Collections

Java Collections Framework, generics and collections, List operations, iterators, wildcards, restrictions

What is a collection?

A **collection** is an object that holds a group of other objects (data). Objects in a collection are called **elements**.

The java.util library has interfaces and classes for implementing various kinds of collections, such as:

- Collection: interface for sets and lists.
- Set: interface for a collection that cannot contain duplicates.
  Implementing classes: HashSet, TreeSet
- List: interface for a sequence of elements.
  Implementing classes: ArrayList, LinkedList, Vector

The interfaces indicate which methods the implementing classes are *guaranteed* to provide.

There are also Maps, which are collections of elements that consist of key-value pairs.

Java collections framework

The collection interfaces together with classes in the Java API that implement them constitute the **Java collections framework**. The classes include:

- Set<E> implementations
  - HashSet<E>. The most commonly used set type because it is the most efficient implementation. ***Hashing*** computes a unique value for each element of the set, so any particular element is quick to look up even if the set is very large.
  - TreeSet<E>. Imposes an order on the elements.
  - LinkedHashSet<E>. A cross between HashSet<E> and TreeSet<E>.
- List<E> implementations
  - ArrayList<E>. List type that has indexes for element access.
  - LinkedList<E>. List elements are linked rather than indexed.

[continued next slide]

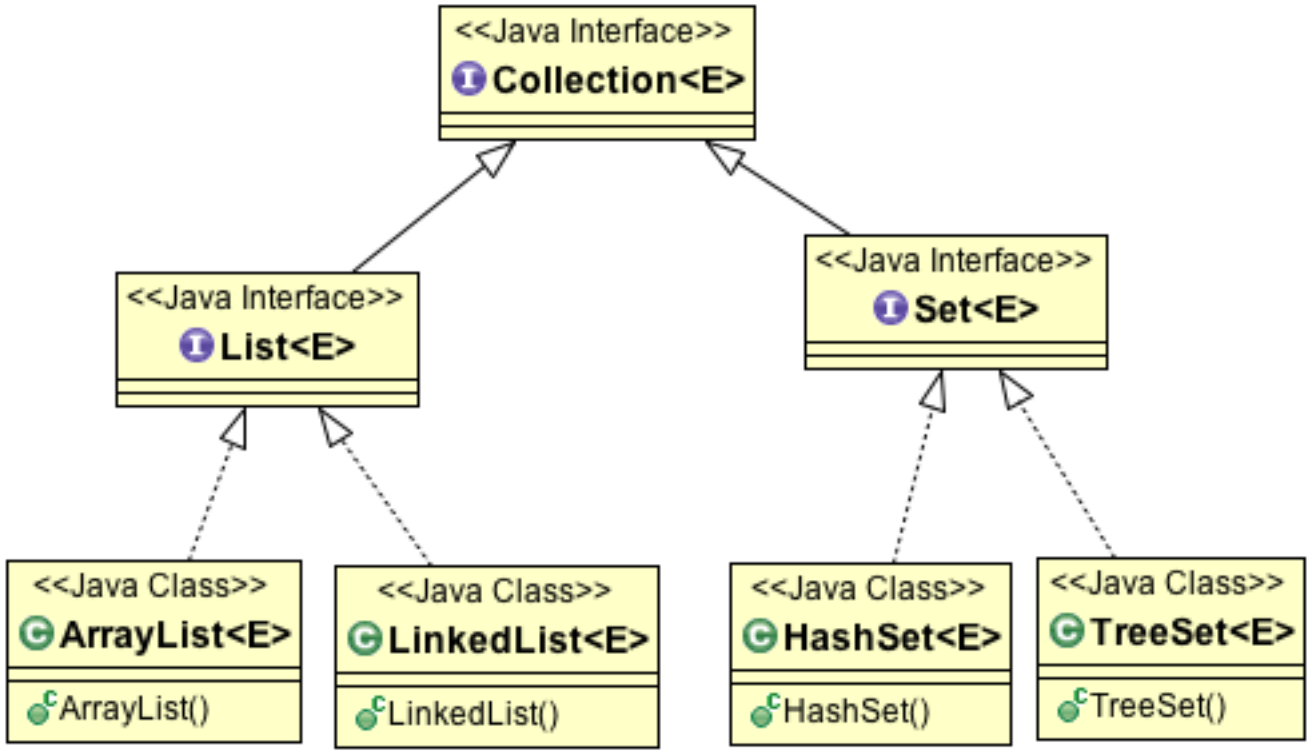
Java collections framework (cont.)

- Map<K, V> implementations. (There are 2 type parameters.)
  - HashMap<K,V>. The most commonly used. Each element (which is a key-value pair) has a hash to determine its location. Hashing makes this an efficient data structure.
  - TreeMap<K,V>. Helpful if the keys should be stored in a particular order.
  - LinkedHashMap<K,V>. Can change the order of storage of elements depending on when they were last accessed.
- Queue<E> implementations
  - LinkedList<E>. The most commonly used. For a standard queue, elements are removed from the front and added to the rear.
  - PriorityQueue<E>. The smallest element is always at the front. Removing the smallest element re-orders the rest of the queue to bring the smallest remaining element to the front.

Java Collections Framework

The **Java Collections Framework** provides reusable data structures.

A tiny part of the Java Collections Framework.



List, ArrayList, LinkedList

A **list** is a sequence of elements. As an abstract data type, a list should support several operations: create(), isEmpty(), size(), add(), remove(), and get().

`List` is a Java Collections Framework interface that declares list operations. Declaration:

```
interface List<E> extends Collection<E>
```

Two framework classes implement `List` :

- ArrayList implements a list in which the underlying data structure is an array.
  - ArrayList elements are contiguous.
  - ArrayLists are indexed.
  - ArrayLists have capacity.
  - Declaration: **`class ArrayList<E> implements List<E>`**
- LinkedList implements a list in which the underlying data structure is a linked list.
  - The list consists of “nodes.” Each node consists of a list element and a reference to the next node in the list.
  - Declaration: **`class LinkedList<E> implements List<E>`**

List operations

`List` declares many operations, including these important ones:

- `int size()` – number of elements in the list
- `boolean isEmpty()` – true if the list contains no elements
- `boolean contains(E item)` – true if `item` is a list element
- `int indexOf(Object item)` – the index of `item` in the list (0-based indexing)
- `void add(E item)` – adds `item` to the end of the list
- `void add(int index, E item)` – inserts `item` to the list at index. Subsequent elements shift to one index higher.
- `E remove(int index)` – removes and returns the element at `index`. Subsequent elements shift to one index lower.
- `void set(int index, E item)` Replaces the element at `index` with `item`
- `E get(int index)` – gets the element at the given index

Declaring collection classes

- You must specify the actual element type to correspond to the formal parameter type, <T> or <E>.
- The actual element type must be a class type – cannot be primitive (use wrapper classes if needed).
- The syntax for a parameterized type uses braces, < >, to enclose the actual type parameter:
  - ArrayList<HighwayVehicle>
  - LinkedList<Integer> numberList = new LinkedList<Integer>();
  - public void foo(LinkedList<String> a)
  - List<String> party = new ArrayList<String>().
- Note:
  - party can be assigned to an instance of a different class that implements List<String>.
  - party can be an actual parameter to a method that has a formal parameter of type List<String>.

For-each loops for collections

Iterating through a collection means to go through the collection, visiting one element at a time. Since collections aren't generally indexed, the easiest way to go through a collection is to use a for-each loop.

Format of a for-each loop:

for (element-type element-name : collection-name) {

visit element-name }

```
Collection<String> example = new HashSet<String>();
// Add some Strings to example
...
int twoCharacterStrings = 0;
for (String s: example) {
   if (s != null && s.length() == 2)
      twoCharacterStrings++;
}
```

Iterators to access elements

- For-each loops are preferred if you are not going to change collection elements.
- To traverse a collection with the intent of possible change, use an iterator.
- **Iterator** = an object that can traverse a collection. It has 3 traversal methods:
  - hasNext() – true if the entire collection hasn't been traversed
  - next() – gets the next element in the traversal, incrementing the iterator to point to the element beyond that.
  - remove() – removes the most recently visited element.
- Iterators are always initialized to start at "the beginning" of a collection.
- Collection <E> interface declares an iterator method: Iterator<E> iterator()

Iterator – example traversal

The following code shows how to use a iterator to traverse a collection.

```
Collection<String> example = new HashSet<String>();
// Add some Strings to example.
//...
int twoCharacterStrings = 0;
Iterator<String> iter = example.iterator();
while (iter.hasNext()) {
     if (iter.next().length() == 2)
        twoCharacterStrings += 1;
}
```

Iterator – example change

The following code shows an iterator can change a collection.

```
String[] items = {"ab", "cdef","gh", "ijk", "lmnop", "yz"};

Collection<String> myWords
          = new ArrayList<String>(Arrays.asList(items));
// .... do something with myWords

Iterator<String> iter = myWords.iterator();
while (iter.hasNext()) {
    if (iter.next().length() == 2)
         iter.remove();
}
```

Parameterized types for method arguments

This example finds the sum of a collection of integers.

```
public static int sum(Collection<Integer> c) {
    int sumOfElements = 0;
    for (Integer x: c) {
        sumOfElements += x;
    }
    return sumOfElements;
}
```

Wildcards

- For a formal collection argument that is independent of the type parameter, use a type **wildcard**.
- Wildcard notation is **<?>**.

This method prints all of the elements of its actual argument.

```
public void printOut(Collection<?> c) {
    for (Object x: c) {
        System.out.println(x);
    }
}
```

- printOut() can be called using any class that implements a parameterized Collection. This includes such types as ArrayList<Integer>, LinkedList<String>, and HashSet<Thermometer>.

Compile Error: arrays of generics

```
// Error
ArrayList<String>[] bad = new ArrayList<String>[100];
```

Error on that line: “Cannot create a generic array of ArrayList<String>.”
The error is because the compiler cannot enforce type safety.

Java Tutorial’s suggested solution is to use wildcards and cast.
But you need to be careful about generating a runtime error.

```
ArrayList<?>[] ok = new ArrayList<?>[100];
ok[0] = new ArrayList<String>();
((ArrayList<String>)ok[0]).add("John");
```