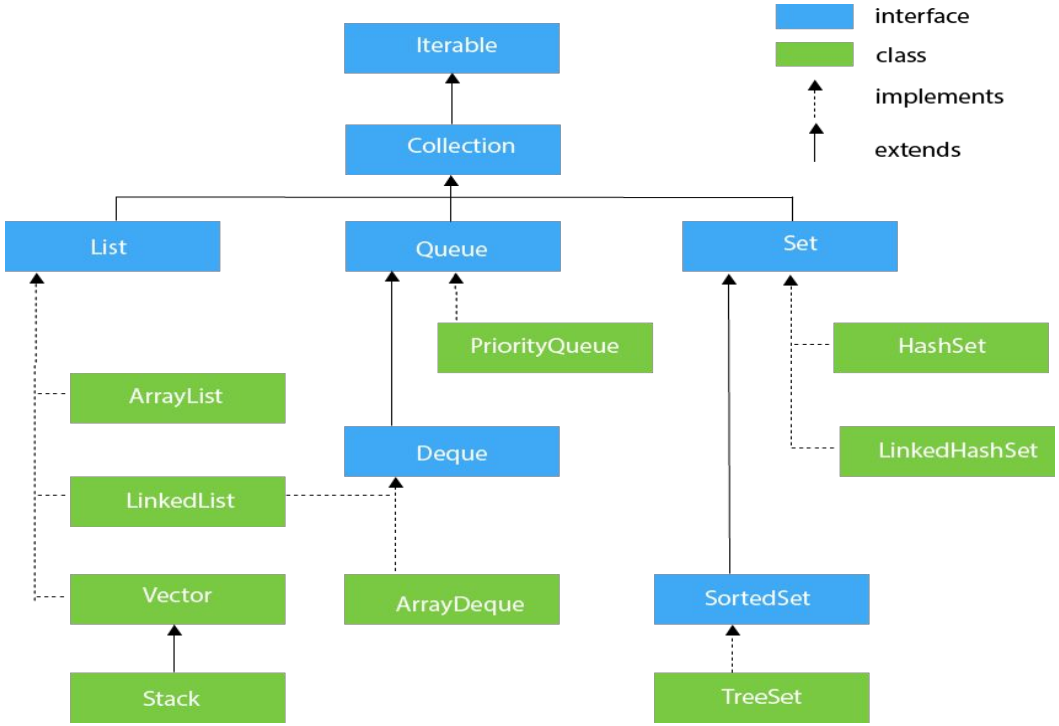
Java Collections & Java 8 Features I

SDET Program

Outline

- Collection in Java
- Map
- Ordering
- Java 8

Java Collection Hierarchy



Collection

- Group of objects
- It’s not specified whether they are
  - Ordered / Not Ordered
  - Duplicated / Not duplicated
- Collections in Java are **generic**, which means you specify the type of elements they hold, providing **type safety** at compile time
- Following constructors are common to all classes implementing Collection
  - T()
    - ```
      List<Integer> list = new ArrayList<>();
      ```
  - T(Collection c)
    - ```
      List<Integer> list = Arrays.asList(1,1,3,4,5);
      Set<Integer> set = new HashSet<>(list);
      ```

Collection Interface

- `int size()`
- `boolean isEmpty()`
- `boolean contains(Object element)`
- `boolean containsAll(Collection c)`
- `boolean add(Object element)`
- `boolean addAll(Collection c)`
- `boolean remove(Object element)`
- `boolean removeAll(Collection c)`
- `void clear()`
- `Object[] toArray()`
- `Iterator iterator()`

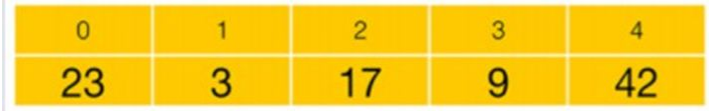
List

- Can contain duplicate elements
- Insertion order is preserved
- User can define insertion order
- Elements can be accessed by position

```
Object get(int index)
Object set(int index, Object element)
void   add(int index, Object element)
Object remove(int index)

boolean addAll(int index, Collection c)
int indexOf(Object o)
int lastIndexOf(Object o)
List subList(int fromIndex, int toIndex)
```

List Implementation

- **ArrayList<E>**
  - get(index): O(1)
  - add(index, element) if index = 0: O(n)
  - add(element): O(1)
  - remove(index): O(n)
- **LinkedList<E>**
  - get(index): O(n)
  - addFirst() & getFirst(): O(1)
  - addLast() & getLast(): O(1)
  - remove(index): O(n)
  - remove(Node): O(1)



Queue

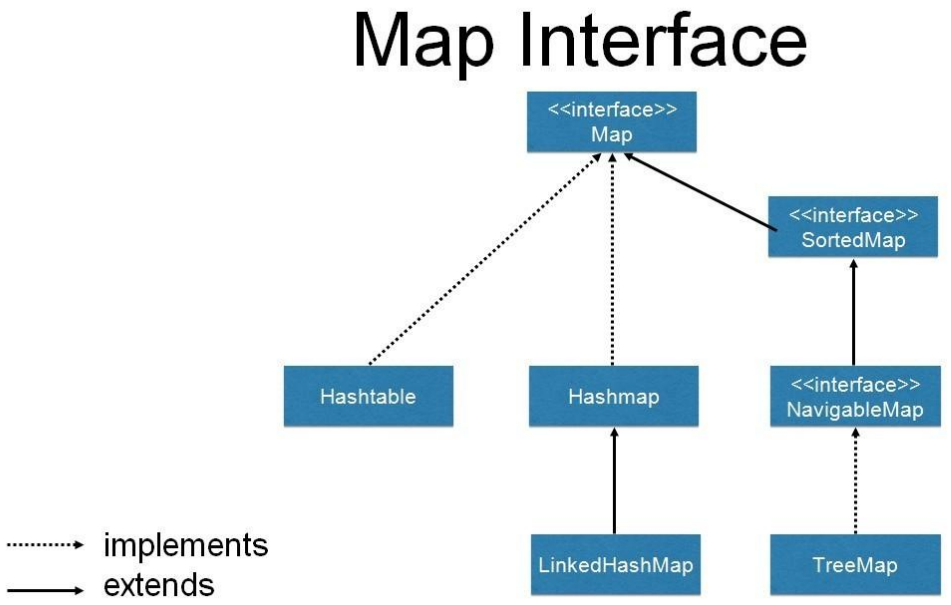
- Collection whose elements have an order
  - **FIFO**: **F**irst **I**n **F**irst **O**ut
- Defines a head position where is the first element that can be accessed
  - offer()
  - poll()
  - peek()
- Queue Implementation
  - **LinkedList<E>**
    - Head is the first element in the queue
  - **PriorityQueue<E>**
    - Head is the smallest/largest element in the queue

Set

- Contains no methods other than those inherited from Collection
- add() method has restriction that no duplicate elements are allowed
- Set Implementation
  - **HashSet<E>**
    - Insertion order is **NOT** preserved
  - **LinkedHashSet<E>**
    - Insertion order **IS** preserved

Map Hierarchy

- Map doesn't extend the Collection interface, it has a separate hierarchy



Map

- An object that associates keys to values (k-v pairs)
- Keys and values must be objects (primitive types must be boxed/Wrapper Classes)
- Keys must be unique
- Only one value per key

- `Object put(Object key, Object value)`
- `Object get(Object key)`
- `Object remove(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `public Set keySet()`
- `public Collection values()`
- `int size()`
- `boolean isEmpty()`
- `void clear()`

Map Example

```
Map<String, String> map = new HashMap<>();
map.put("Doe", "a deer, a female deer");
map.put("Ray", "a drop of golden sun");
map.put("Me", "a name I call myself");
map.put("Far", "a long, long way to run");

System.out.println(map.get("Me")); // a name I call myself
System.out.println(map.keySet()); // [Far, Me, Doe, Ray] (order may vary)

map.remove("Far");
System.out.println(map.containsKey("Far")); // false
System.out.println(map.values());
// [a name I call myself, a deer, a female deer, a drop of golden sun] (order may vary)
```

Map

- get()/put() takes constant time (in case of no collisions)
- Implementations
  - HashMap<K, V>
    - No ordering
  - LinkedHashMap<K, V>
    - Maintains insertion order
  - TreeMap <K, V>
    - Maintains natural order

- `Object put(Object key, Object value)`
- `Object get(Object key)`
- `Object remove(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `public Set keySet()`
- `public Collection values()`
- `int size()`
- `boolean isEmpty()`
- `void clear()`

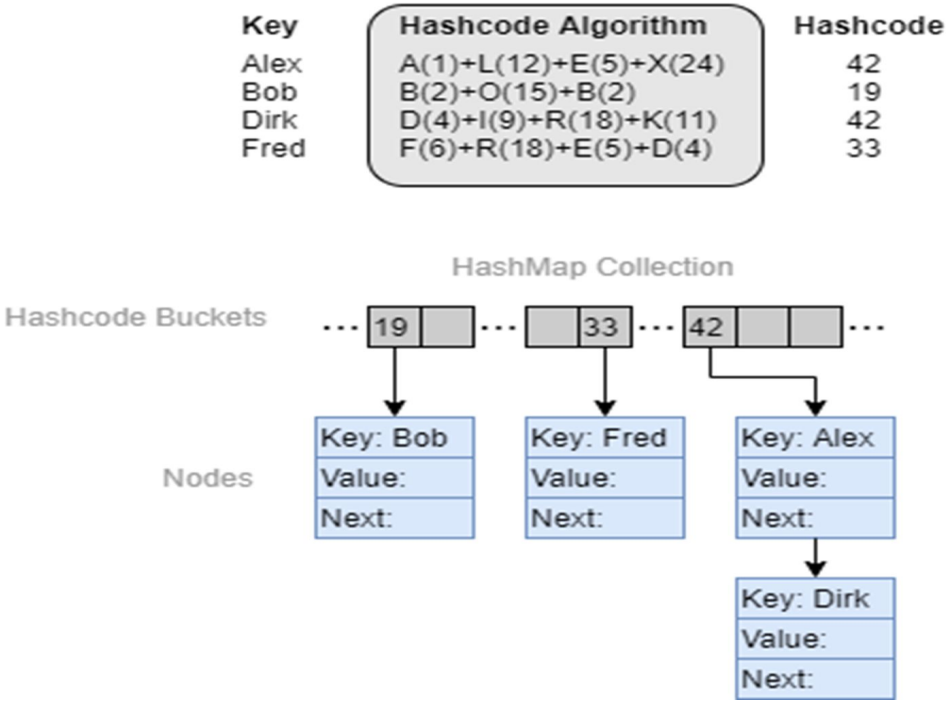
LinkedHashMap & TreeMap Example

```
// LinkedHashMap maintains insertion order
LinkedHashMap<Integer, String> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put(1, "one");
linkedHashMap.put(3, "three");
linkedHashMap.put(2, "two");
System.out.println(linkedHashMap.toString()); // Output: {1=one, 3=three, 2=two}

// TreeMap sorts keys in natural order
TreeMap<Integer, String> sortedHashMap = new TreeMap<>();
sortedHashMap.put(1, "one");
sortedHashMap.put(3, "three");
sortedHashMap.put(2, "two");
System.out.println(sortedHashMap.toString()); // Output: {1=one, 2=two, 3=three}
System.out.println(sortedHashMap.firstKey()); // Output: 1
```

How is HashMap Implemented in Java

- Hashing
  - Hashing is a process of converting an object into an integer value (called hash code) using the hashCode() method.
  - This hash code determines the index (bucket) where the entry (k-v pair) will be stored in a hash-based collection.
- Bucket
  - A bucket is a slot in the underlying array of a HashMap.
  - Each bucket can store one or more k-v pairs(entries/nodes), the entries are linked together using a LinkedList (or a tree in case of high collision, since Java 8).

hashCode() Method



equals() Method

- Usage of equals()
  - Used to compare the **logical equality** of two objects.
  - Default implementation in Object class compares references.
  - Often overridden to compare values. (e.g., in String, Integer)
- Java Contract
  - Reflexive: x.equals(x) must be true
  - Symmetric: if x.equals(y) is true, then y.equals(x) must be true
  - Transitive: if x.equals(y) and y.equals(z), then x.equals(z) must also be true
  - Consistent: multiple calls return the same result
  - Non-null: x.equals(null) must be false

equals() vs ==

- Compares
  - == compares object references
  - equals() compares object content (if overridden)
- Override needed?
  - == cannot be overridden
  - equals() often overridden for custom value checks

hashCode() and equals() Contract in HashMap

- HashMap uses:
  - hashCode() to determine the bucket
  - equals() to find the correct key in that bucket if collision happens
- If you override equals(), you **MUST** override hashCode() accordingly
- hashCode() and equals() contract:
  - If a.equals(b) is true, then a.hashCode() == b.hashCode() must be true
  - If a.hashCode() == b.hashCode(), a.equals(b) may return false
  - Violating this contract breaks collections like HashMap, HashSet, and HashTable

Object Ordering

- How to sort collections?
- Java allows manual sorting algorithms
  - Bubble sort
  - Insertion sort
  - Quick sort
  - Merge sort
- Java also allows built-in utilities
  - For List<E>
    - Use **Collections.sort(list)**
    - It uses **TimSort** internally: merge sort + insertion sort
  - For Array
    - Use Arrays.sort(array)

Natural Ordering

- Many core Java classes implement an interface called Comparable<T> to define natural ordering
- This enables built-in support for sorting in Collections.sort(), Arrays.sort(), TreeSet<E>, TreeMap<K, V> etc.
- Examples of Natural Ordering:
  - String in Lexicographical (Alphabetical)
  - Number and subclasses in Numeric (Ascending)
  - Date/LocalDate in Chronological

Custom Ordering

- Comparable<T> interface
  - Used to define natural ordering of objects
  - The class itself must implement the interface and override the compareTo() method
- Comparator<T> interface
  - Used to define custom ordering, separate from the class's definition
  - Can implement multiple ways to sort the same type

Custom Ordering (Comparable & Comparator)

```java
// Movie.java
public class Movie implements Comparable<Movie> {
    private double rating;
    private String name;
    private int year;

    // Constructor
    public Movie(String name, double rating, int year) {
        this.name = name;
        this.rating = rating;
        this.year = year;
    }

    // Natural order: sort by year (ascending)
    @Override
    public int compareTo(Movie other) {
        return this.year - other.year;
    }

    // Getters
    public double getRating() {
        return rating;
    }

    public String getName() {
        return name;
    }

    public int getYear() {
        return year;
    }
}
```

```java
// RatingCompare.java
public class RatingCompare implements Comparator<Movie> {
    @Override
    public int compare(Movie m1, Movie m2) {
        return Double.compare(m1.getRating(), m2.getRating());
    }
}
```

```java
// NameCompare.java
public class NameCompare implements Comparator<Movie> {
    @Override
    public int compare(Movie m1, Movie m2) {
        return m1.getName().compareTo(m2.getName());
    }
}
```

Custom Ordering

- Override compare(o1, o2) or compareTo(o2), return value must follow this contract
  - < 0 → if o1 comes before o2
  - ==0 → if o1 is equal to o2 in order
  - > 0 → if o1 comes after o2

```
// Natural order: sort by year (ascending)
@Override
public int compareTo(Movie other) {
    return this.year - other.year;
}
```

Java 8 Enhances Custom Ordering

Java 8 new features

- Functional Interface
- Lambda Expressions
- Method Reference

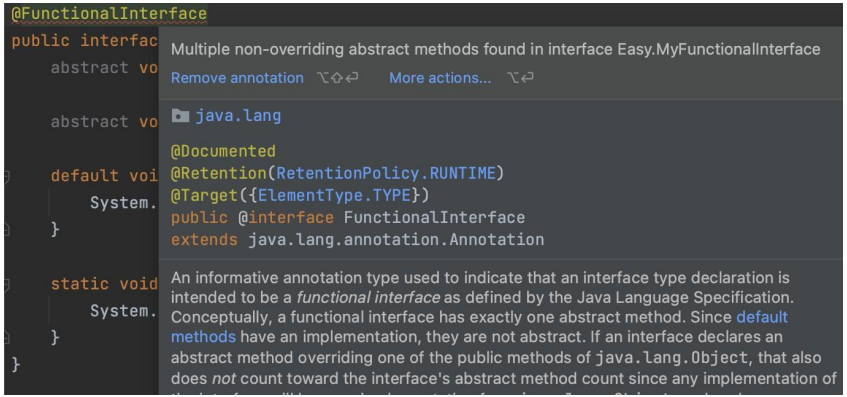
Functional Interface (Java 8)

- A functional interface is an interface that contains
  - Exactly ONE ABSTRACT method
  - Any number of default or static methods
    - Java 8 also introduce default and static method to allow implementation in interfaces
- Examples of functional interfaces:
  - **Comparator<T>** → int compare(T a, T b)
  - **Comparable<T>** → int compareTo(T o)
  - Consumer<T> → void accept(T t)
  - Supplier<T> → T get()
  - Function<T, R> → R apply(T t)
  - Predicate<T> → boolean test(T t)

Functional Interface (Java 8)

- @FunctionalInterface annotation is optional but recommended
- Ensures that only one abstract method is allowed
  - Compiler will throw error if more than one abstract method exists

```java
public interface MyFunctionalInterface {

    void oneAbstractMethod();

    void anotherAbstractMethod();

    default void oneDefaultMethod() { System.out.println("This is a default method"); }

    default void anotherDefaultMethod() { System.out.println("This is another default method"); }

    static void oneStaticMethod() { System.out.println("This is one static method"); }

    static void anotherStaticMethod() { System.out.println("This is another static method"); }
}
```



Lambda Expression (Java 8)

- Enables writing anonymous methods
- Simplifies code by removing boilerplate code in writing functional interface
- Replace anonymous classes used before Java 8
- Syntax:
  - `(param) -> expression`
  - `(param) -> {statement}`

```
// Anonymous class is used before Java 8
Comparator<String> byLength = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
};
```

```
// Lambda Expression is used after Java 8 to simplify code
Comparator<String> byLength = (s1, s2) -> s1.length() - s2.length();
```

Functional Interface & Lambda Expression

- Question: Why lambda expression works well with functional interface?
  - Functional interface contains ONLY ONE abstract method
  - Lambda expression provides the implementation of one abstract method
  - No ambiguity: only one method to implement
- Together they simplify code and reduce verbosity

```
@FunctionalInterface
interface Printer {
    void print(String msg);
}

Printer p = msg -> System.out.println(msg);
```

Method Reference (Java 8)

- A shorter alternative to lambda expressions
- Refers to an existing method of functional interface by name
- Can be used to reference static, non-static methods and constructor
  - Static method → `ClassName::staticMethodName`
  - Instance method → `objectRef::instanceMethodName`
  - Constructor → `ClassName::new`
- Improves code readability

```
@FunctionalInterface
interface Printer {
    void print(String msg);
}

Printer p = System.out::println;
```

Thank You!