CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java** (Part 3)

Prepared by

Renetha J.B.
AP
Dept.of CSE,
Lourdes Matha College of Science and Technology

Topics

☑ Java Library

- ❑**Collections framework**
    - ❑ Collections overview
    - ❑Collections Interfaces- Collection Interface

Collections Framework

- The **java.util** package contains one of Java's most powerful subsystems: The ***Collections Framework.***

- The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology(best possible technlogy) for **managing groups of objects**.

  - The **Collection** in **Java** is a **framework** that provides an architecture to **store and manipulate the group of objects**.

  - **Java Collection framework** provides many **interfaces** (Set, List, Queue, Deque) and **classes** (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet)

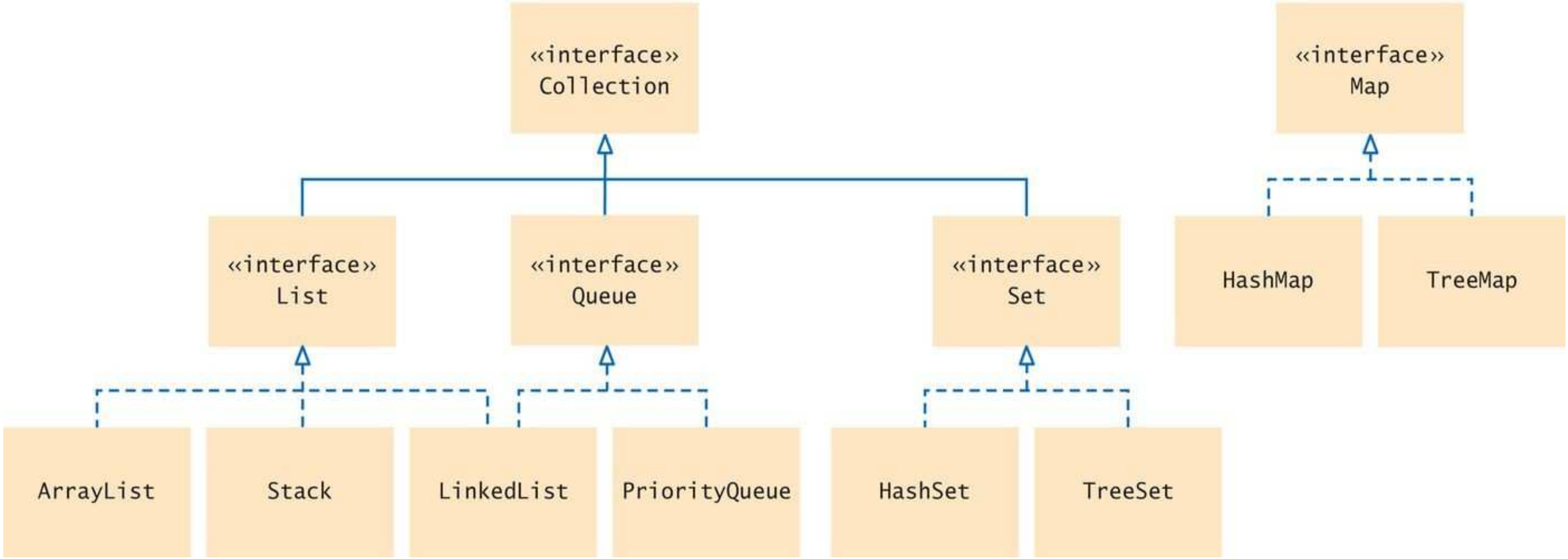


Figure 1 Interfaces and Classes in the Java Collections Framework

Collections Overview

- The Java Collections Framework standardizes **the way in which groups of objects are handled** by our programs.
- The entire Collections Framework is **built upon a set of standard interfaces.**
- Mechanisms were added that allow the **integration of standard arrays** into the Collections Framework.

Collections Overview(contd.)

- The **Collections** Framework was designed to meet several goals.
  - First, the framework had to be **high-performance**.
    - The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
  - Second, the framework had to **allow different types of collections to work in a similar manner** and with a **high degree of interoperability**.
  - Third, **extending and/or adapting** a collection had to be **easy**.

Collections Overview(contd.)

- **Algorithms** are an important part of the **collection** mechanism.
  - *Algorithms* operate on collections and are defined as **static methods** within the **Collections class.**
  - The algorithms provide a standard means of **manipulating collections.**
- **Java Collections Framework** provides **algorithm implementations that are commonly used** such as sorting, searching etc.
  - void sort(List list)
  - int binarySearch(List list, Object value)

Collections Overview(contd.)

- Another item closely associated with the Collections Framework is the Iterator **interface.**
  - An iterator offers a general-purpose, **standardized way of accessing the elements** within a collection, **one at a time**.
  - An iterator provides a means of ***enumerating the contents*** *of a collection.*
  - Because each collection implements **Iterator,** the elements of any collection class can be accessed through the methods defined by **Iterator**

Collections Overview(contd.)

- The framework defines several **map** interfaces and classes.
  - *Maps* store key/value pairs.
    - A **map** cannot contain duplicate keys.
    - Although maps are part of the Collections Framework, they are not “collections” in the strict use of the term

Recent Changes to Collections

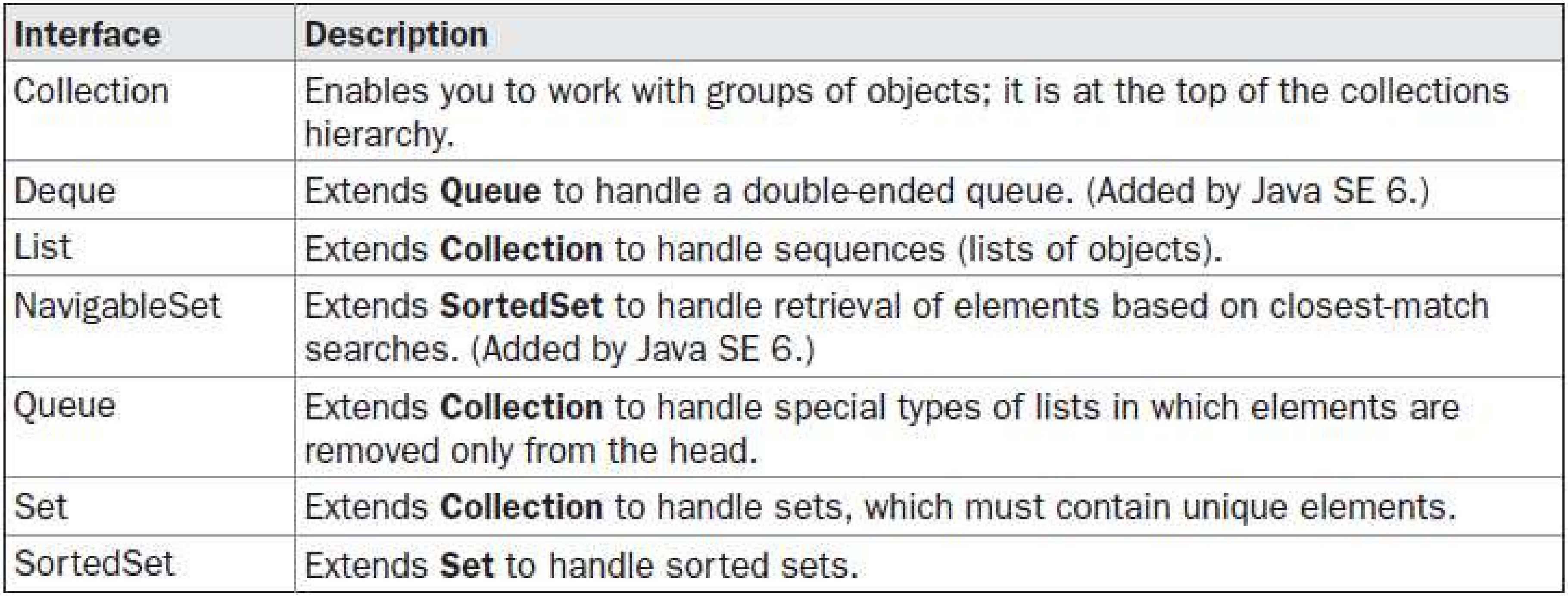
- Collections Framework underwent a fundamental change that significantly increased its power and streamlined its use.
  - The changes were caused by the addition of
    - **generics**
    - **autoboxing/unboxing**, and
    - **for-each** style **for loop.**

Recent Changes to Collections

- **Generics** add the one feature : **type safety**.
  - With generics, it is possible to **explicitly state the type of data** being stored, and run-time type mismatch errors can be avoided.
- **Autoboxing/unboxing** facilitates the **storing of primitive types** in collections.
  - IN THE PAST, if we wanted ***to store a primitive value, such as an int, in a collection*****, we had to manually box it into its type wrapper**.
  - When *the value was* ***retrieved***, it needed to be **manually unboxed** (by using an explicit cast) into its proper primitive type.
  - Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types.

The Collection Framework

- The Collections Framework defines several interfaces.



Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue. (Added by Java SE 6.)
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches. (Added by Java SE 6.)
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

Collection interface

- Collection interface helps to work with group of objects
- The **Collection interface** is at the **top** of collections hierarchy.
- **Collection interface** is the foundation upon which the Collections Framework is built
  - because it must be implemented by any class that defines a collection.
- **Collection** is a generic interface that has this declaration:

```
interface Collection<E>
```

  - Here, **E** specifies the **type of objects** that the collection will hold.
- Collection **extends** the **Iterable interface.**
  - This means that all collections can be cycled through by use of the for-each style **for loop.**

Collection interface(contd.)

Collection declares the **core methods** that all collections will have.

Several of these methods can throw an **UnsupportedOperationException** **if** a collection cannot be modified.

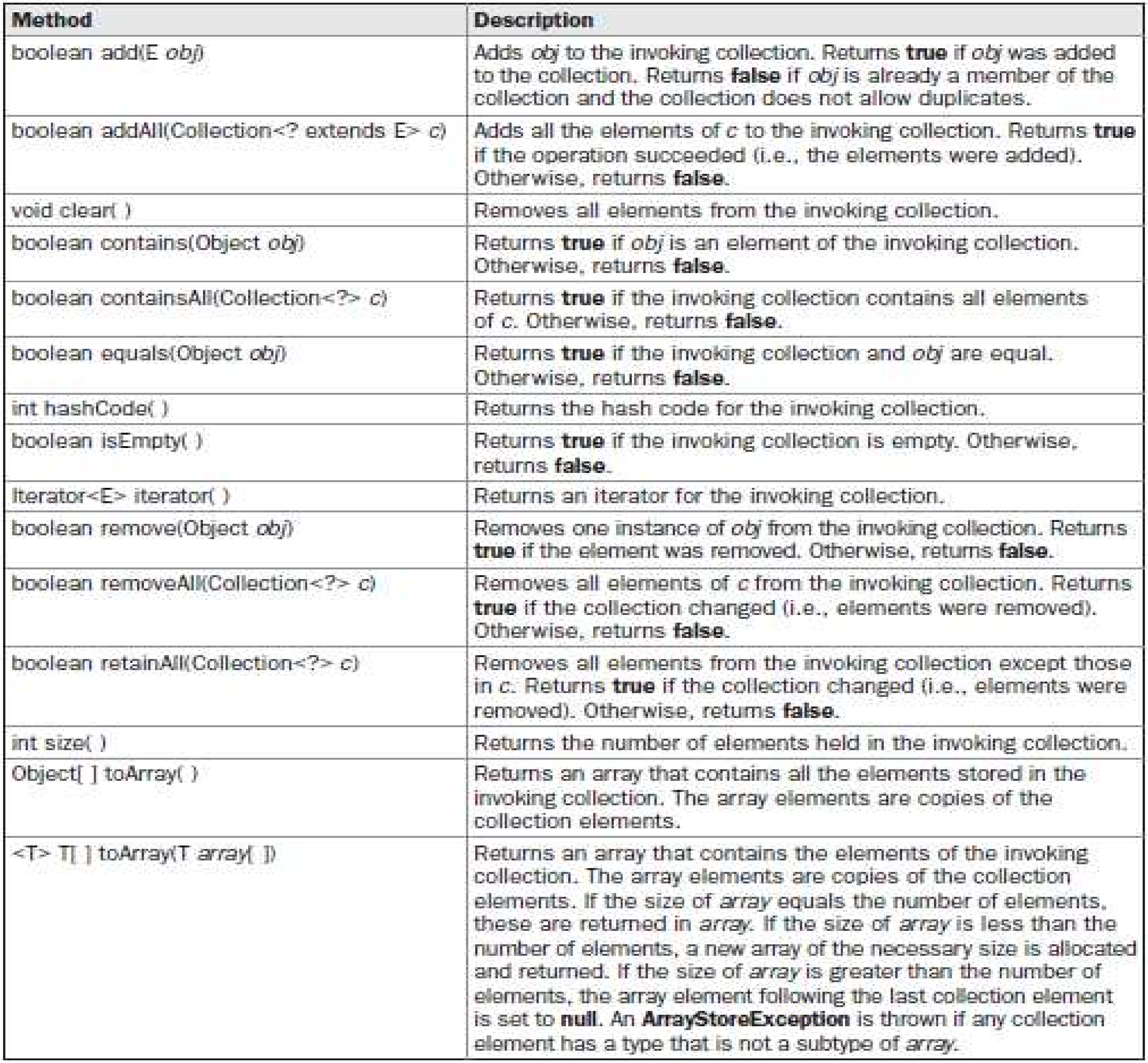
A **ClassCastException** is generated when one object is **incompatible** with another.

A **NullPointerException** is thrown if an attempt is made to **store a null object** and null elements are not allowed in the collection.

An **IllegalArgumentException** is thrown if an **invalid argument** is used.

An **IllegalStateException** is thrown if an attempt is made to **add an element to a fixed-length collection that is ful**l.

The Methods Defined by Collection



Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the operation succeeded (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

Collection interface(contd.)

- Objects are added to a collection by calling **add( ).**
  - **add( )** takes an argument of type **E,** which means that objects added to a collection must be compatible with the type of data expected by the collection.
- To add the entire contents of one collection to another by calling **addAll( ).**
- To remove an object call **remove( ).**
- To remove a group of objects, call **removeAll( ).**
- To remove all elements except those of a specified group by call **retainAll**( ).
- To empty a collection, call **clear( ).**

Collection interface(contd.)

- We can check whether a collection contains a specific object by calling **contains( ).**
- To check whether one collection contains all the members of another, call **containsAll( ).**
- To determine whether a collection is empty call **isEmpty().**
- The number of elements currently held in a collection can be determined by calling **size( ).**
- The **toArray( )** methods return an array that contains the elements stored in the invoking collection.
  - ➢ **Object[ ] toArray**( ) returns an array of **Object.**
  - ➢ **<T> T[ ] toArray(T array[ ])** returns an array of elements that have the same type as the array specified as a parameter.

Collection interface(contd.)

- Two collections can be compared whether they are equal or not by calling **equals( ).**
- The precise meaning of "equality" may differ from collection to collection.
  - **equals( )** can be implemented to ***compare the values of elements*** stored in the collection.
  - **equals( )** can be implemented to ***compare references*** to those elements.
- The method **iterator( )** returns an iterator to a collection.
  - Iterators help to loop through the collections.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.