https://github.com/kkh3363/bsu_java2025

Java Programming
Chapter 12

Java Data Structures

https://github.com/kkh3363/bsu_java2025

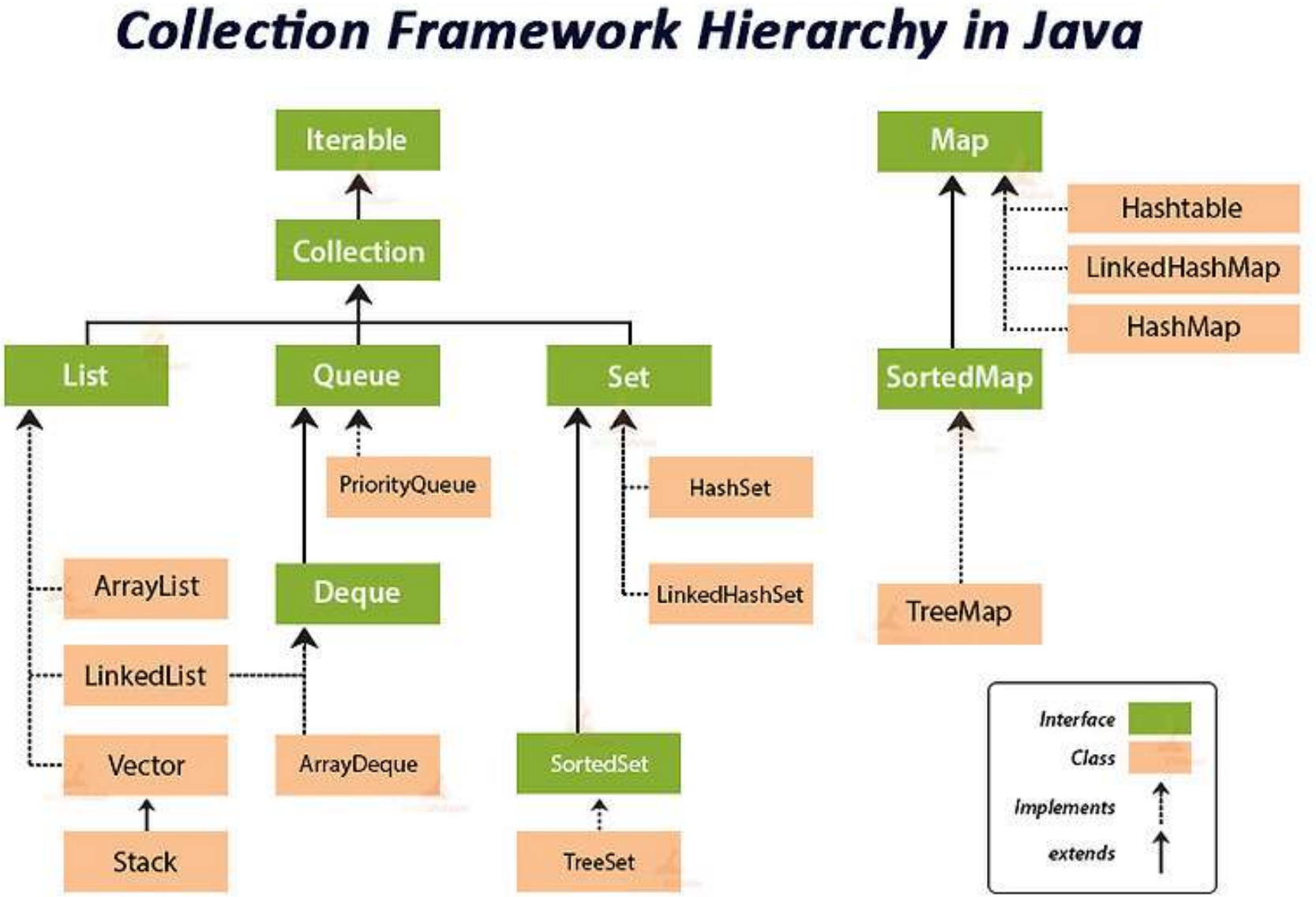
Data Structures

- Data structures are ways to store and organize data so you can use it efficiently.
- An array is an example of a data structure, which allows multiple elements to be stored in a single variable.
- Java includes many other data structures as well, in the java.util package. Each is used to handle data in different ways.
- Some of the most common are:
  - ArrayList
  - HashSet
  - HashMap

Collections Framework

- The Java Collections Framework provides
  - a set of interfaces (like List, Set, and Map)
  - a set of classes (ArrayList, HashSet, HashMap, etc.) that implement those interfaces.
- All of these are part of the java.util package.
- They are used to store, search, sort, and organize data more easily - all using standardized methods and patterns.

Collections Framework



Collections Framework

- Core Interfaces in the Collections Framework

Interface	Common Classes	Description
`List`	`ArrayList`, `LinkedList`	Ordered collection that allow s duplicates
`Set`	`HashSet`, `TreeSet`, `LinkedHashSet`	Collection of unique element s
`Map`	`HashMap`, `TreeMap`, `LinkedHashMap`	Stores key-value pairs with u nique keys

Collections Framework

- Overview of Classes

Interface	Class	Description
List	ArrayList	Resizable array that maintains order and allows duplicates
	LinkedList	List with fast insert and remove operations
Set	HashSet	Unordered collection of unique elements
	TreeSet	Sorted set of unique elements (natural order)
	LinkedHashSet	Maintains the order in which elements were inserted
Map	HashMap	Stores key/value pairs with no specific order
	TreeMap	Sorted map based on the natural order of keys
	LinkedHashMap	Maintains the order in which keys were inserted

Collections Framework

- When Using
  - **List** classes when you care about order, you may have duplicates, and want to access elements by index.
  - **Set** classes when you need to store unique values only.
  - **Map** classes when you need to store pairs of keys and values, like a name and its phone number.

List Interface

- The List interface is part of the Java Collections Framework and represents an ordered collection of elements.
- You can access elements by their index, add duplicates, and maintain the insertion order.
- Since List is an interface, you cannot create a List object directly.
- Instead, you use a class that implements the List interface, such as:
  - ArrayList - like a resizable array with fast random access
  - LinkedList - like a train of cars you can easily attach or remove

List Interface

- Common List Methods

Method	Description
`add()`	Adds an element to the end of the list
`get()`	Returns the element at the specified position
`set()`	Replaces the element at the specified position
`remove()`	Removes the element at the specified position
`size()`	Returns the number of elements in the list

List Interface

- List vs. Array

Array	List
Fixed size	Dynamic size
Faster performance for raw data	More flexible and feature-rich
Not part of Collections Framework	Part of the Collections Framework

ArrayList

- ArrayList
  - An ArrayList is like a resizable array.
  - It is part of the java.util package and implements the List interface.
  - The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want.

ArrayList

- An ArrayList is a resizable array that can grow as needed.
- It allows you to store elements and access them by index.

```java
public class code1201 {
    Run | Debug
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add(e: "Volvo");
        cars.add(e: "BMW");
        cars.add(e: "Ford");

        cars.add(index: 0, element: "Mazda"); // Insert el

        System.out.println(cars);
        // get
        System.out.println(cars.get(index: 0));
        // Change an element
        cars.set(index: 0, element: "Opel");
        System.out.println(cars.get(index: 0));
        //Remove an Element
        cars.remove(index: 0);
        System.out.println(cars);
        // ArrayList Size
        System.out.println(cars.size());
    }
}
```

ArrayList

- Loop Through an ArrayList

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

for-each

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

ArrayList

- Other Types
  - Elements in an ArrayList are actually objects.
  - In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: Integer.
  - For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

ArrayList

```java
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

ArrayList

- Sort an ArrayList
  - Another useful class in the java.util package is the Collections class, which include the sort() method for sorting lists alphabetically or numerically:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    Collections.sort(cars);  // Sort cars
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

ArrayList

- Sort an ArrayList of Integers:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

LinkedList

- LinkedList
  - The LinkedList class is almost identical to the ArrayList:

```java
// Import the LinkedList class
import java.util.LinkedList;

public class Main {
  public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

LinkedList

- ArrayList vs. LinkedList
  - The **LinkedList** class is a collection which can contain many objects of the same type, just like the ArrayList.
  - The **LinkedList** class has the same methods as **ArrayList** because both follow the **List** interface. This means you can add, change, remove, or clear elements in a LinkedList just like you would with an ArrayList.
  - However, while the ArrayList class and the LinkedList class can be used in the same way, they are built very differently.

LinkedList

- LinkedList Methods
  - For many cases, the ArrayList is more efficient as it is common to need access to random elements in the list, but the LinkedList provides several methods to do certain operations more efficiently:

Method	Description
`addFirst()`	Adds an element to the beginning of the list
`addLast()`	Add an element to the end of the list
`removeFirst()`	Remove an element from the beginning of the list
`removeLast()`	Remove an element from the end of the list
`getFirst()`	Get the element at the beginning of the list
`getLast()`	Get the element at the end of the list

```java
package arraysec;
import java.util.LinkedList;

public class LinkedExam {
    Run | Debug
    public static void main(String[] args) {
        LinkedList<String> cars = new LinkedList<String>();
        cars.add(e: "Volvo");
        cars.add(e: "BMW");
        cars.add(e: "Ford");

        // Use addFirst() to add the item to the beginning
        cars.addFirst(e: "Mazda");
        System.out.println(cars);
        // Use addLast() to add the item to the end
        cars.addLast(e: "Mazda");
        System.out.println(cars);
        // Use removeFirst() remove the first item from the list
        cars.removeFirst();
        System.out.println(cars);
        // Use getFirst() to display the first item in the list
        System.out.println(cars.getFirst());
        // Use getLast() to display the last item in the list
        System.out.println(cars.getLast());
    }
}
```

LinkedList

- Sort an ArrayList
  - Sort an ArrayList of Strings alphabetically in ascending order:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    Collections.sort(cars);  // Sort cars

    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

LinkedList

- numerically in ascending order

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

LinkedList

- Reverse the Order

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    Collections.sort(cars, Collections.reverseOrder()); // Sort cars

    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

LinkedList

- Integers numerically in reverse/descending order

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers, Collections.reverseOrder()); // Sort myNumbers

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```