Java Collection Framework

Set

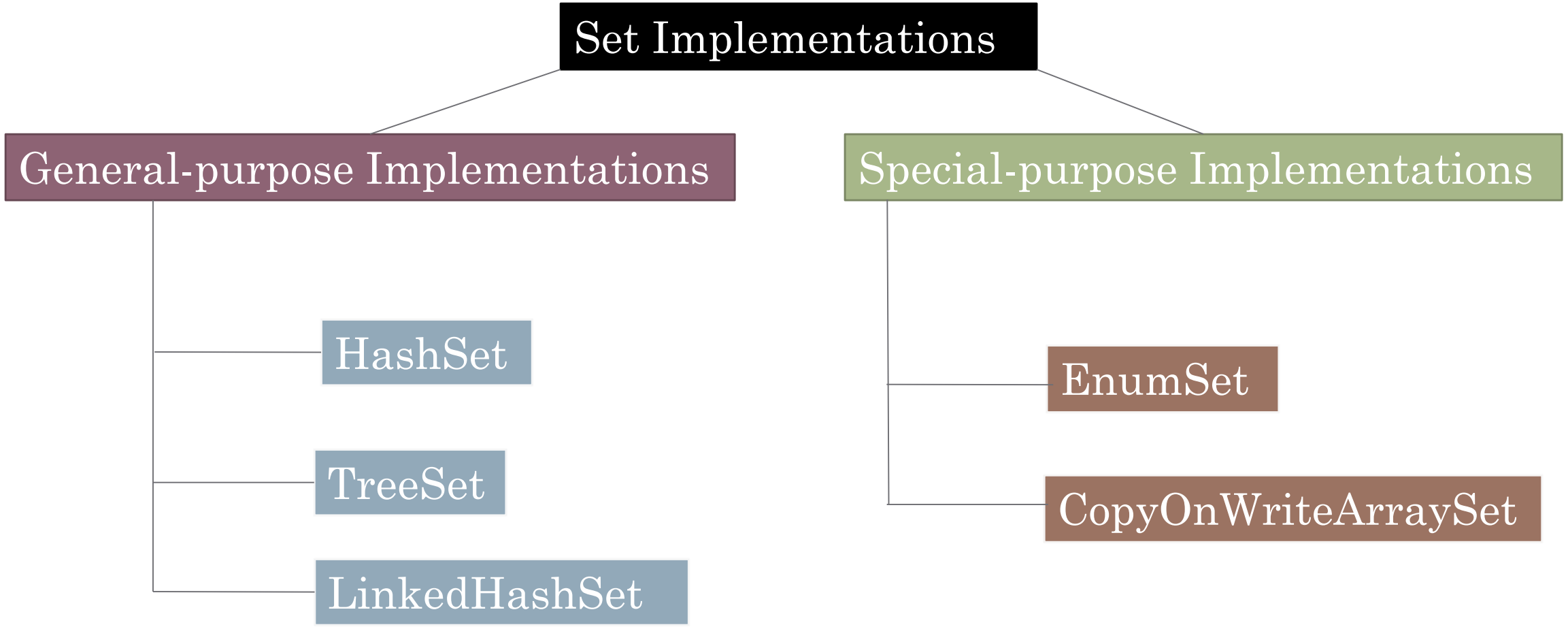
Mahboob Ali

Set *Interface*

- The Set interface extends the Collection interface.
- It does not introduce new methods or constants, but it stipulates that an instance of Set contains no duplicate elements.
- The concrete classes that implement Set must ensure that no duplicate elements can be added to the set.
- That is no two elements e1 and e2 can be in the set such that e1.equals(e2) is true.

Set Implementation

- Set implementations are grouped into two groups



Hash Set

- No duplicates.
- Useful when *uniqueness* & *fast lookup* matters
- HasSet is much faster than TreeSet (constant-time vs long-time for most operations)
- Insertion order does not matter (no guarantee of order).
- Default capacity is 16.
- Capacity increases with power of 2. (X^2)
- It is a Hash Table implementation of a Set interface.
- Internally uses **HashMap.**

Typical Uses and Useful methods

- **Quick** *lookup, insertion, and deletion* ~ O(1)
- **Insertion** order is not important
- **Better** for *removeAll()* and *retainAll()*
- ***add() //To add elements to the set***
- ***contains() //To check if a particular element present in the HashSet***
- ***remove() //To remove the specified element***
- ***clear() //To remove all the elements***
- ***size() //To check number of elements***
- ***isEmpty() //To check if an instance of HasSet is empty or not***
- ***iterator() //Return an iterator over the elements, elements visited in no order.***

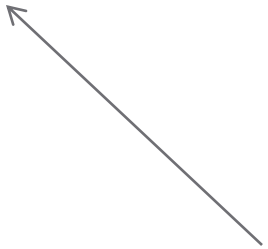
```
import java.util.*;

public class TestHashSet {
  public static void main(String[] args) {
    // Create a hash set with default capacity
    Set<String> set = new HashSet<>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    System.out.println(set);

    // Display the elements in the hash set
    for (String s: set) {
      System.out.print(s.toUpperCase() + " ");
    }

    // Process the elements using a forEach method
    System.out.println();
    set.forEach(e -> System.out.print(e.toLowerCase() + " "));
  }
}
```

Added twice

[San Francisco, New York, Paris, Beijing, London]

Lambda Expression replacing the Inner class Implementation

```
import java.util.HashSet;
import java.util.Set;

public class SetDemo {

 private static void hashSetDemo() {

 Book book1 = new Book("Walden", "Henry Thoreau", 1854);
 Book book2 = new Book("Walden", "Henry Thoreau", 1854);
       Set<Book> set2 = new HashSet<>();
       set2.add(book1);
       set2.add(book2);
       System.out.println("set2: " +  set2);
 }

 public static void main(String[] args) {
       hashSetDemo();
 }

}
```



set2: [Book [title=Walden, author=Henry Thoreau , year=1854], Book [title=Walden, author=Henry Thoreau, year=1854]]

```
class Book {
	private String title;
	private String author;
	private int year;
	public String getTitle() {
			return title;		}
	public void setTitle(String title) {
			this.title = title;	}
	public String getAuthor() {
			return author;		}
	public void setAuthor(String author) {
			this.author = author;}
	public int getYear() {
			return year;		}
	public void setYear(int year) {
			this.year = year;	}
	public Book(String title, String author, int year) {
			super();
			this.title = title;
			this.author = author;
			this.year = year;
	}
	@Override
	public String toString() {
		return "Book [title=" + title + ",
	author=" + author + ", year=" + year + "]";
	}

	public int hashCode() {
		return title.hashCode();
	}

	public boolean equals(Object o) {
		return (year == (((Book)o).getYear())) &&
	(author.equals(((Book)o).getAuthor())));
	}

}
```

After implementing the Hashcode:
set2: [Book [title=Walden, author=Henry Thoreau
, year=1854]]

LinkedHashSet

- Extends HashSet with a linked-list implementation.
- Supports the order of the elements in which they are entered.
- Provides insertaion-ordered iteration (least recently inserted to most recently) and run almost as fast as HashSet.
- If you don’t need to main the order of the elements then use HashSet. (which is more efficient)
- The other tuning parameters works just same as HashSet, but the iteration time is not affected by the capacity.

```
import java.util.*;

public class TestLinkedHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new LinkedHashSet<>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    System.out.println(set);

    // Display the elements in the hash set
    for (String elements: set) {
      System.out.print(elements.toLowerCase() + " ");
    }
}
```

Output:
[London, Paris, New York, San Francisco, Beijing]
london paris new york san francisco beijing

TreeSet

- Guarantees that the elements in the set are sorted.
- Sorts the elements in ascending order.
- A TreeSet should be our primary choice if we want to keep our entries sorted as a TreeSet may be accessed and traversed in either ascending or descending order.
- Operations like add, remove and search takes longer time than in the HashSet.
- first() and last() methods, return the first and last elements in the set.

Set and List Performance

```
import java.util.*;
public class SetListPerformanceTest {
  static final int N = 5000;

  public static void main(String[] args) {
        // Add numbers 0, 1, 2, ..., N - 1 to the array list
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < N; i++)
      list.add(i);
    Collections.shuffle(list); // Shuffle the array list

    // Create a hash set, and test its performance
    Collection<Integer> set1 = new HashSet<>(list);
    System.out.println("Member test time for hash set is " + getTestTime(set1) + " milliseconds");
    System.out.println("Remove element time for hash set is " + getRemoveTime(set1) + " milliseconds");

    // Create a linked hash set, and test its performance
    Collection<Integer> set2 = new LinkedHashSet<>(list);
    System.out.println("Member test time for linked hash set is " + getTestTime(set2) + " milliseconds");
    System.out.println("Remove element time for linked hash set is " + getRemoveTime(set2) + " milliseconds");
```

Member test time for hash set is 20 milliseconds

Remove element time for hash set is 27 milliseconds

Member test time for linked hash set is 27 milliseconds

Remove element time for linked hash set is 26 milliseconds

```
// Create a tree set, and test its performance
    Collection<Integer> set3 = new TreeSet<>(list);
    System.out.println("Member test time for tree set is " + getTestTime(set3) + " milliseconds");
    System.out.println("Remove element time for tree set is " +
      getRemoveTime(set3) + " milliseconds");

    // Create an array list, and test its performance
    Collection<Integer> list1 = new ArrayList<>(list);
    System.out.println("Member test time for array list is " +
      getTestTime(list1) + " milliseconds");
    System.out.println("Remove element time for array list is " +
      getRemoveTime(list1) + " milliseconds");

    // Create a linked list, and test its performance
    Collection<Integer> list2 = new LinkedList<>(list);
    System.out.println("Member test time for linked list is " +
      getTestTime(list2) + " milliseconds");
    System.out.println("Remove element time for linked list is " +
      getRemoveTime(list2) + " milliseconds");
  }
```

Member test time for tree set is 47 milliseconds

Remove element time for tree set is 34 milliseconds

Member test time for array list is 39802 milliseconds

Remove element time for array list is 16196 milliseconds

Member test time for linked list is 52197 milliseconds

Remove element time for linked list is 14870 milliseconds

```
public static long getTestTime(Collection<Integer> c) {
    long startTime = System.currentTimeMillis();

    // Test if a number is in the collection
    for (int i = 0; i < N; i++)
      c.contains((int)(Math.random() * 2 * N));

    return System.currentTimeMillis() - startTime;
  }

  public static long getRemoveTime(Collection<Integer> c) {
    long startTime = System.currentTimeMillis();

    for (int i = 0; i < N; i++)
      c.remove(i);

    return System.currentTimeMillis() - startTime;
  }
}
```

Conclusion: **Sets** are more efficient than **List** for testing whether an element is in a set or list.