# STRING & JAVA COLLECTION

Instructor:

# Table of contents

- String
- StringBuffer
- StringBuilder
- List Collection

Section 1

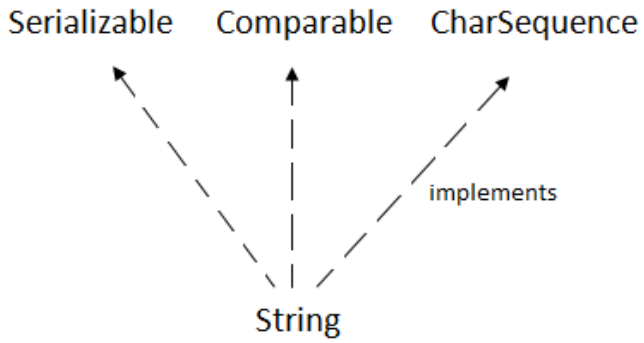
# STRING CLASS

# String class

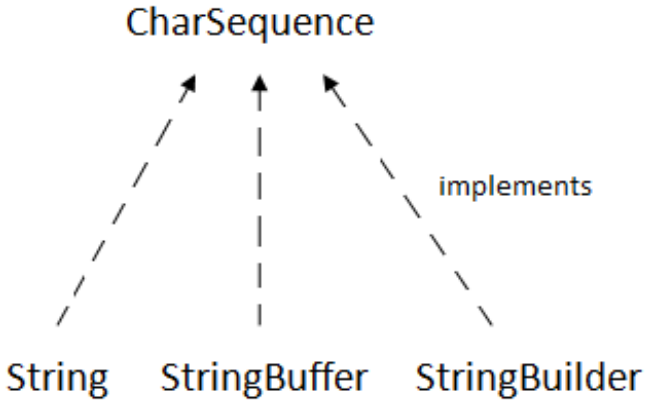
- **String** is a sequence of characters, for e.g. “Hello” is a string of 5 characters.
- In java, string is an **immutable object** which means it is **constant** and can **cannot be changed** once it has been created.
- **Creating a String**
  - ✓ There are two ways to create a String in Java
    - String **literal**
    - Using **new** keyword

# Java String

- The **java.lang.String** class implements **Serializable**, **Comparable** and **CharSequence** interfaces.



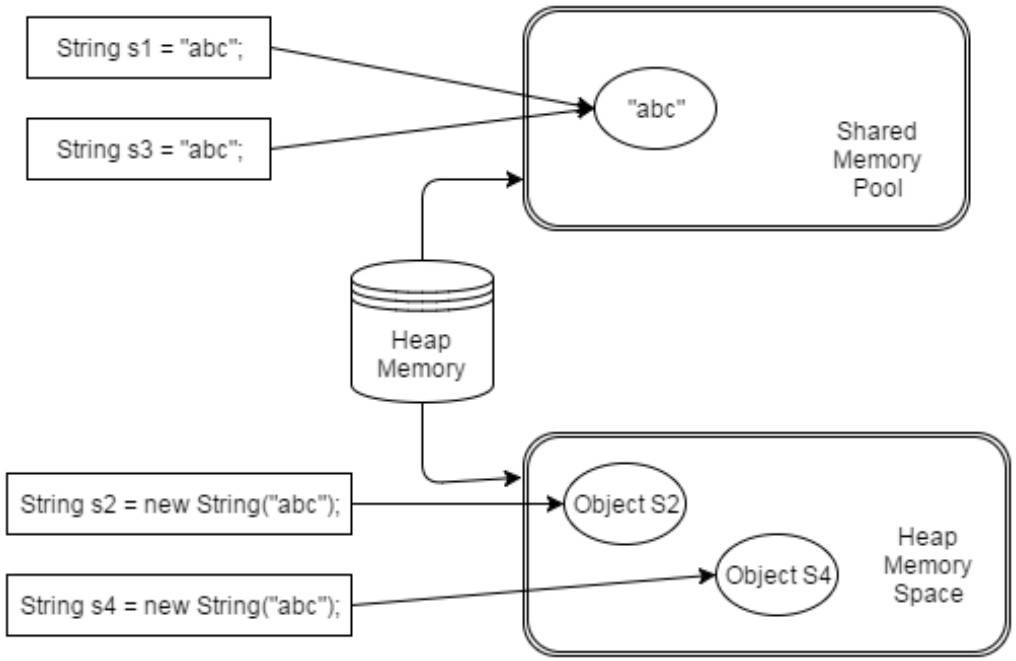

- The **CharSequence** interface is used to represent the sequence of characters. **String**, **StringBuffer** and **StringBuilder** classes implement it.



# String literal

- In java, Strings can be created like this: Assigning a String literal to a String instance.
- Example:

```
String s1= “abc";
String s3= “abc";
```



# String object

- Using new keyword
- The compiler would **create two different object** in memory having the **same string**.
- **Example:**

```
public class StringSample {

  public static void main(String[] args) {
    // creating a string by java string literal
    String s1 = "FPT";
    String s3 = "FPT";

    char arrch[] = { 'h', 'e', 'l', 'l', 'o' };
    // converting char array arrch[] to string str2
    String s2= new String(arrch);

    // creating another java string str3 by using new keyword
    String s4 = new String("hello");

    // Displaying all the three strings
    System.out.println(s1.equals(s3));
    System.out.println(s2.equals(s4));

    System.out.println(s1 == s3);
    System.out.println(s2 == s4);
  }
}
```

**Output:**

```
true
true
true
false
```

# Immutable String

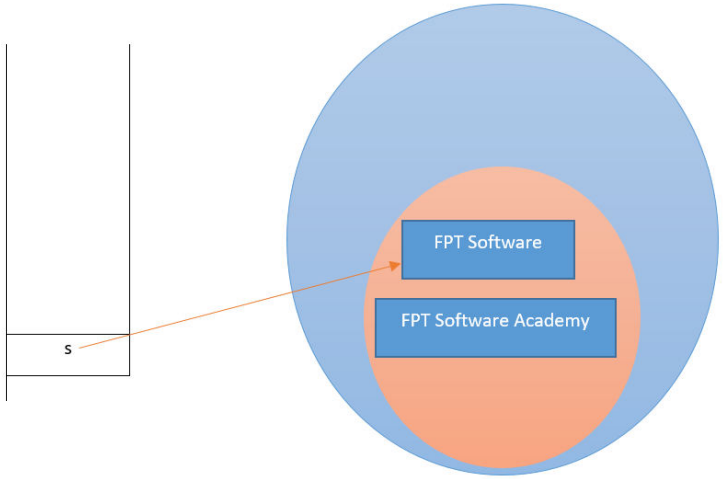
- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
- **Example**:

```
public class TestImmutableString {

  public static void main(String[] args) {
    String s = "FPT Software";
    s.concat(" Academy"); // concat() method appends the string at the end
    System.out.println(s);// will print FPT Software because strings are
                          // immutable objects
  }
}
```

- **Output**:

```
FPT Software
```




**Solution:**

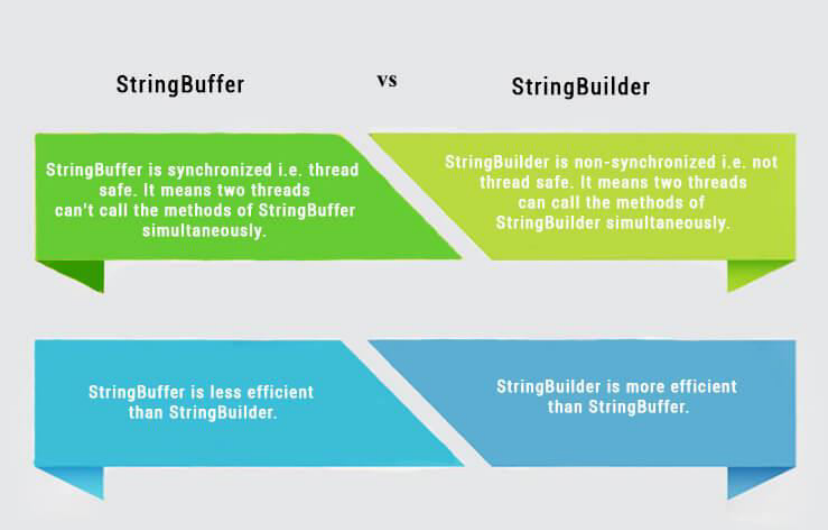
```
s = s.concat(" Academy");
```

# String Comparison

- We can compare string in java on the basis of content and reference.
- It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.
- There are three ways to compare string in java:
  - ✓ In **authentication:** by equals()/equalsIgnoreCase() method
  - ✓ In **sorting:** by == operator
  - ✓ **Reference matching**: by compareTo() method. This method compares values **lexicographically** (từ vựng) and returns an integer value that describes if first string is *less than*, *equal to* or *greater* than second string.

# StringBuilder and StringBuffer class

- The **StringBuffer** and **StringBuilder** classes: to make a lot of modifications to Strings of characters.
- The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilders methods are *not thread safe* (not Synchronised).
- It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However if thread safety is necessary the best option is StringBuffer objects.



# String/StringBuilder/StringBuffer

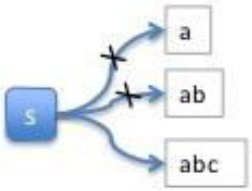
- String is **immutable**, if you try to alter their values, another object gets created,
- StringBuffer and StringBuilder are **mutable** so they can change their values.

## String

### Immutable

Every time you alter String values, it will allocate another exact amount of space in the heap. The previous value in the memory will be garbage-collected later.

```
String s = "a";
s += "b";
s += "c";
```

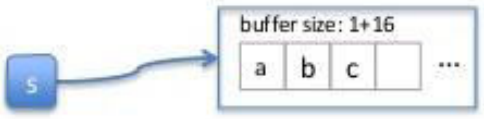



## StringBuffer

### Mutable

When created it reserves a certain amount of space in the heap, which can be larger than the value. Within that space, values can be modified without additional memory use. When the value requires more space, the space will automatically grow larger.

```
StringBuffer s = new StringBuffer("a");
s.append("b");
s.append("c");
```



# StringBuilder class

- The Java **StringBuilder** class is same as **StringBuffer** class except that it is non-synchronized.

| No. | StringBuffer | StringBuilder |
|---|---|---|
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

- **Example**:

```
public class ConcatTest {
  public static void main(String[] args) {
    long startTime = System.currentTimeMillis();

    StringBuffer sb = new StringBuffer("Java");
    for (int i = 0; i < 1000000; i++) {
      sb.append(" Learning");
    }

    System.out.println("Time taken by StringBuffer: "
        + (System.currentTimeMillis() - startTime) + "ms");

    startTime = System.currentTimeMillis();

    StringBuilder sb2 = new StringBuilder("Java");
    for (int i = 0; i < 1000000; i++) {
      sb2.append(" Learning");
    }

    System.out.println("Time taken by StringBuilder: "
        + (System.currentTimeMillis() - startTime) + "ms");
  }
}
```

- **Output:**

```
Time taken by StringBuffer: 51ms
Time taken by StringBuilder: 26ms
```

# StringBuilder/StringBuffer examples

```
StringBuilder sb = new StringBuilder("abc");
```

- ✓ `sb.append(" def");` // "abc def"
- ✓ `char letter = str.charAt(2);` // "b"
- ✓ `char ch[] = new char[3];`

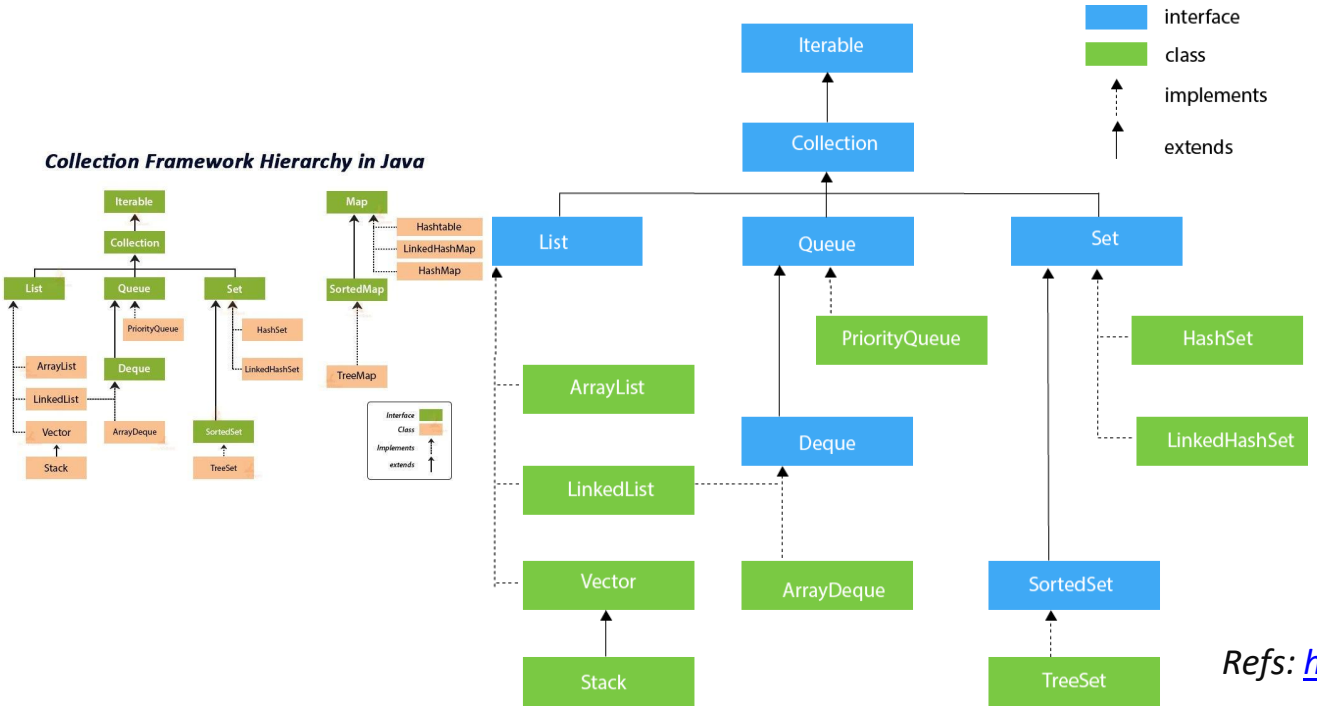
  `str.getChars(1,3,ch,0);` // Bây giờ biến "ch" chứa "abc"
- ✓ `sb.delete(3, 5);` // "abcef"
- ✓ `sb.deleteCharAt(4);` // "abce"
- ✓ `sb.insert(3, " d");` // "abc de"
- ✓ `sb.replace(2, 4, " ghi");` // "ab ghide"
- ✓ `sb.reverse();` // "edihg ba"
- ✓ `sb.setCharAt(5, 'j');` // "edihgjba"

Section 2

# JAVA COLLECTIONS

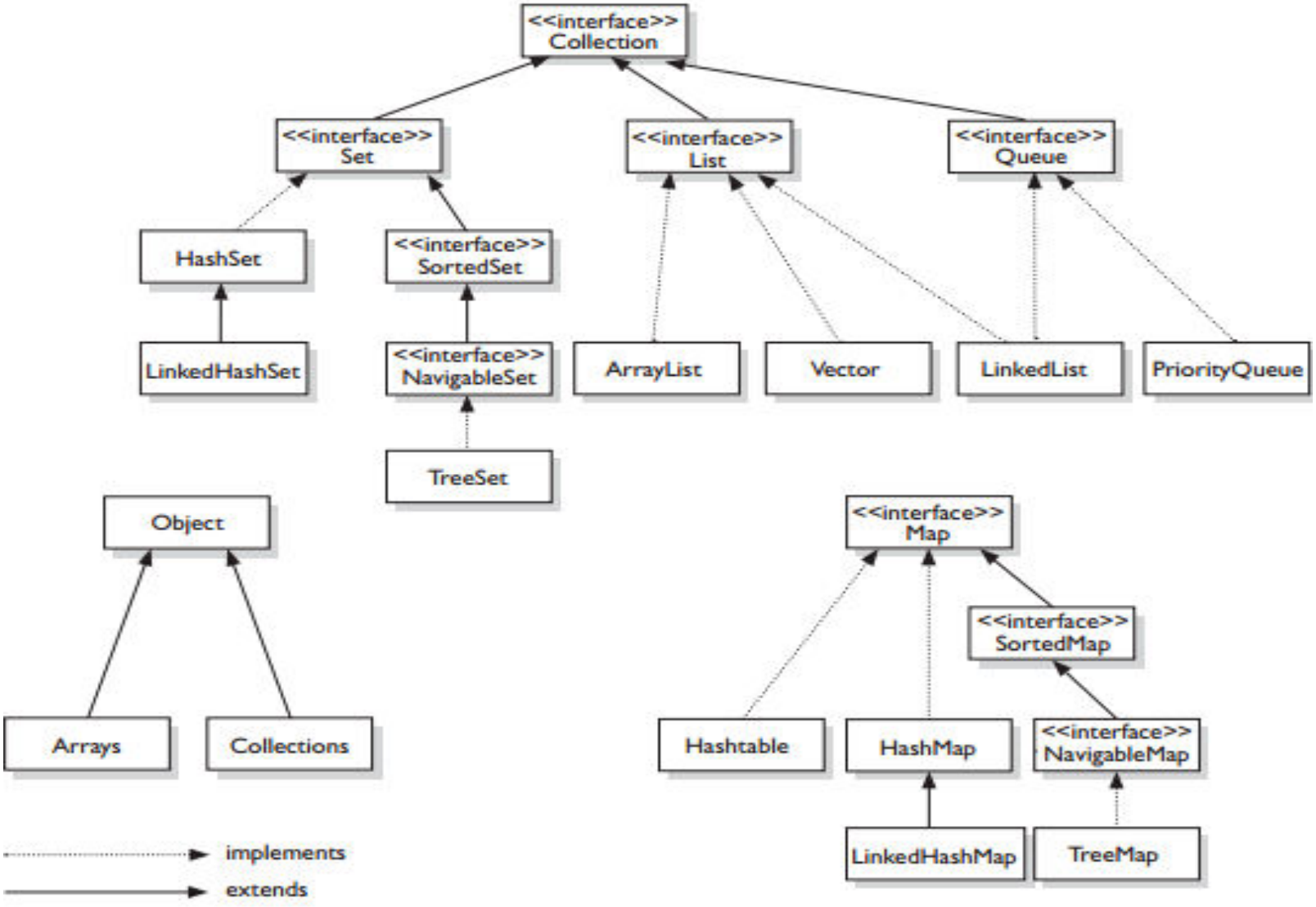
# Overview

- The Collection in Java is a framework that provides an architecture to **store** and **manipulate** the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (**Set**, **List**, **Queue**, **Deque**) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



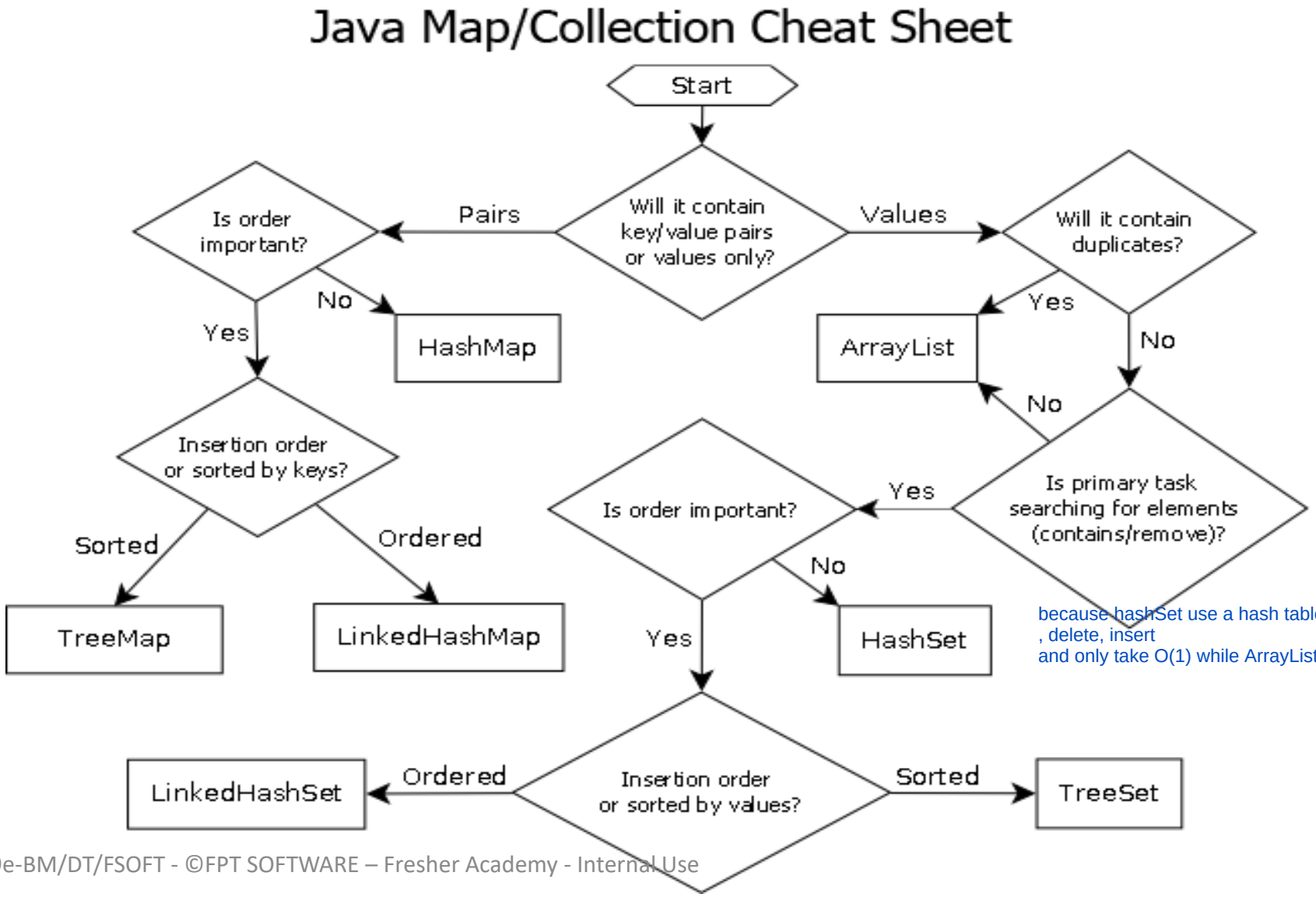

*Refs: https://www.javatpoint.com/*

# Overview


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
HashSet
<<interface>>
SortedSet
LinkedHashSet
<<interface>>
NavigableSet
ArrayList
Vector
LinkedList
PriorityQueue
TreeSet
Object
<<interface>>
Map
<<interface>>
SortedMap
Arrays
Collections
Hashtable
HashMap
<<interface>>
NavigableMap
LinkedHashMap
TreeMap
implements
extends

# Java Collection Cheat Sheet

- Collection Interface

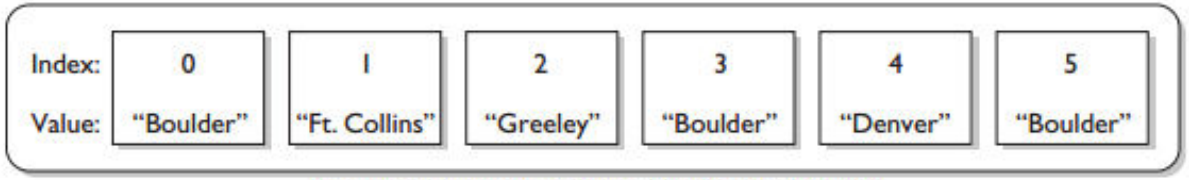


# Java Collection

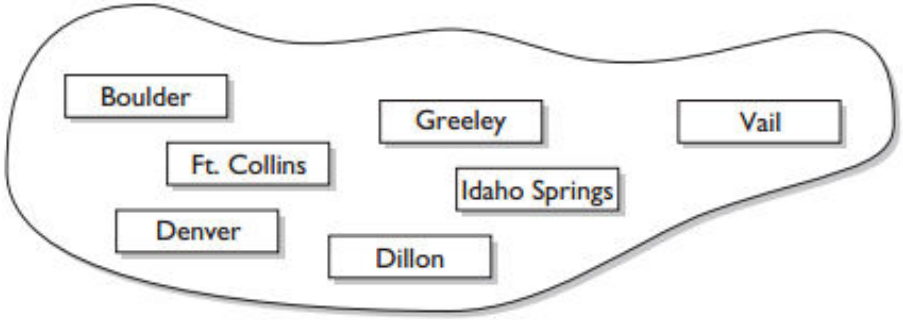
Figure 7-3 illustrates the structure of a List, a Set, and a Map.

**FIGURE 7-3**

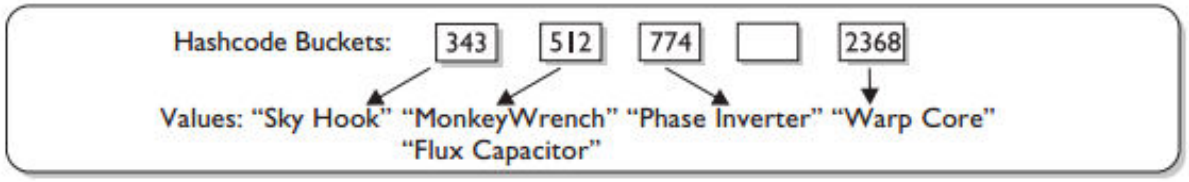
The structure of a List, a Set, and a Map




List: The salesman's itinerary (Duplicates allowed)




Set: The salesman's territory (No duplicates allowed)




HashMap: the salesman's products (Keys generated from product IDs)

Section 3

# LIST COLLECTION

# List Interface

- **List** interface is the child interface of **Collection** interface.
- List interface is implemented by the classes **ArrayList**, **LinkedList**, **Vector**, and **Stack**.
- To instantiate the List interface, we must use:

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

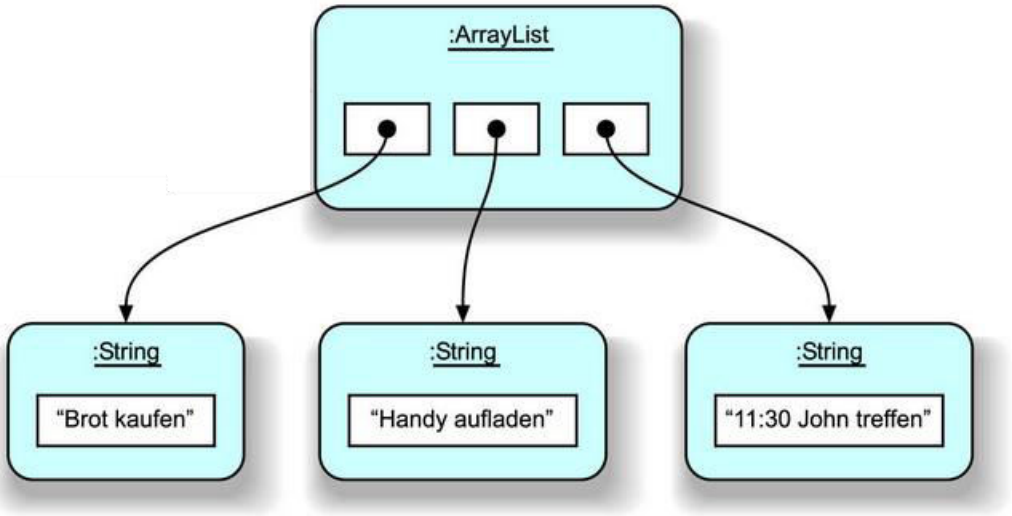
- There are various methods in List interface that can be used to *insert*, *delete*, and *access* the elements from the list.

# ArrayList

- **ArrayList** supports dynamic arrays that can grow as needed.
  - ✓ Array lists are created with an initial size.
  - ✓ When this size is exceeded, the collection is automatically enlarged.
  - ✓ When objects are removed, the array may be shrunk
- **Syntax:**

  **List**<DataType> arrName = new **ArrayList<>**();

mặc định tạo 10 phần tử



# ArrayList

- **ArrayList** is implemented as a resizable array. The important points about Java ArrayList class are:
  - ✓ Java ArrayList class can contain duplicate elements.
  - ✓ Java ArrayList class maintains insertion order.
  - ✓ Java ArrayList class is non synchronized.
  - ✓ Java ArrayList allows random access because array works at the index basis. The elements in an ArrayList can be accessed directly and efficiently by using the get() and set() methods.
  - ✓ In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

insert, delete

- ArrayList class declaration:

```
public class ArrayList<E> extends AbstractList<E>
            implements List<E>, RandomAccess, Cloneable, Serializable
```

# Main methods of ArrayList

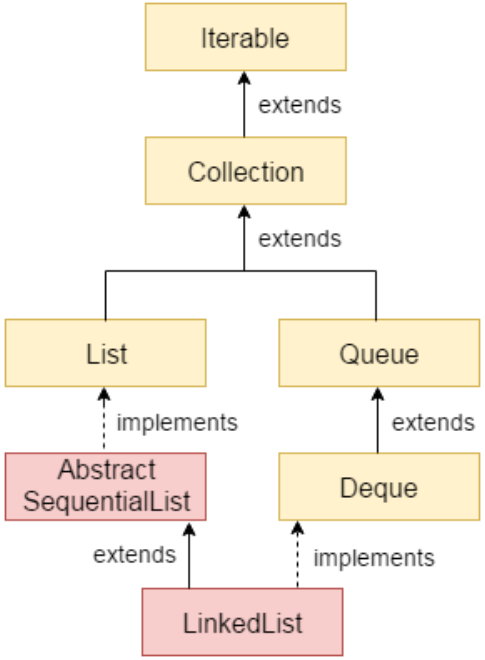
| Constructor | Description |
|---|---|
| ArrayList() | It is used to build an empty array list. |
| ArrayList(Collection<? extends E> c) | It is used to build an array list that is initialized with the elements of the collection c. |
| ArrayList(int capacity) | It is used to build an array list that has the specified initial capacity. |
| **Method** | **Description** |
| void add(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean add(E e) | It is used to append the specified element at the end of a list. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| E get(int index) | It is used to fetch the element from the particular position of the list. |
| boolean isEmpty() | It returns true if the list is empty, otherwise false. |
| boolean contains(Object o) | It returns true if the list contains the specified element |

# Main methods of ArrayList

| Method | Description |
|---|---|
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element. |
| boolean removeAll(Collection<?> c) | It is used to remove all the elements from the list. |
| boolean removeIf(Predicate<? super E> filter) | It is used to remove all the elements from the list that satisfies the given predicate. |
| protected void removeRange(int fromIndex, int toIndex) | It is used to remove all the elements lies within the given range. |
| void retainAll(Collection<?> c) | It is used to retain all the elements in the list that are present in the specified collection. |
| ist<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements lies within the given range. |
| int size() | It is used to return the number of elements present in the list. |

# LinkedList

- Java **LinkedList** class uses a doubly linked list to store the elements.
- The important points about Java LinkedList are:
  - ✓ Java LinkedList class can contain duplicate elements.
  - ✓ Java LinkedList class maintains insertion order.
  - ✓ Java LinkedList class is non synchronized.
  - ✓ In Java LinkedList class, manipulation is fast because no shifting needs to occur.
  - ✓ Java LinkedList class can be used as a list, stack or queue.



# LinkedList

- **Doubly Linked List**


fig- doubly linked list

- **Declaration:**

```
public class LinkedList<E> extends AbstractSequentialList<E>
                implements List<E>, Deque<E>, Cloneable, Serializable
```

- **LinkedList** is implemented as a double linked list. Its performance on add() and remove() is better than the performance of Arraylist. The get() and get() methods have worse performance than the ArrayList, as theLinkedList does not provide direct access.

| | ArrayList | LinkedList |
|---|---|---|
| get() | O(1) | O(n) |
| add() | O(1) | O(1) amortized |
| remove() | O(n) | O(n) |

# Main methods of LinkedList

| Method | Description |
|---|---|
| void addFirst(E e) | It is used to insert the given element at the beginning of a list. |
| void addLast(E e) | It is used to append the given element to the end of a list. |
| E get(int index) | It is used to return the element at the specified position in a list. |
| E getFirst() | It is used to return the first element in a list. |
| E getLast() | It is used to return the last element in a list. |
| E peek() | It retrieves the first element of a list |
| E peekFirst() | It retrieves the first element of a list or returns null if a list is empty. |
| E peekLast() | It retrieves the last element of a list or returns null if a list is empty. |
| E poll() | It retrieves and removes the first element of a list. |

# Main methods of LinkedList

| Method | Description |
|---|---|
| E poll() | It retrieves and removes the first element of a list. |
| E pollFirst() | It retrieves and removes the first element of a list, or returns null if a list is empty. |
| E pollLast() | It retrieves and removes the last element of a list, or returns null if a list is empty. |
| E pop() | It pops an element from the stack represented by a list. |
| void push(E e) | It pushes an element onto the stack represented by a list. |
| E removeFirst() | It removes and returns the first element from a list. |
| E removeLast() | It removes and returns the last element from a list. |

# LinkedList Example

```
public class TestLinkedList {

  public static void main(String args[]) {

    LinkedList<String> ll = new LinkedList<String>();
    System.out.println("Initial list of elements: " + ll);
    ll.add("Java");
    ll.add("Net");
    ll.add("Android");
    System.out.println("After invoking add(E e) method: " + ll);
    // Adding an element at the specific position
    ll.add(1, "iOs");
    System.out
        .println("After invoking add(int index, E element) method: " + ll);
    LinkedList<String> ll2 = new LinkedList<String>();
    ll2.add("Test");
    ll2.add("Automation Test");
    // Adding second list elements to the first list
    ll.addAll(ll2);
    System.out.println(
        "After invoking addAll(Collection<? extends E> c) method: " + ll);
    // Adding an element at the first position
    ll.addFirst("C++");
    System.out.println("After invoking addFirst(E e) method: " + ll);
    // Adding an element at the last position
    ll.addLast("Kotlin");
    System.out.println("After invoking addLast(E e) method: " + ll);

    ll.removeFirst();
    System.out.println("After invoking removeFirst() method: " + ll);

    ll.removeLast();
    System.out.println("After invoking removeLast() method: " + ll);

  }

}
```

**Output:**

```
Initial list of elements: []
After invoking add(E e) method: [Java, Net,
After invoking add(int index, E element) met
After invoking addAll(Collection<? extends E
After invoking addFirst(E e) method: [C++, J
After invoking addLast(E e) method: [C++, Ja
```

# Difference between ArrayList and LinkedList

| ArrayList | LinkedList |
|---|---|
| 1. ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2. Manipulation with ArrayList is **slow** because it internally uses an array.<br>If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3. An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4. ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

# Summary

- String
- StringBuffer
- StringBuilder
- List Collection

# Thank you