Java Collections Framework

Xuetao Wei

weixt@sustech.edu.cn

Objectives

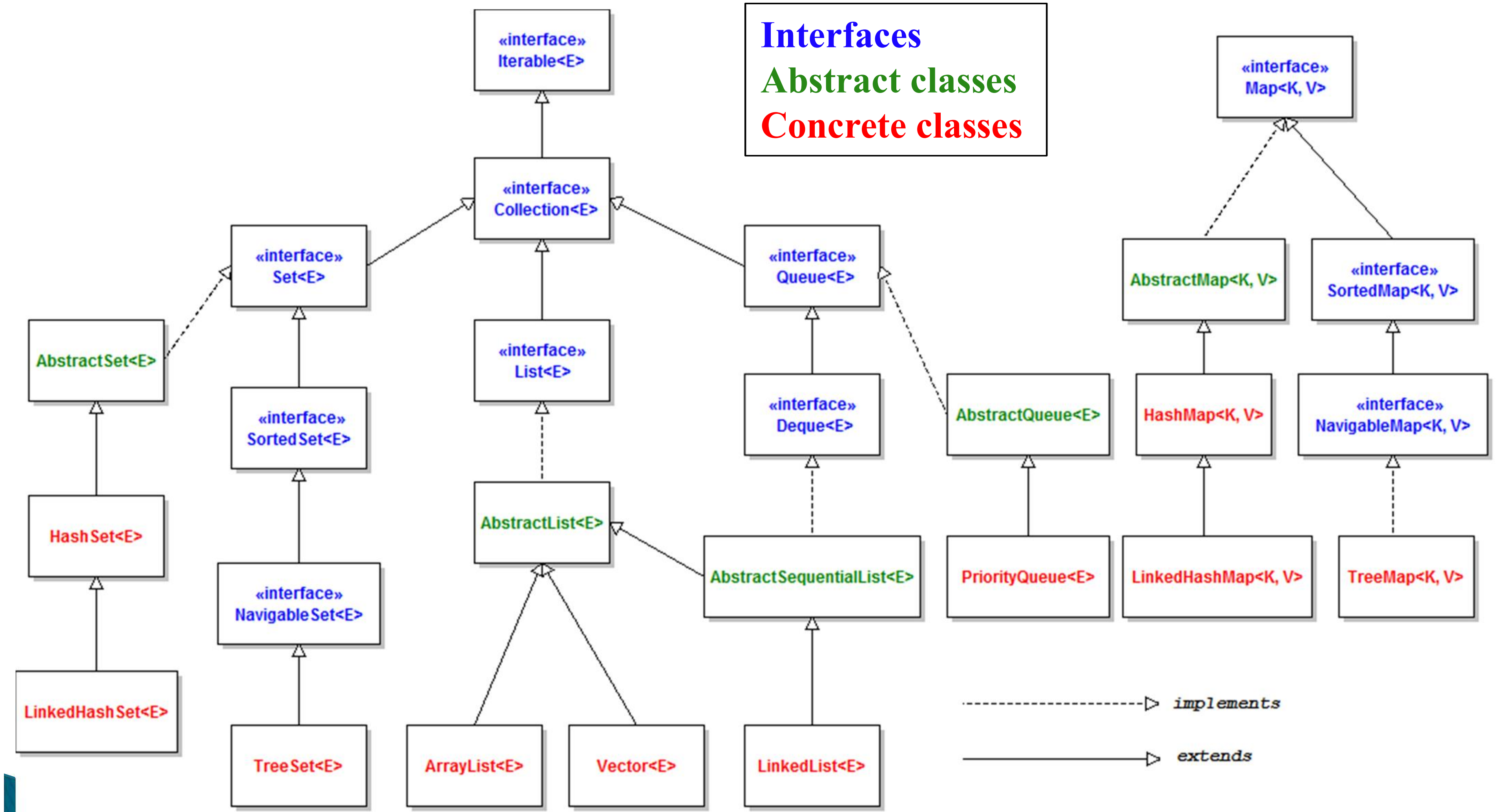
- Java collections framework
- Three common types of collections
- `ArrayList`
- `HashMap`

Java Collections Framework (JCF)

- JCF is a set of **classes** and **interfaces** that implement reusable **data structures** (or containers) to help **group and manage related objects**
- Similar to arrays, collections hold references to objects that can be managed as a group (**one object represents a group of objects**)
- Unlike arrays, collections do not need to be assigned a certain capacity when instantiated. **Their size can grow and shrink automatically** when objects are added or removed.
- Unlike arrays, **collections cannot hold primitive type elements** (e.g., `int`), they can only hold object references (arrays can do both).

https://en.wikipedia.org/wiki/Java_collections_framework

JCF Class Hierarchy



http://www.codejava.net/java-core/collections/java-map-collection-tutorial-and-examples

The Collection Interface

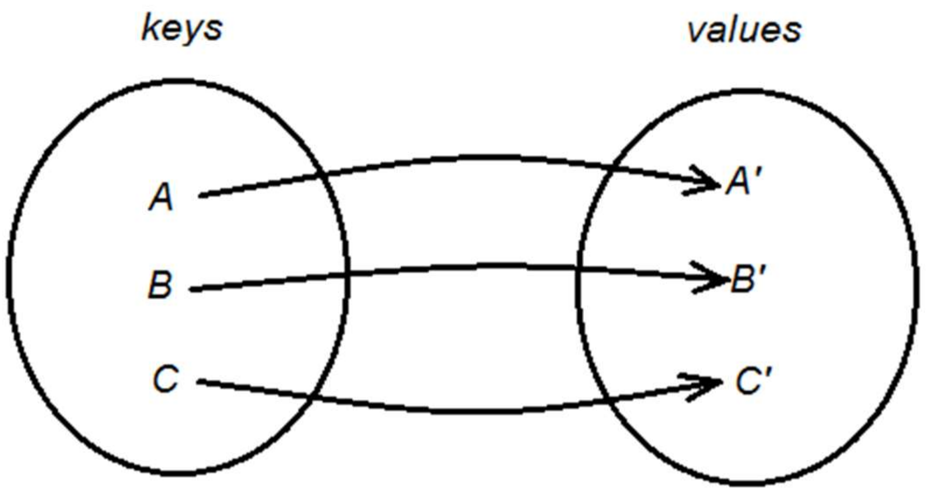
- `java.util.Collection` is the root interface in the *collection hierarchy*
- Methods declared (**not implemented**) in `Collection`:
  - `add, addAll` (adding elements)
  - remove, removeAll, removeIf, clear (removing elements)
  - `contains, containsAll` (checking the existence of elements)
  - `size` (returning the number of elements)
  - `toArray` (returning an array containing all elements in the collection)
  - …

List and Set

- `Collection` has two important offspring: `List` and `Set`
- **A list is an ordered collection** (also known as a **sequence**). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Lists typically allow duplicate elements.
- **A set is collection that contains no duplicate elements**. This interface models the mathematical set abstraction

The Map Interface

- A `Map` is an object that maps **keys** to **values**, or is a collection of **attribute-value pairs**.
  - A map of error codes and their descriptions (404 → Not found)
  - A map of zip codes and cities (518000 → Shenzhen)

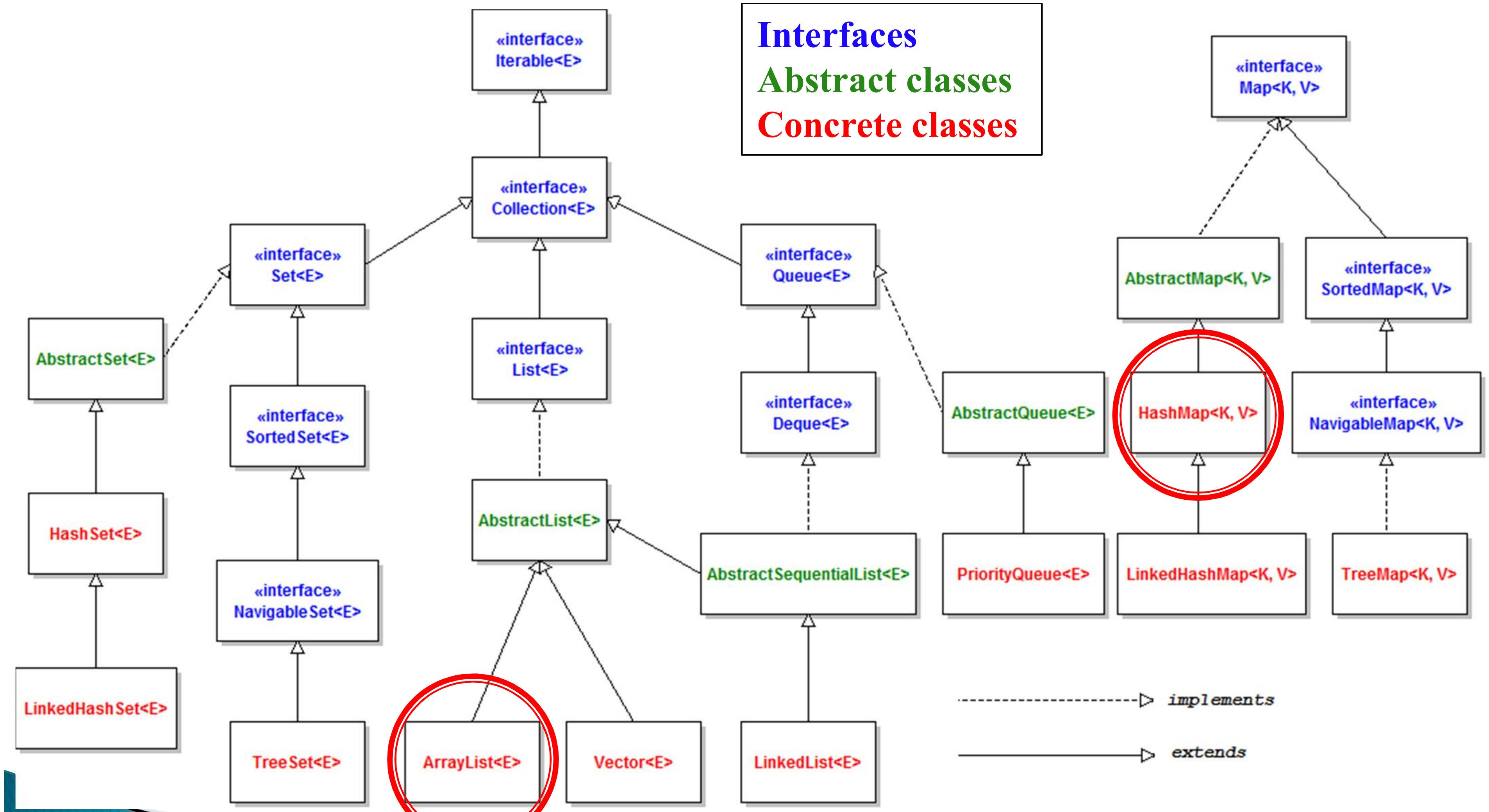

$$\begin{tikzcd} A \arrow[r] & A' \\ B \arrow[r] & B' \\ C \arrow[r] & C' \end{tikzcd}$$

The Map Interface

- `java.util.Map` does not extend `java.util.Collection`. It is not considered to be a true collection and has its own branch in the JCF.
- Methods declared (**not implemented**) in `Map`:
  - `put(K key, V value), putAll` (associating keys and values)
  - `remove(Object key), clear` (removing mappings)
  - `containsKey, containsValue` (checking the existence of keys and values)
  - `keySet, values` (returning a **collection view** of the keys and values)
  - `size` (returning the number of key-value mappings)
  - …

JCF Class Hierarchy

Interfaces
Abstract classes
Concrete classes



http://www.codejava.net/java-core/collections/java-map-collection-tutorial-and-examples

ArrayList

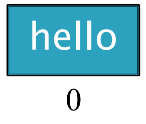
- Arrays store sequences of objects (and primitive values). Arrays do not change their size at runtime to accommodate additional elements.
- `ArrayList<T>` can dynamically change its size at runtime.
- `ArrayList<T>` is a **generic class**, where `T` is a placeholder for the type of elements that you want the `ArrayList` to hold.

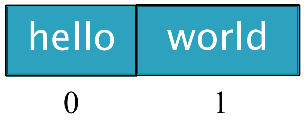
```
ArrayList<String> list;
```

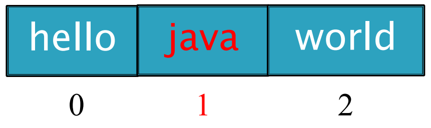
Declares `list` as an `ArrayList` collection to store only `String` objects

Adding Elements to ArrayList

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<String>(); // the list is empty after creation
    printList(list);      // prints nothing since the list is empty
    list.add("hello");    // adding an element to the end of the list
    printList(list);      // prints “hello”
    list.add("world");    // adding one more element to the end
    printList(list);      // prints “hello world”
    list.add(1, "java");// adding one more element to the specified position
    printList(list);     // prints “hello java world”
}
public static void printList(ArrayList<String> list) { // traverse the list
    for(String s : list) System.out.printf("%s ", s); // enhanced for loop
    System.out.println();
}
```







Removing Elements from ArrayList

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello");
list.add("world");
System.out.printf("The list contains %d element(s)\n", list.size());
for(int i = 0; i < list.size(); i++) {
    if(list.get(i).startsWith("w")) list.remove(i);
}
System.out.printf("After removing, the list contains %d element(s)\n", list.size());
```

```
The list contains 2 element(s)
After removing, the list contains 1 element(s)
```

Sorting Elements in ArrayList

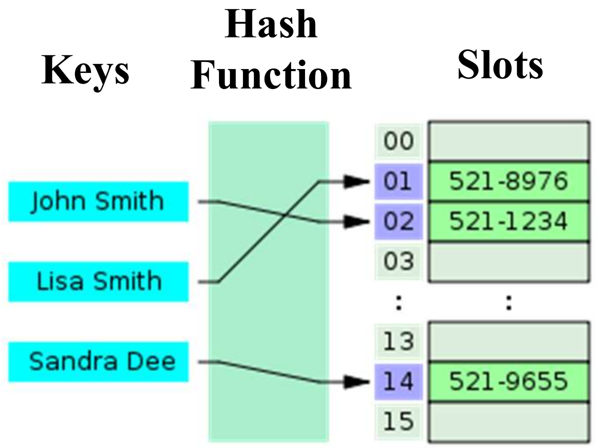
```
public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(new Integer(5));
    list.add(new Integer(124));
    list.add(new Integer(-8));
    printList(list);
    Collections.sort(list);  // sort the elements in the list into ascending order
    printList(list);
}
public static void printList(ArrayList<Integer> list) {
    for(Integer s : list) System.out.printf("%d ", s.intValue());
    System.out.println();
}
```

```
-8 5 124
```

`java.util.Collections` class provides static methods that operate on collections (e.g., `shuffle, reverseOrder, sort`)

HashMap

- `HashMap` (hash table) is a data structure that can map keys to values. It is a concrete implementation of the `Map` interface.
- It uses a **hash function** to compute an index into an array of slots, from which the desired value can be found (**very efficient**).
- A **hash function** is any function that can map data of arbitrary size to data of fixed size. Well-defined hash functions have low chances of collisions (mapping two different keys to the same hash values).



Creating a HashMap

Use interface name to declare variable

```
Map<Integer, String> mapHttpErrors = new HashMap<>();

mapHttpErrors.put(400, "Bad Request"); // key: Integer; Value: String

mapHttpErrors.put(301, "Moved Permanently");

mapHttpErrors.put(404, "Not Found");

mapHttpErrors.put(500, "Internal Server Error");

System.out.println(mapHttpErrors);
```

```
{400=Bad Request, 404=Not Found, 500=Internal Server Error,
301=Moved Permanently}
```

Getting a value associated with a key

```
String status301 = mapHttpErrors.get(301);

System.out.println("301: " + status301);
```

```
301: Moved Permanently
```

Checking existence of keys and values

```java
if (mapHttpErrors.containsKey(301)) {
    System.out.println("Found key");
}

if (mapHttpErrors.containsValue("Bad Request")) {
    System.out.println("Found value");
}
```

```
Found key
Found value
```

Removing a mapping

```
String removedValue = mapHttpErrors.remove(500);

if (removedValue != null) {
    System.out.println("Removed value: " + removedValue);
}
```

```
Removed value: Internal Server Error
```

Update the value of a pair

```
Map<Integer, String> mapHttpErrors = new HashMap<>();
mapHttpErrors.put(500, "Not found");
System.out.println(mapHttpErrors);
mapHttpErrors.put(500, "Internal Server Error");
System.out.println(mapHttpErrors);
```

Simply call the put method: If the map previously contained a mapping for the key, the old value is replaced by the specified value

```
{500=Not found}
{500=Internal Server Error}
```