

Java Collections API

Lecturer: Dr Anwarul Patwary

Outline

We discuss:

- ▶ What is a collection?
- ▶ What is a collection *framework*, and how is this realized in Java?
- ▶ What are some of the core interfaces and implementations in the Java Collections framework?

Collections

- ▶ A *collection* (sometimes called a container) is an object that groups multiple elements into a single unit.
- ▶ We might use collection objects to store ...
 - ▶ cards in a poker hand
 - ▶ emails in a mail folder
 - ▶ a *mapping* of names to phone numbers, for an address book.
- ▶ If you have used Java – or almost any other programming language – you would already have some familiarity with collections.

Types of collection

- ▶ We have now seen a number of sorts of collection:
 - ▶ arrays
 - ▶ queues
 - ▶ stacks
 - ▶ lists
 - ▶ trees
 - ▶ sets
 - ▶ maps
- ▶ In future lectures, we will examine *graphs*

Types of collection

Why do we have so many types of collection?

- ▶ Different tasks have different requirements for the complexity of operations

Types of collection

Why do we have so many types of collection?

- ▶ Different tasks have different requirements for the complexity of operations
 - ▶ e.g. sometimes it is important we be able to access any individual item in a collection in constant time – *arrays* and structures built on them allow for this.

Types of collection

Why do we have so many types of collection?

- ▶ Different tasks have different requirements for the complexity of operations
 - ▶ e.g. sometimes it is important we be able to access any individual item in a collection in constant time – *arrays* and structures built on them allow for this.
- ▶ Different sorts of collection make trade-offs between space and time efficiency

Types of collection

Why do we have so many types of collection?

- ▶ Different tasks have different requirements for the complexity of operations
 - ▶ e.g. sometimes it is important we be able to access any individual item in a collection in constant time – *arrays* and structures built on them allow for this.
- ▶ Different sorts of collection make trade-offs between space and time efficiency
- ▶ It is better to use a collection that offers *just* the operations we needed, than to try to use one that is ill-suited.

Operations on collections

Typical operations on a collection are to:

- ▶ *add* something to the collection
- ▶ *find* whether (and where) an item is in a collection
- ▶ *retrieve* an item
- ▶ *remove* or *replace* an item
- ▶ *clone* the whole collection (make a copy)

Using Java Collections

In Java, built-in (API) classes can be accessed in several ways:

1. by providing their “full name”

```
java.util.LinkedList<String> b =  
    new java.util.LinkedList<>();
```

Here `LinkedList` is a class in the API package `java.util`.

2. by *importing* the class

```
import java.util.LinkedList;  
// ...  
LinkedList<String> b = new LinkedList<>();
```

3. You can also import all classes in a package:

```
import java.util.*;
```

Java collections packages

Most general data structures in the Java API are in the `util` package. There are:

1. **Collections:**

`LinkedList<E>`, `ArrayList<E>`, `PriorityQueue<E>`, `Set<E>`,
`Stack<E>`, `TreeSet<E>`

2. **Maps:**

`SortedMap<K, V>`, `TreeMap<K,V>`, `HashMap<K, V>`

3. and others:

`Iterator<E>`, `BitSet`

These allow you to create most of the data structures you will ever need.

However, it is important to be able to compare the performance and understand the limitations of each.

A collections framework

- ▶ A *collections framework* is a unified architecture for representing and manipulating collections.
- ▶ Collections frameworks contain the following:
 - ▶ **Interfaces**
 - ▶ **Implementations**
 - ▶ **Algorithms**

Collection interfaces

Interfaces are **abstract data types** that represent collections.

- ▶ This means collections can be manipulated independently of the details of their representation.
- ▶ For instance, if something obeys the rules for being a `List` or a `Stack`, say – then we can treat it as one in code, regardless of how it is implemented.
- ▶ (We will note that there are other possible ways of implementing a `List` interface besides using the singly-linked list data structure we have seen. `List` is more general than that.)

Collection interfaces

- ▶ In object-oriented (OO) languages, interfaces generally form a *hierarchy*
- ▶ For example - we have seen that a Stack ADT has push and pop methods.
- ▶ We could generalize this, and imagine an ADT called “Collection”, say, that has generic add and remove methods.
- ▶ In an OO language like Java, we would represent this by having Stack inherit from Collection.
- ▶ In Java, an interface that inherits from an interface is said to *extend* the original interface.

Collection implementations

The **implementations** are the concrete implementations of the collection interfaces.

- ▶ In essence, they are reusable data structures.

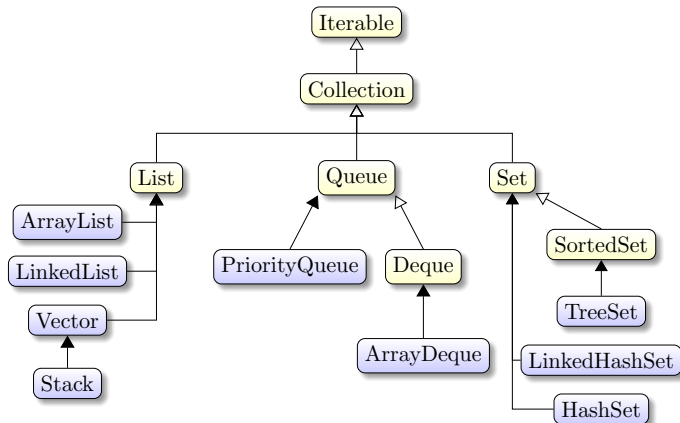
Collection algorithms

The **algorithms** are methods that perform useful computations on objects that implement collection interfaces.

- ▶ For example, we have seen sorting and searching algorithms – these are common sorts of algorithms we would wish to use with collections.
- ▶ Note that the algorithms can be used with any collection that implements the appropriate interface.
 - ▶ They are said to be *polymorphic* –
 - ▶ that is, the same method can be used on many different implementations of the appropriate collection interface.
- ▶ In essence, algorithms are reusable functionality.

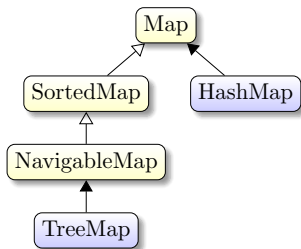
Java collections framework

- ▶ The Java Collections Framework forms a hierarchy, shown below, which we will discuss.
- ▶ The Java [online documentation](#) discusses the Collections Framework in more detail, and provides links to individual classes and interfaces.



Maps

- There is a *separate* hierarchy in the collections framework, containing *maps*; these do not inherit from `Collection`.



Thus the hierarchy consists of two distinct trees — a `Map` is not a true `Collection`.

Use of Java generics

- ▶ Note that all the core collection interfaces are generic.
- ▶ For example, this is the declaration of the Collection interface.

```
public interface Collection<E> { // ...
```

- ▶ The <E> syntax tells us that the interface is generic.
- ▶ When you declare a Collection instance you can *and should* specify the type of object contained in the collection:

```
Collection<String> myCollection;
```

- ▶ As we have seen previously – specifying the type allows the compiler to verify at compile-time that the type of object you put into the collection is correct, thus reducing errors at runtime.

Core collection interfaces

We will consider some of the core collection interfaces.

Collection

- ▶ the root of the collection hierarchy.
- ▶ A collection represents a group of objects known as its *elements*.
- ▶ The Collection interface is the “least common denominator” that all collections implement
- ▶ It is used to pass collections around and to manipulate them when maximum generality is desired.
- ▶ Some types of collections allow duplicate elements, and others do not.
- ▶ Some are ordered and others are unordered.

Collection api

The methods for Collection include:

- ▶ `int size()`
- ▶ `boolean isEmpty()`
- ▶ `boolean contains(Object element)`
- ▶ `boolean add(E element)`
- ▶ `boolean remove(Object element)`
- ▶ `Iterator<E> iterator()`

Core collection interfaces

Set

- ▶ A collection that cannot contain duplicate elements.
- ▶ This interface models the mathematical set abstraction
- ▶ It is used to represent sets such as the cards comprising a poker hand, say

Core collection interfaces

List

- ▶ An *ordered* collection (sometimes called a *sequence*).
- ▶ Lists can contain duplicate elements.
- ▶ The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

Core collection interfaces

Queue

- ▶ A collection used to hold multiple elements prior to processing.
- ▶ Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.
- ▶ We have seen that queues typically order elements in a FIFO (first-in, first-out) manner.
- ▶ An exception are priority queues, which order elements according to a supplied comparator or the elements' natural ordering.

Core collection interfaces

Deque

- ▶ A collection used to hold multiple elements prior to processing.
- ▶ Besides basic Collection operations, aDeque provides additional insertion, extraction, and inspection operations.
- ▶ Deques can be used both as FIFO (first-in, first-out) **and** LIFO (last-in, first-out).
- ▶ In a deque elements can be inserted, retrieved and removed at both ends.

Core collection interfaces

Map

- ▶ An object that maps *keys* to *values*.
- ▶ A Map cannot contain duplicate keys; each key can map to at most one value.

Core collection implementations

We have seen some of the core interfaces; what about the implementations?

- ▶ The Java Collections framework comes with many implementation classes for the interfaces.
- ▶ The most commonly used implementations are `ArrayList`, `HashMap` and `HashSet`.

ArrayList Class

- ▶ Resizable-array implementation of the List interface.
- ▶ Implements all optional list operations, and permits all elements, including null.
- ▶ In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.
- ▶ Because this is implemented using an array ...
the size, isEmpty, get, set, iterator, and listIterator operations run in constant time.
- ▶ The add operation runs in amortized constant time
 - ▶ that is, adding n elements requires $O(n)$ time.
- ▶ All of the other operations run in linear time (roughly speaking).
- ▶ The constant factor is low compared to that for the LinkedList implementation.

HashMap Class

- ▶ Hash table based implementation of the Map interface.
- ▶ This implementation provides all of the optional map operations, and permits null values and the null key.
- ▶ This class makes no guarantees for the order of the map.
- ▶ This implementation provides constant-time performance for the basic operations (get and put).

HashSet Class

- ▶ This is the basic implementation the Set interface that is backed by a HashMap.
- ▶ It makes no guarantees for iteration order of the set and permits the `null` element.
- ▶ This class offers constant time performance for basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets.

LinkedList Class

- ▶ Doubly-linked list implementation of the List and Deque interfaces.
- ▶ Implements all optional list operations, and permits all elements (including null).

PriorityQueue Class

- ▶ PriorityQueue class was introduced in Java 1.5.
- ▶ PriorityQueue is an unbounded queue based on a priority heap and the elements of the priority queue are ordered by default in natural order or we can provide a **Comparator** for ordering at the time of instantiation of queue.
- ▶ The head of the Java priority queue is the **least** element based on the natural ordering or comparator based ordering, if there are multiple objects with same ordering, then it can poll any one of them randomly. When we poll the queue, it returns the head object from the queue.
- ▶ PriorityQueue implementation provides $O(\log n)$ time for enqueueing and dequeuing method.

Further reading

Further reading

For further reading, see the Java Collections Framework Tutorial (<http://www.journaldev.com/1260/java-collections-framework-tutorial>)