Chapter 15 -The Java Collections Framework

Chapter 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

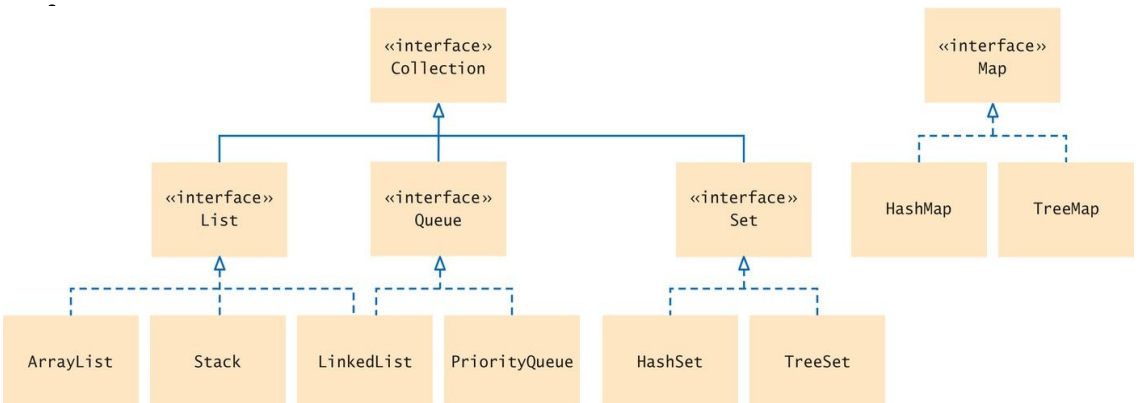


Figure 1 Interfaces and Classes in the Java Collections Framework

- The `Collection` interface is at the root
  - All `Collection` class 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`

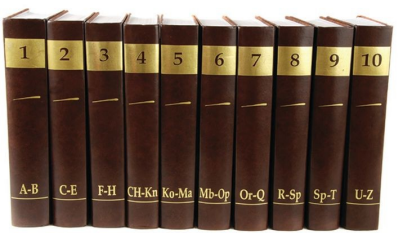


Figure 2 A List of Books

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

An Overview of the Collections Framework

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

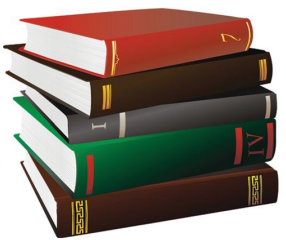


Figure 4 A Stack of Books

An Overview of the Collections Framework

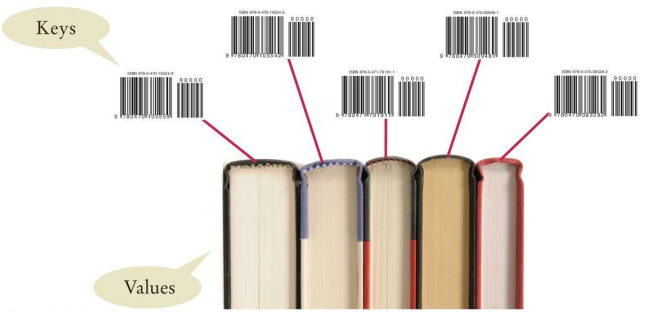
- Queue
  - Add items to one end (the tail) and remove them from the other end (the head)
- A queue of people

Photodisc/Punchstock.

- 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.



(books) © david franklin/iStockphoto.

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<>();`	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).

Self Check 15.1

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.

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 15.4

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.

The Diamond Syntax

- Convenient syntax enhancement for array lists and other generic classes.
  - Mentioned in Chapter 7 Special Topic 7.5.
- 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.

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.

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

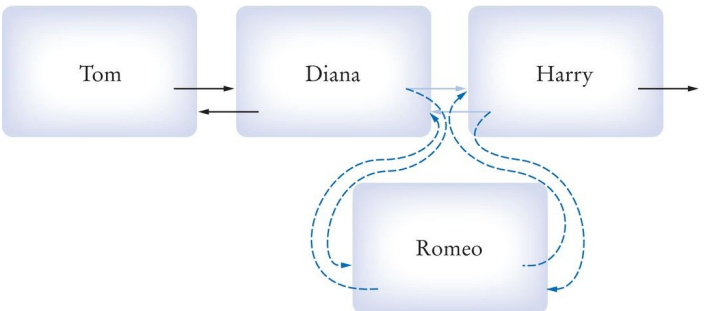


Figure 7 Inserting a Node into a Linked List

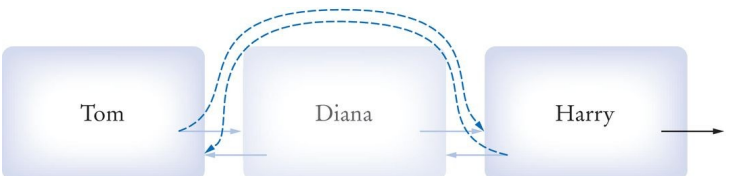


Figure 8 Removing a Node From A Linked List

- Visiting the elements of a linked list in sequential order is efficient.
- Random access is not efficient.

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

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).

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`

A List Iterator

- The `add` method adds an object after the iterator.

  Then moves the iterator position past the new element.

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

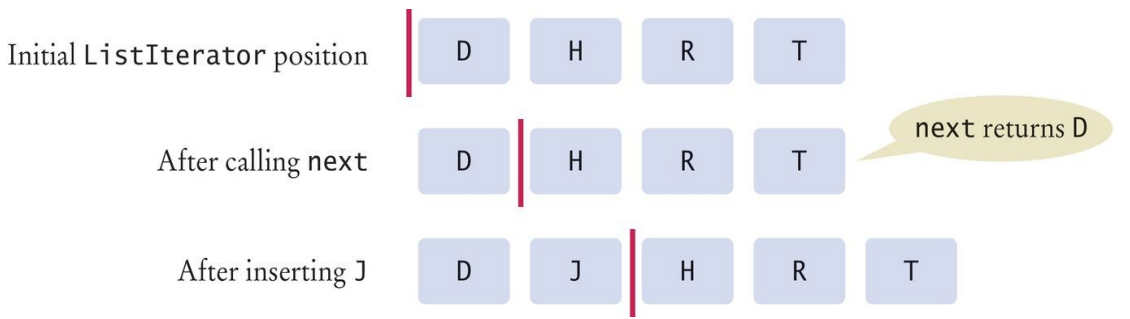


Figure 9 A Conceptual View of the List Iterator

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

```
LinkedList<String> staff = new LinkedList<>();
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
// 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]");
```