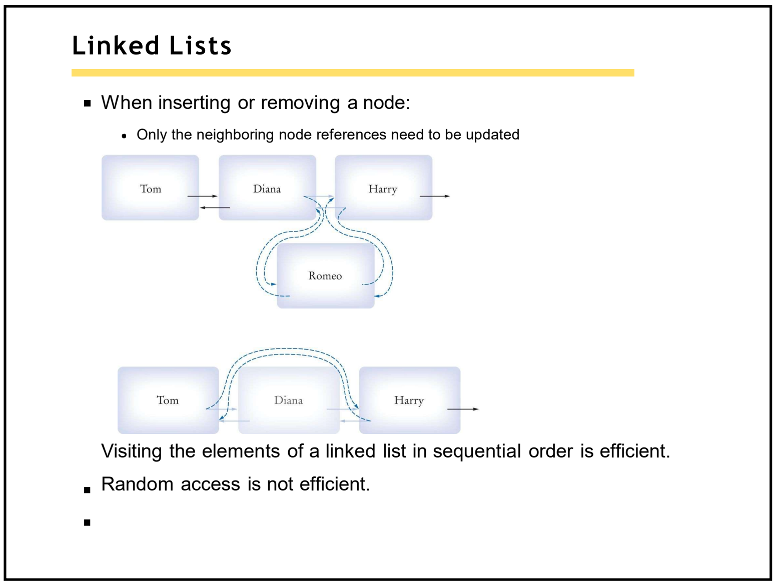
15

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.

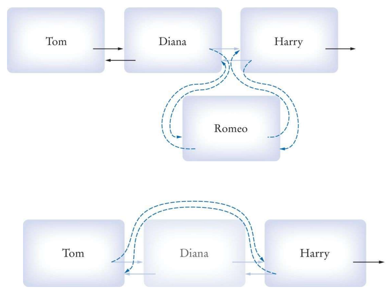


16



Linked Lists

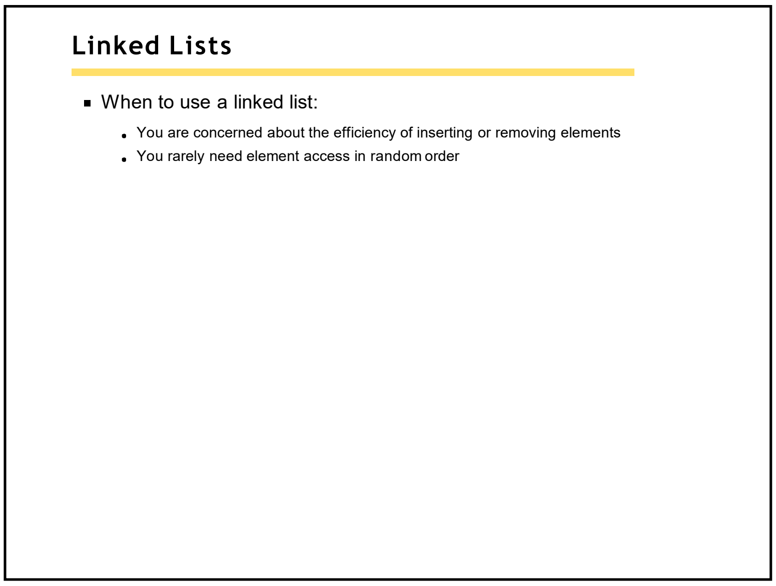
- When inserting or removing a node:
  - Only the neighboring node references need to be updated



Visiting the elements of a linked list in sequential order is efficient.

- Random access is not efficient.
-

17



Linked Lists

- 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

18

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.
- Some additional `LinkedList` methods:

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<>();`	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 684).

19

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.

20

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.

21

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
}
```

22

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`

23

A List Iterator

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

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

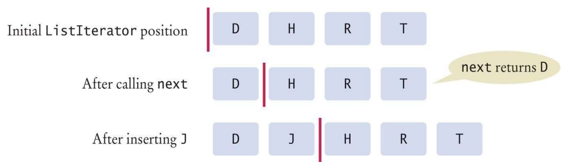


Figure 9 A Conceptual View of the List Iterator

24

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`

25

List Iterator

- `ListIterator` interface extends `Iterator` interface.
- Methods of the `Iterator` and `ListIterator` Interfaces

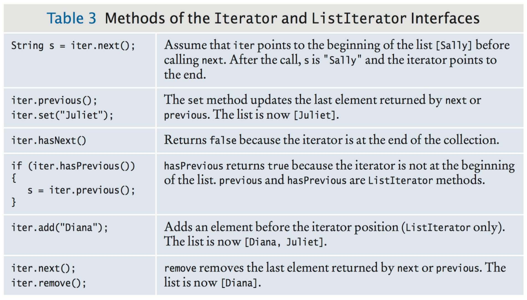
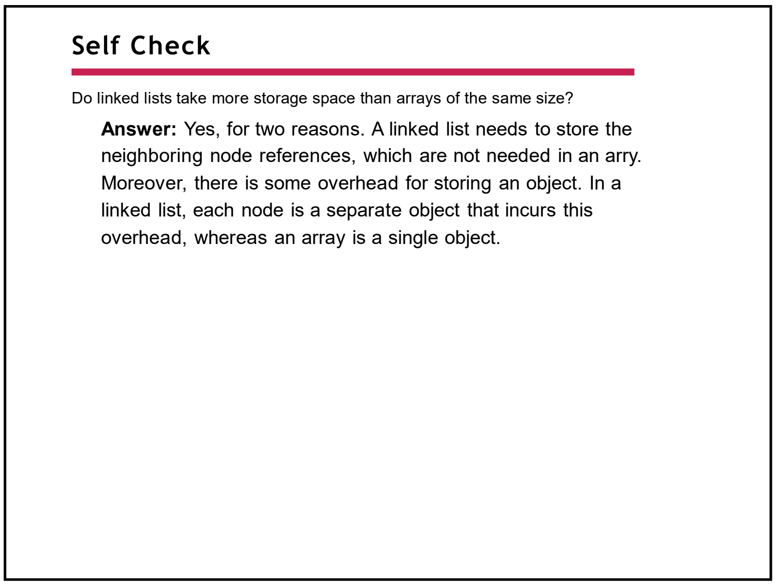


Table 3 Methods of the Iterator and ListIterator Interfaces

`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]`.

26

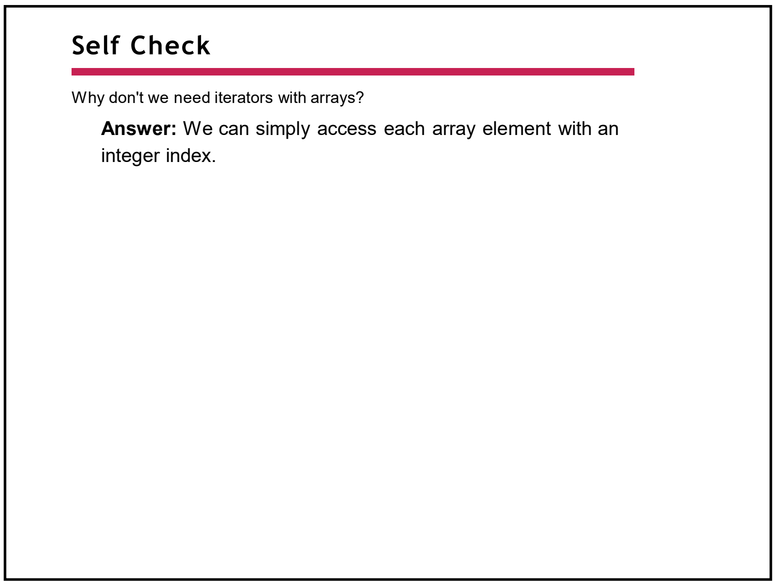


Self Check

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 arry. 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.

27

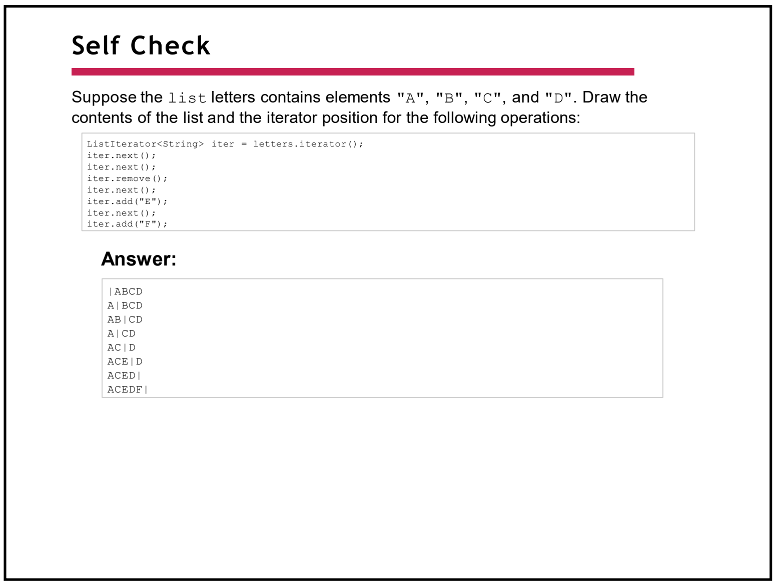


Self Check

Why don't we need iterators with arrays?

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

28



Self Check

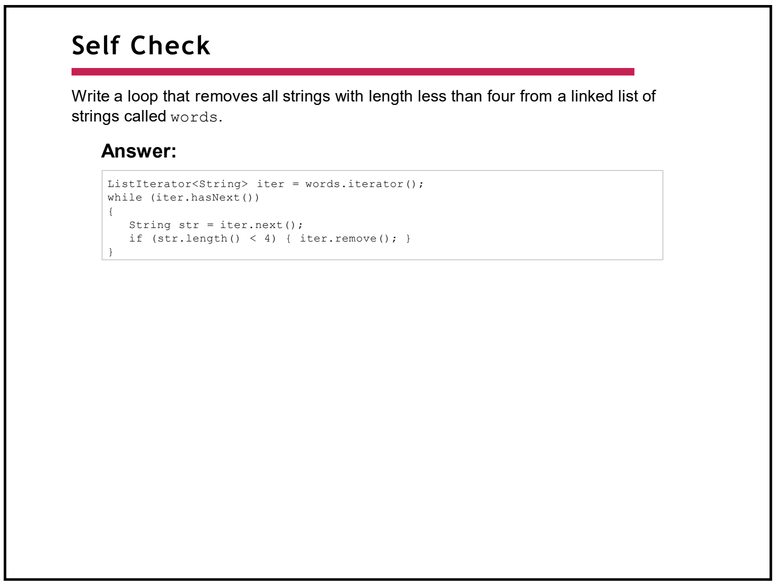
Suppose the `list` letters contains elements `"A"`, `"B"`, `"C"`, and `"D"`. Draw the contents of the list and the iterator position for the following operations:

```
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|
```

29



Self Check

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(); }
}
```

30

Self Check

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
   }
}
```

31

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.

32

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.

33

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 that is 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>>`
- 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.

© Alfredo Ragazzoni/iStockphoto.

34

Sets

- In a `TreeSet`
  - Elements are kept in sorted order

© Volkan Ersoy/iStockphoto.

- 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`.

35

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.

36

Sets

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

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

- After constructing the collection object:

  The implementation no longer matters

  Only the interface is important

37

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

38

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
}
```

- 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.

39

Working with Sets

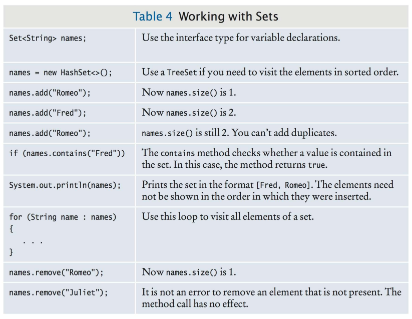


Table 4 Working with Sets

`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<>();`	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.

40

Self Check

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.

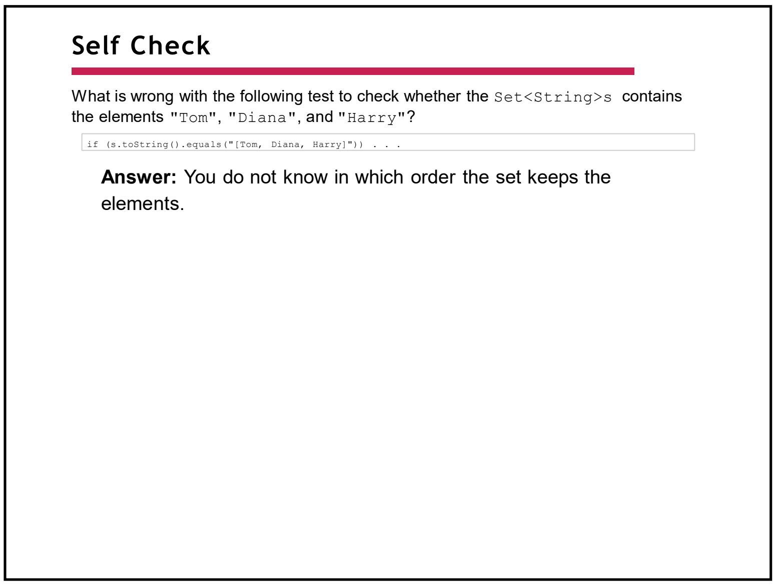
41

Self Check

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.

42



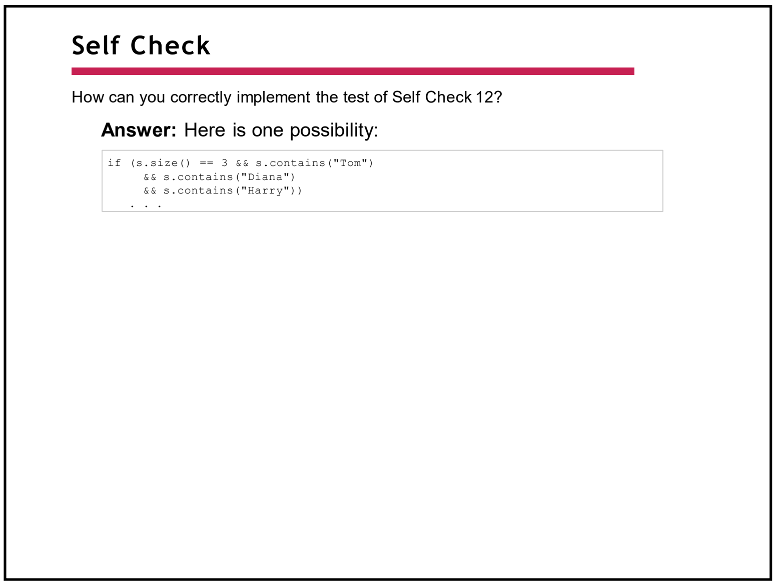
Self Check

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.

43



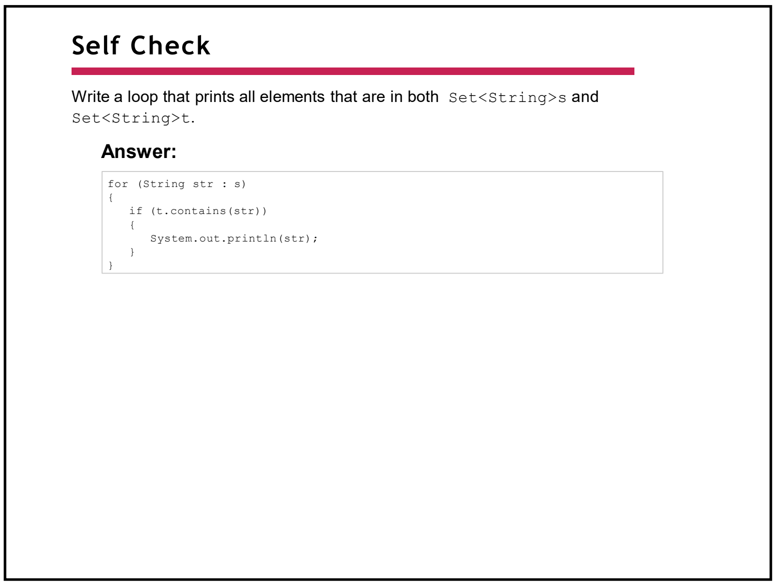
Self Check

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"))
   . . .
```

44



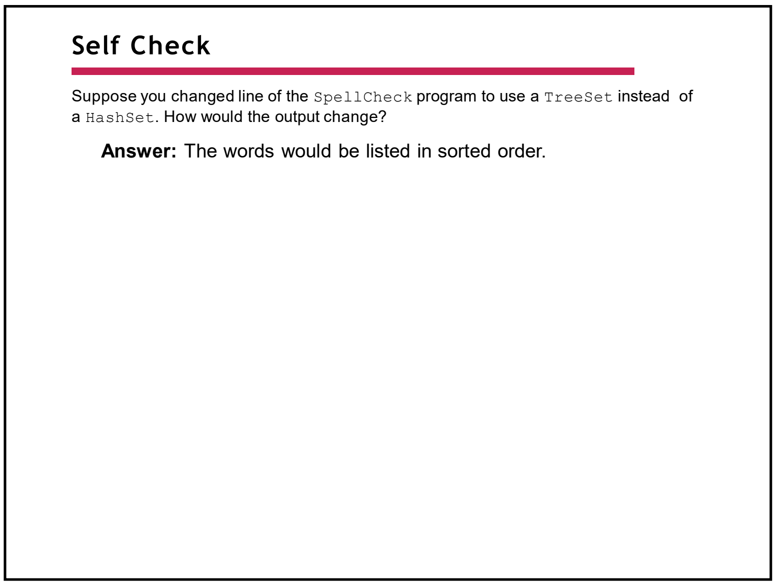
Self Check

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);
   }
}
```

45



Self Check

Suppose you changed line 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.

46

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.

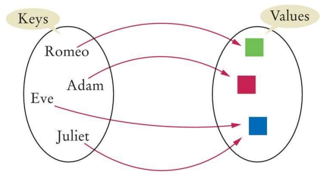


Figure 10 A Map

- 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<>();
```

47

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 julietsFavoriteColor = favoriteColors.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");
```

48

Working with Maps

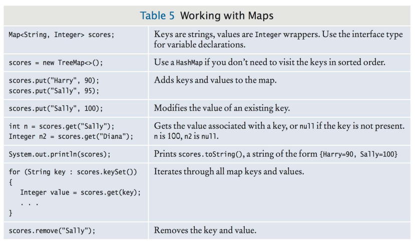


Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	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.

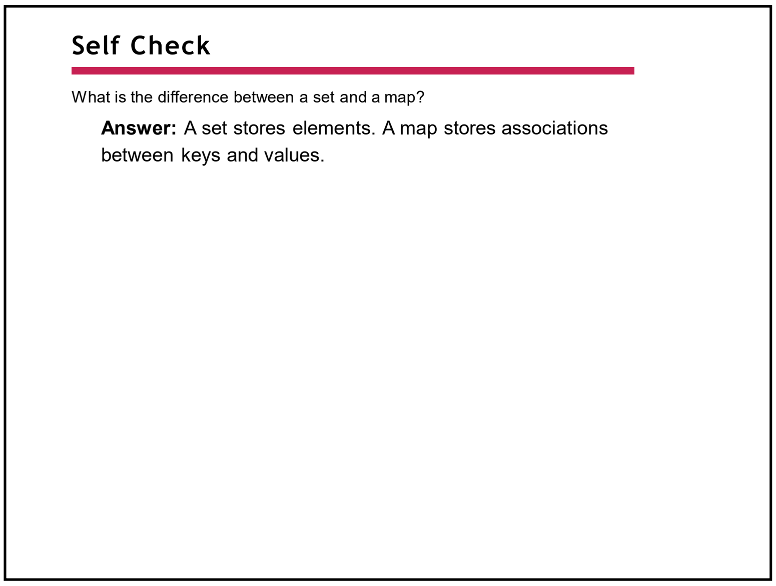
49

Map

- 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);
}
```

50

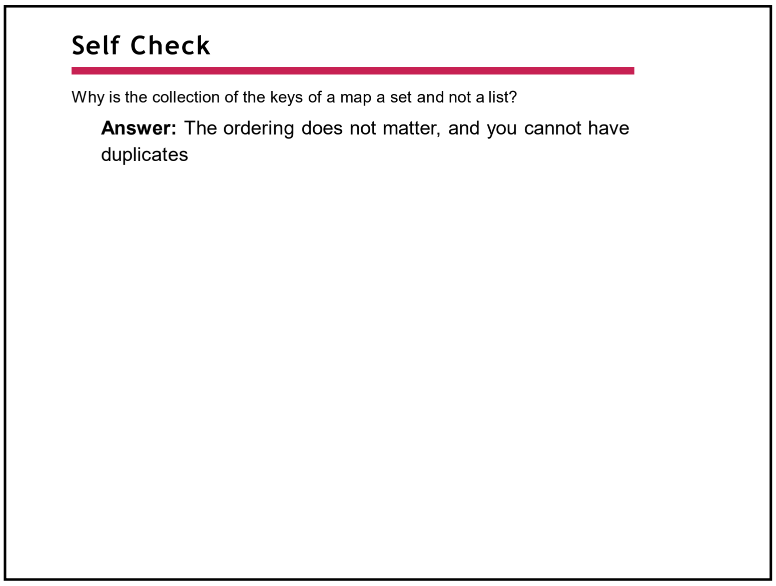


Self Check

What is the difference between a set and a map?

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

51



Self Check

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

52

Self Check

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

Answer: Because it might have duplicates.

53

Self Check

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

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

54

Self Check

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"`].

55

Java 8 Note

- Updating Map Entries used to be tedious

```
Integer count = frequencies.get(word); // Get the old frequency count
// If there was none, put 1; otherwise, increment the count
if (count == null) { count = 1; }
else { count = count + 1; }
frequencies.put(word, count);
```

- New `merge` method in `Map` interface
- Specify:

  Key

  Value to be used if key not yet present

  Function to compute the updated value if key is present

- Replace code above with single line using lambda expression:

```
frequencies.merge(word, 1, (oldValue, value) -> oldValue + value);
```

56

Choosing a Collection

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

57

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.
- 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`".

58

Hash Functions

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

59

Hash Functions

- Override `hashCode` methods in your own classes by combining the hash codes for the instance variables.
- A hash function for a `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;
   }
}
```

- Easier to use `Objects.hash()` method
  - Takes hashcode of all arguments and multiplies them:

```
public int hashCode()
{
   return Objects.hash(name, area);
}
```

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

60

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`.
- Example

```
Stack<String> s = new Stack<>();
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.

© John Madden/iStockphoto.

61

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.

© budgetstockphoto/iStockphoto.

- 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.

62

Stack in the Java Library

- `Stack` class provides `push`, `pop` and `peek` methods.

Table 7 Working with Stacks

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

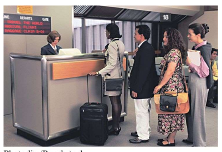
`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; s is now `[1, 2, 3]`. (Following the `toString` method of the `Stack` class, we show the top of the stack at the end.)

`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.

63

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.

64

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

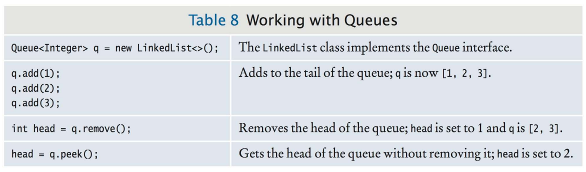


Table 8 Working with Queues

`Queue<Integer> q = new LinkedList<>();`	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.

65

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.
- 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.

© paul kline/iStockphoto.

66

Priority Queues

- Example: objects of a class `WorkOrder` into a priority queue.

```
PriorityQueue<WorkOrder> q = new PriorityQueue<>();
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.

Table 9 Working with Priority Queues

`PriorityQueue<Integer> q = new PriorityQueue<>();`	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.

67

Self Check

Why would you want to declare a variable as

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

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.

68

Self Check

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.

69

Self Check

What does this code print?

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

Answer: A B C

70

Self Check

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.

71