Object-Oriented Programming Using Java

Java Collections Framework

Quan Thai Ha

HUS

April 2, 2024

Presentation Outline

1 Java Collection Framework

2 Lists

3 Sets

4 Queues

5 Maps

6 Collections and Iterators

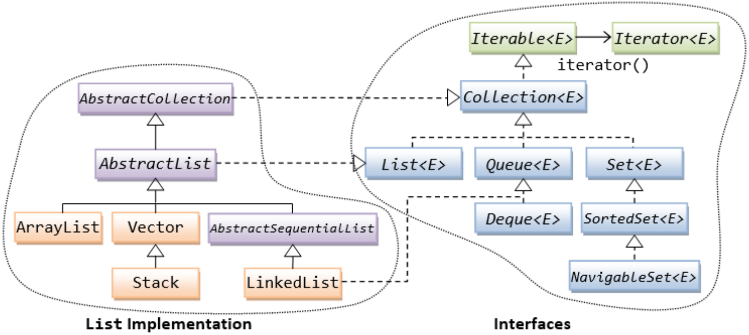
7 Algorithms

Java Collections Framework

- Although we can use an array as a container to store a group of elements of the same type (primitives or references). The array, however, does not support so-called dynamic allocation - it has a fixed length which cannot be changed once allocated. Furthermore, array is a simple linear structure. Many applications may require more complex data structure such as linked list, stack, hash table, set, or tree.
- In Java, dynamically allocated data structures (such as ArrayList, LinkedList, Vector, Stack, HashSet, HashMap, Hashtable) are supported in a unified architecture called the Collection Framework, which mandates the common behaviors of all the classes.
- A collection, as its name implied, is simply a container object that holds a collection of objects. Each item in a collection is called an element. A framework, by definition, is a set of interfaces that force you to adopt some design practices. A well-designed framework can improve your productivity and provide ease of maintenance.
- The collection framework provides a unified interface to store, retrieve and manipulate the elements of a collection, regardless of the underlying actual implementation. This allows the programmers to program at the interface specification, instead of the actual implementation.

Java Collections Framework

- The Java Collection Framework (JCF) is a set of classes and interfaces implementing commonly reusable data structures.
- The JCF (package java.util) provides
  - A set of interfaces defining functionalities;
  - Abstract classes for shared code aggregation;
  - Implementation classes implementing functionalities;
  - Algorithms (such as sorting and searching).

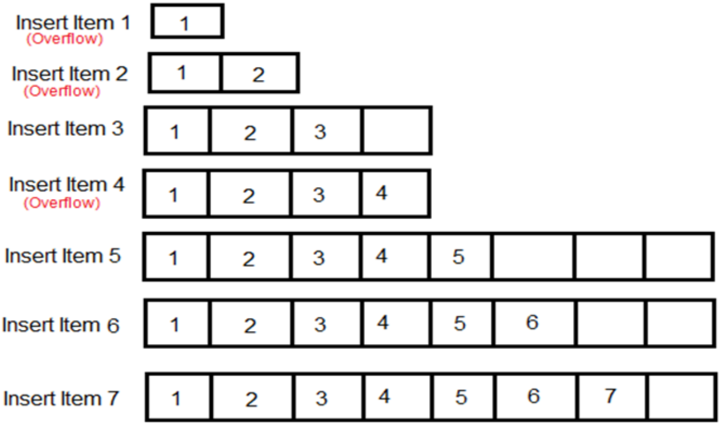


Key Concepts

- Resizable Array
- Linked List
- Balanced Tree
- Hash Table

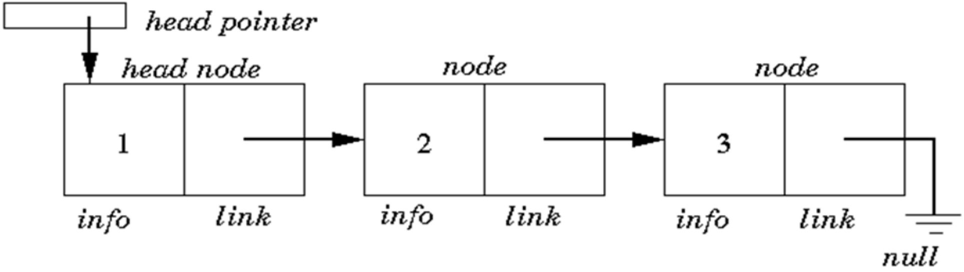
Resizable Array ~O(n)

Initially table is empty and size is 0

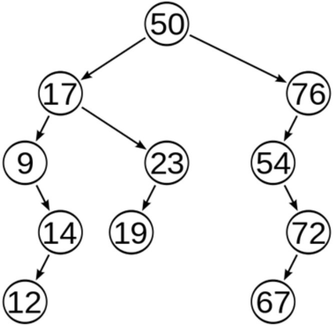


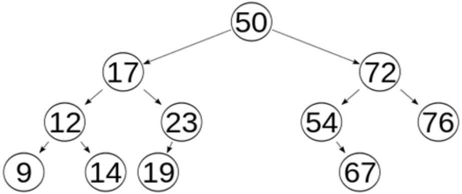
Next overflow would happen when we insert 9, table size would become 16

Linked List ~O(n)



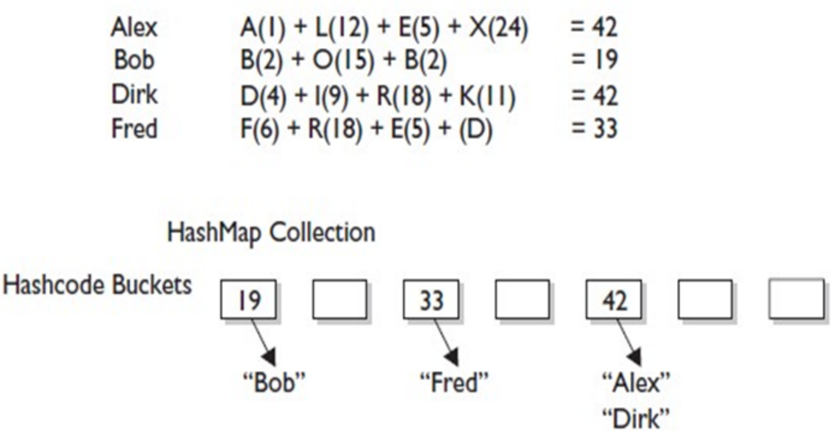
Balanced Tree ∼O(log(n))





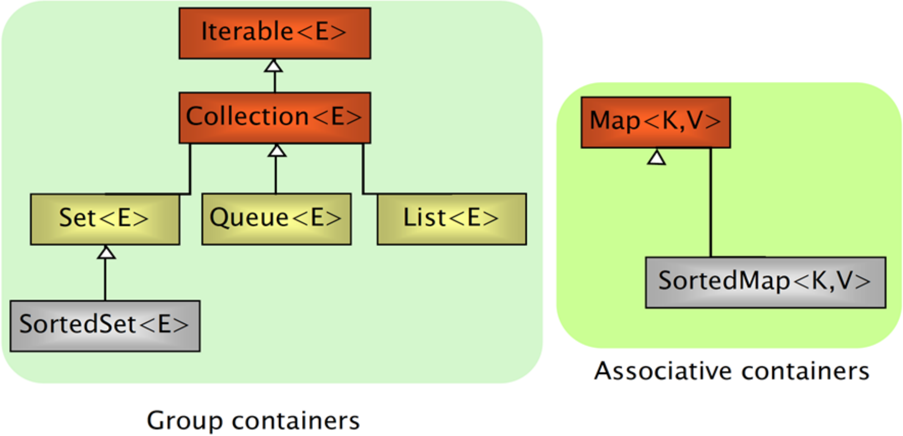
- *A binary tree is balanced if, for each node it holds that, the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1.*
- *A binary tree is balanced if for any two leaves the difference of the depth is at most 1.*

Hash Table ∼O(1)

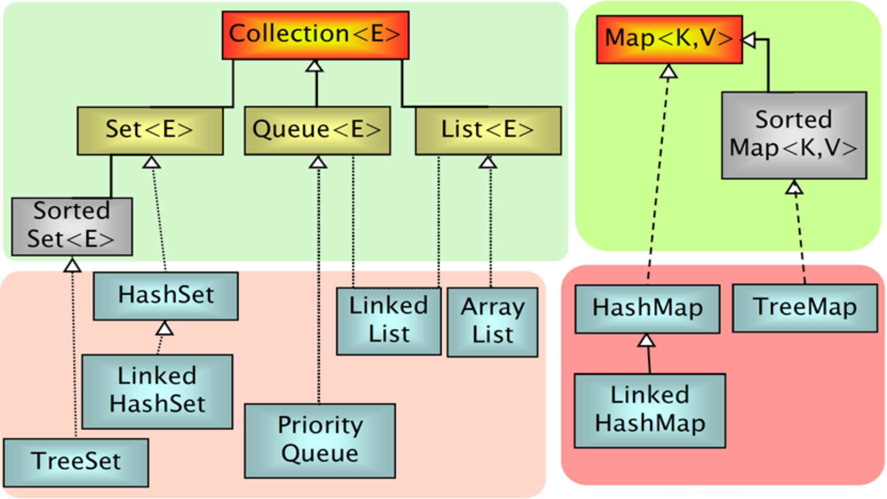


Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) + I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + (D)	= 33

INTERFACES


Iterable<E>
Collection<E>
Set<E>
Queue<E>
List<E>
SortedSet<E>
Map<K,V>
SortedMap<K,V>
Associative containers
Group containers

IMPLEMENTATIONS



INTERNALS

data structure

	Hash table	Resizable array	Balanced tree	Linked list	Hash table Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

interface

classes

Iterable and Iterator Interfaces

- The Iterable interface (java.lang.Iterable) is the root interface of the Java collection framework. Iterable, literally, means that "can be iterated".
- Technically, it means that an Iterator can be returned. Iterable objects (objects implementing the iterable interface) can be used with the for-each loop.

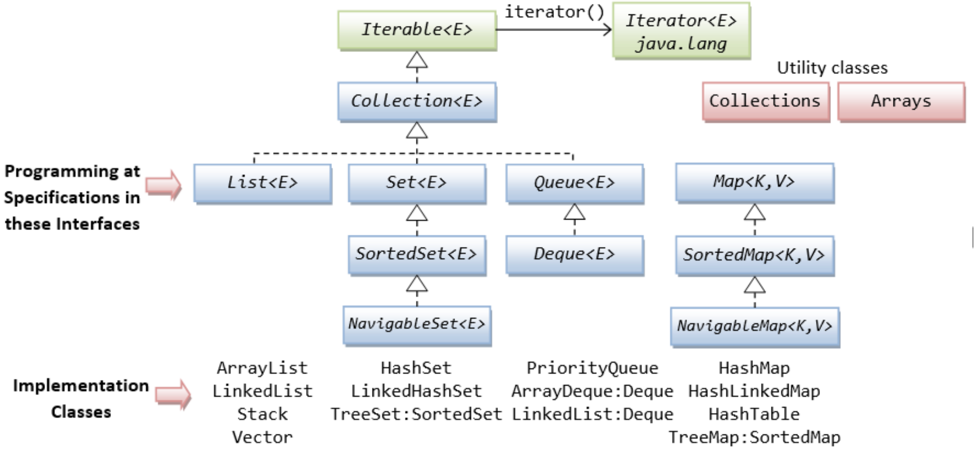
```java
public interface Iterable<T> {
  Iterator<T> iterator();
}

List<Object> list = new ArrayList<Object>();
for (Object obj : list) {
  // do something;
}
```

Iterable and Iterator Interfaces

- The Iterator interface extracts the traversal behaviour of a collection into a separate object called an iterator. This Iterator object can then be used to traverse through all the elements of the associated collection.
- https://refactoring.guru/design-patterns/iterator

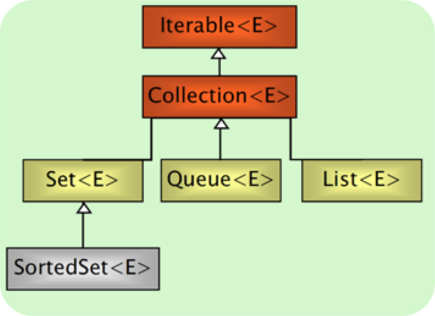
```java
public interface Iterator<T> {
  boolean hasNext();
  T next();
  void remove();
}

ArrayList<Object> list = new ArrayList<Object>();
for (Iterator<Object> it = list.iterator(); it.hasNext();) {
  Object obj = it.next();
  // do something
}
```

- The hierarchy of the interfaces and the commonly-used implementation classes in the Collections Framework is as shown below:



Collections Interface

- Group of elements (references to objects).
- It is not specified whether they are
  - Ordered / not ordered;
  - Duplicated / not duplicated.
- Common constructors
  - Collection();
  - Collection(Collection c).



Collection Interface

```java
// Interface java.util.Collection<E>
// Basic Operations
abstract int size();                 // Returns the number of elements
abstract boolean isEmpty();          // Returns true if there is no element

// "Individual Element" Operations
abstract boolean add(E element);         // Add the given element
abstract boolean remove(Object element);
abstract boolean contains(Object element);

// "Bulk" (mutable) Operations
abstract void clear();                  // Removes all the elements
abstract boolean addAll(Collection<? extends E> c);
abstract boolean removeAll(Collection<?> c);
abstract boolean retainAll(Collection<?> c);

// Array Operations
abstract Object[] toArray();            // Convert to an Object array
```

Presentation Outline

1 Java Collection Framework

2 Lists

3 Sets

4 Queues

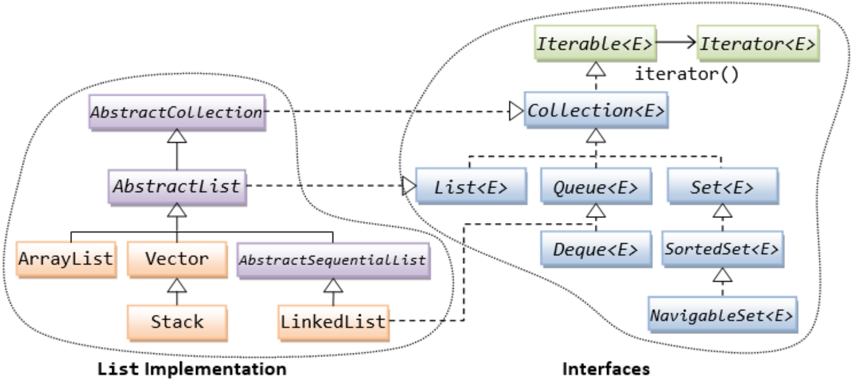
5 Maps

6 Collections and Iterators

7 Algorithms

List Interface

- A List<E> models a resizable linear array, which supports numerical indexed access, with index starts from 0. Elements in a list can be retrieved and inserted at a specific index position based on an int index. It can contain duplicate elements. It can contain null elements. You can search a list, iterate through its elements, and perform operations on a selected range of values in the list.



List additional methods

```java
// Methods inherited from Interface java.lang.Iterable<E>
abstract Iterator<E> iterator();

// Methods inherited from Interface java.util.Collection<E>
abstract int size();
abstract boolean isEmpty();
abstract boolean add(E element);
abstract boolean remove(Object obj);
abstract boolean contains(Object obj);
abstract void clear();

// Interface java.util.List<E>
// Operations at a specified index position
abstract void add(int index, E element);   // add at index
abstract E set(int index, E element);      // replace at index
abstract E get(int index);                 // retrieve at index without remove
abstract E remove(int index);              // remove at index
abstract int indexOf(Object obj);
```

ArrayList

ArrayList is an implementation of the List interface. The list automatically grows if elements exceed initial size. ArrayList has a numeric index.

- Elements are accessed by index.
- Elements can be inserted based on index.
- Elements can be overwritten.

```java
List<Integer> partList = new ArrayList<>();
partList.add(new Integer(1111));
partList.add(new Integer(2222));
partList.add(new Integer(3333));
partList.add(new Integer(4444));    // ArrayList auto grows

System.out.println("First Part: " + partList.get(0));   // First item
partList.add(0, new Integer(5555));   // Insert an item by index
```

Autoboxing and Unboxing

Autoboxing and Unboxing simplifies the syntax. It produces cleaner, easier-to-read code.

```java
public class AutoBox {
  public static void main(String[] args){
    Integer intObject = new Integer(1);
    int intPrimitive = 2;

    Integer tempInteger;
    int tempPrimitive;

    tempInteger = new Integer(intPrimitive);
    tempPrimitive = intObject.intValue();

    tempInteger = intPrimitive;   // Auto box
    tempPrimitive = intObject;    // Auto unbox
  }
}
```

Iterate over an ArrayList

```java
public class GenericArrayList {
  public static void main(String[] args) {
    List<Integer> partList = new ArrayList<>();
    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));

    Iterator<Integer> elements = partList.iterator();
    while (elements.hasNext()) {
      Integer partNumberObject = elements.next();
      int partNumber = partNumberObject.intValue();

      System.out.println("Part number: " + partNumber);
    }
  }
}
```

List Initialization

```java
/* Plain, simple, long */
List<Integer> list = new ArrayList<Integer>();
list.add(14);
list.add(73);
list.add(18);
...

/* More compact version (mutable) */
List<Integer> list = new ArrayList<>(Arrays.asList(14, 73, 18));
List<Integer> list = new ArrayList<>(List.of(14, 73, 18));

/* More compact version (immutable) */
List<Integer> list = List.of(14, 73, 18);
```

List Implementations

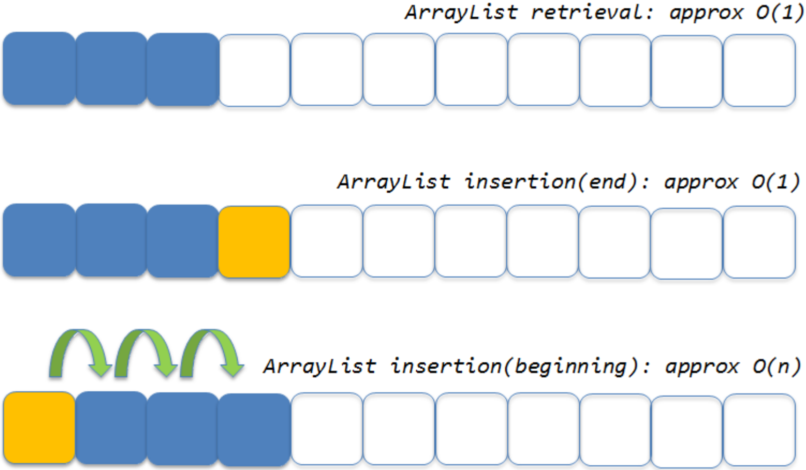
- Decoupling references from actual objects allows to change implementation (and related performance!) by changing a single line of code!

- ArrayList implements List
  - get(index) -> Constant time
  - add(index, obj) -> Linear time

- LinkedList implements List, Queue
  - get(index) -> Linear time
  - add(index, obj) -> Linear time (but more lightweight)

```java
List<Car> garage = new LinkedList<Car>();
// List<Car> garage = new ArrayList<Car>();

garage.add(new Car());
garage.add(new SelfDrivingCar());
garage.add(new SelfDrivingCar());
garage.add(new Car());

for (Car car : garage) {
    car.turnOn();
}
```

List Implementations

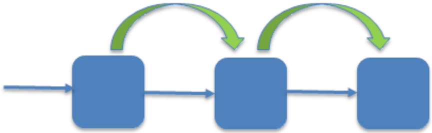


ArrayList retrieval: approx O(1)

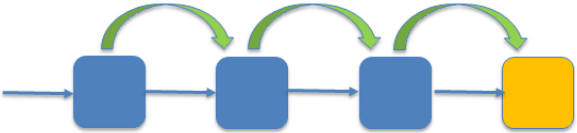
ArrayList insertion(end): approx O(1)

ArrayList insertion(beginning): approx O(n)

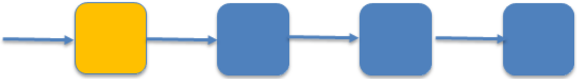
List Implementations



LinkedList retrieval: approx O(n)



LinkedList insertion(end): approx O(n)



LinkedList insertion(beginning): approx O(1)

Presentation Outline

1 Java Collection Framework

2 Lists

3 Sets

4 Queues

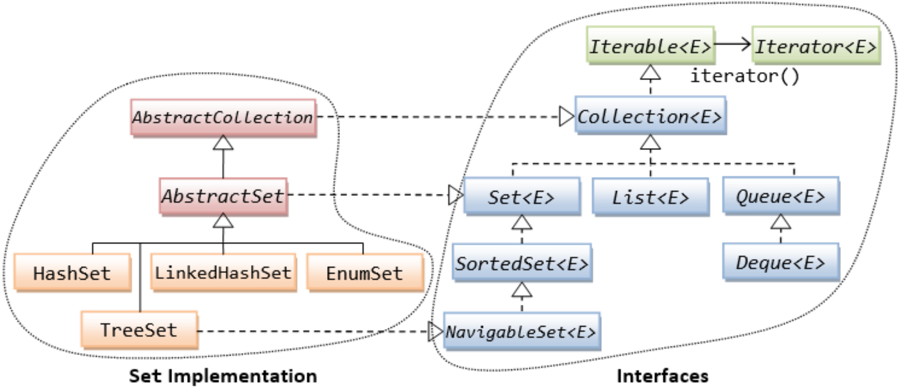
5 Maps

6 Collections and Iterators

7 Algorithms

Set Interface

- The Set<E> interface models a mathematical set, where no duplicate elements are allowed (e.g., playing cards). It may contain a single null element.
- Contains no methods other than those inherited from Collection.



Set Interface

The main difference between List and Set in Java is that List is an ordered collection, which allows duplicates, whereas Set is an unordered collection, which does not allow duplicates.

- A Set is an interface that contains only unique elements.
- A Set has no index.
- Duplicate elements are not allowed.
- You can iterate through elements to access them.
- TreeSet provides sorted implementation.

Set Implementations

- HashSet implements Set
  - Hash tables as internal data structure (fast!)
  - Insertion order not preserved
- LinkedHashSet extends HashSet
  - Insertion order preserved
- TreeSet implements SortedSet (an extension of Set)
  - R-B trees as internal data structure (provide ordering)
  - User definable internal ordering TreeSet(Comparator c)
  - Slow when compared to hash-based implementations

Set Example

```java
ArrayList<String> list = new ArrayList<>(
  List.of("Nicola", "Agata", "Marzia", "Agata")
);
System.out.println(list);
// [Nicola, Agata, Marzia, Agata]

Set<String> hashSet = new HashSet<>(list);
System.out.println(hashSet);
// [Marzia, Nicola, Agata]

Set<String> linkedHashSet = new LinkedHashSet<>(list);
System.out.println(linkedHashSet);
// [Nicola, Agata, Marzia]

Set<String> treeSet = new TreeSet<>(list);
System.out.println(treeSet);
// [Agata, Marzia, Nicola]
```

SET EXAMPLE

```java
public class SetExample {
  public static void main(String[] args){
    Set set = new TreeSet<>();
    set.add("one");
    set.add("two");
    set.add("three");
    set.add("three");  // not added, only unique

    for (String item : set) {
      System.out.println("Item: " + item);
    }
  }
}
```

TreeSet Internal Ordering

- Depending on the constructor used, SortedSet implementations can use different orderings.
- TreeSet()
  - Natural ascending ordering
  - Elements must implement the Comparable Interface.
- TreeSet(Comparator c)
  - Ordering is defined by the Comparator c.

Presentation Outline

1 Java Collection Framework

2 Lists

3 Sets

4 **Queues**

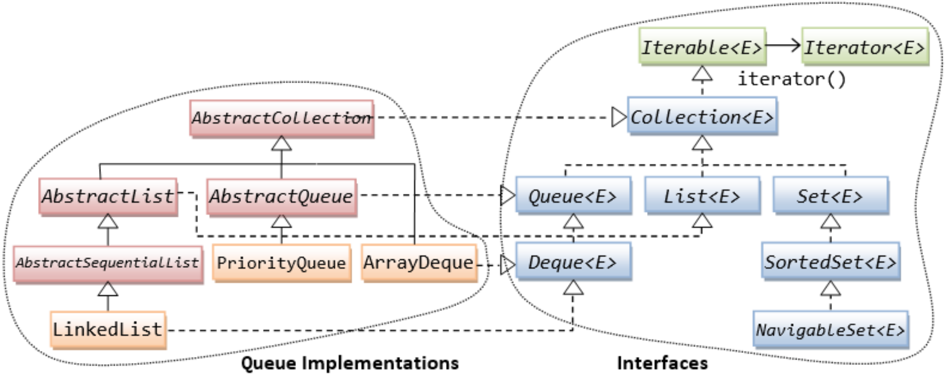
5 Maps

6 Collections and Iterators

7 Algorithms

Queue Interface

- A queue is a collection whose elements are added and removed in a specific order, typically in a first-in-first-out (FIFO) manner. A deque (pronounced "deck") is a double-ended queue that elements can be inserted and removed at both ends (head and tail) of the queue.



Queue additional methods

```java
// Interface java.util.Queue<E>
// Insertion at the end of the queue
abstract boolean add(E e);     // throws IllegalStateException if no space is currently available
abstract boolean offer(E e);   // returns true if the element was added to this queue, else false

// Extract element at the head of the queue
abstract E remove();           // throws NoSuchElementException if this queue is empty
abstract E poll();             // returns the head of this queue, or null if this queue is empty

// Inspection (retrieve the element at the head, but does not remove)
abstract E element();          // throws NoSuchElementException if this queue is empty
abstract E peek();             // returns the head of this queue, or null if this queue is empty
```

Queue Implementations

- LinkedList implements List, Queue
  - Insertion order conserved.
  - Head is the first element of the list
  - FIFO (First-In-First-Out) policy
- PriorityQueue implements Queue
  - Internal ordering policy. Default is natural ascending ordering, if defined. Can be modified by implementing the Comparable interface.

Queue Example

```java
ArrayList<Integer> list = new ArrayList<>(List.of(3, 1, 2));
Queue<Integer> fifo = new LinkedList<Integer>(list);
Queue<Integer> pqueue = new PriorityQueue<Integer>(list);

System.out.println(fifo.peek());     // 3
System.out.println(pqueue.peek());   // 1
```

PriorityQueue or TreeSet?

- Similarities
  - Both provide O(log(N)) time complexity for adding, removing, and searching elements.
  - Both provide elements in sorted order.
- Differences
  - TreeSet is a Set and doesn't allow a duplicate element, while PriorityQueue is a queue and doesn't have such restriction.
  - Another key difference between TreeSet and PriorityQueue is iteration order, though you can access elements from the head in a sorted order e.g. head always give you lowest or highest priority element depending upon your Comparable or Comparator implementation but iterator returned by PriorityQueue doesn't provide any ordering guarantee.

Presentation Outline

1 Java Collection Framework

2 Lists

3 Sets

4 Queues

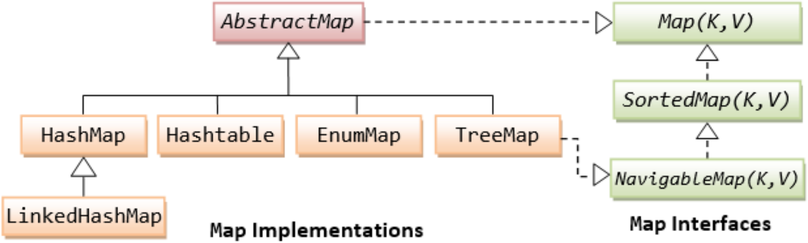
5 Maps

6 Collections and Iterators

7 Algorithms

Map Interface

- A map is a collection of key-value pairs (e.g., name-address, name-phone, isbn-title, word-count). Each key maps to one and only value. Duplicate keys are not allowed, but duplicate values are allowed. Maps are similar to linear arrays, except that an array uses an integer key to index and access its elements; whereas a map uses any arbitrary key (such as Strings or any objects).



Map Interface

```java
// Interface java.util.Map<K,V>
abstract int size();                    // Returns the number of key-value pairs
abstract boolean isEmpty();             // Returns true if this map contain no key-value pair

abstract V get(Object key);             // Returns the value of the specified key
abstract V put(K key, V value);         // Associates the specified value with the specified key

abstract boolean containsKey(Object key);          // Returns true if this map has specified key
abstract boolean containsValue(Object value);      // Returns true if this map has specified value

abstract void clear();                      // Removes all key-value pairs
abstract void V remove(Object key);         // Removes the specified key
```

HashMap

- Get/set takes constant time (without considering collisions).
- Automatic re-allocation when load factor reached.
- Constructor optional arguments
  - load factor (default = .75)
  - initial capacity (default = 16)

Map Example I

```java
Map<String, Integer> map = new HashMap<>();
map.put("Agata", 2);
map.put("Marzia", 3);
map.put("Agata", 4);
map.put("Nicola", 1);

/* More compact version */
Map<String, Integer> map = new HashMap<>(
  Map.of("Agata", 2, "Marzia", 3, "Agata", 4, "Nicola", 1)
);

/* Immutable version */
Map<String, Integer> map =
  Map.of("Agata", 2, "Marzia", 3, "Agata", 4, "Nicola", 1);
```

Map Implementations

- HashMap implements Map
  - Hash tables as internal data structure (fast!).
  - Insertion order not preserved.
- LinkedHashMap extends HashMap
  - Insertion order preserved.
- TreeMap implements SortedMap
  - R-B trees as internal data structure.
  - User definable internal ordering.
  - Slow when compared to hash-based implementations.

```java
Map<Integer, String> src;
src = new HashMap<>();
src.put(77, "Nicola");
src.put(17, "Marzia");
src.put(22, "Agata");
System.out.println(src);

// {17=Marzia, 22=Agata, 77=Nicola}
src = new LinkedHashMap<>();
src.put(77, "Nicola");
src.put(17, "Marzia");
src.put(22, "Agata");
System.out.println(src);
// {77=Nicola, 17=Marzia, 22=Agata}

src = new TreeMap<>();
src.put(77, "Nicola");
src.put(17, "Marzia");
src.put(22, "Agata");
System.out.println(src);
// {17=Marzia, 22=Agata, 77=Nicola}
```

Map Example II

```java
Map<String, Integer> map = new HashMap<String, Integer>();
...
// Looping keys and accessing values
Set<String> keys = map.keySet();
for (String key : keys) {
  System.out.println(key + " -> " + map.get(key));
}

// Contains key
if (map.containsKey(key)) {
  System.out.println(map.get(key));
}

// Looping values
List<Integer> values = map.values();
for (int value : values) {
  System.out.println(value);
}
```

Presentation Outline

1 Java Collection Framework

2 Lists

3 Sets

4 Queues

5 Maps

6 Collections and Iterators

7 Algorithms

Collections and Iterators

- It is unsafe to modify (adding or removing elements) a Collection while iterating over it!.

```java
List<Double> list = new LinkedList<Double>(
  List.of(10.8, 11.1, 13.2, 30.2)
);

int count = 0;
for (double i : list) {
  if (count == 1) {
    list.remove(count);
  }

  if (count == 2) {
    list.add(22.3);
  }

  count++;
} // Run-time error! We modify the list while iterating
```

Iterator and ListIterator Interfaces

- Interface Iterator provides a transparent means for cycling through all elements of a Collection (forward only) and removing elements.

```java
boolean hasNext()
Object next()
void remove()
```

- Interface ListIterator provides a transparent means for cycling through all elements of a Collection (forward and backward) and removing and adding elements.

```java
boolean hasNext()
boolean hasPrevious()
Object next()
Object previous()
void add()
void set()
void remove()
int nextIndex()
int previousIndex()
```

Iterator and ListIterator Interfaces

```java
List<Double> list = new LinkedList<Double>(
    List.of(10.8, 11.1, 13.2, 30.2)
);

int count = 0;
for (Iterator<Double> it = list.iterator(); it.hasNext();) {
    double d = it.next();

    if (count == 1) {
        it.remove();
    }

    count++;
}
```

Iterator and ListIterator Interfaces

```
List<Double> list = new LinkedList<Double>(
  List.of(10.8, 11.1, 13.2, 30.2)
);

int count = 0;
for (Iterator<Double> it = list.iterator(); it.hasNext();) {
  double d = it.next();

  if (count == 1) {
    it.remove();
  }

  if (count == 2) {
    it.add(22.3);
  }

  count++;
}
```

Iterations and Iterators

```java
List<Person> personList = new ArrayList<Person>();

/* C style */
for (int i = 0; i < personList.size(); i++) {
  System.out.println(personList.get(i))
}

/* for-each style */
for (Person person : personList) {
  System.out.println(person);
}

/* iterator style */
for (Iterator<Person> it = personList.iterator(); it.hasNext();) {
  Person person = it.next();
  System.out.println(person);
}

/* while style */
Iterator it = personList.iterator();
while (it.hasNext()) {
  System.out.println((Person)it.next());
}
```

Presentation Outline

1 Java Collection Framework

2 Lists

3 Sets

4 Queues

5 Maps

6 Collections and Iterators

7 Algorithms

- Comparable and Comparator

java.util.Collections

- This class contains various methods for manipulating arrays such as sorting, searching, filling, printing or being viewed as an array.

```java
sort()            // merge sort implementation, n log(n)
binarySearch()    // requires ordered collection
shuffle()         // unsort
reverse()         // requires ordered collection
rotate()          // rotate elements of a given distance
min()             // min in a collection
max()             // max in a collection
```

```java
ArrayList<String> list = new ArrayList<String>(
    List.of("Nicola", "Agata", "Marzia", "Agata")
);

Collections.sort(list);
System.out.println(list);     // [Agata, Agata, Marzia, Nicola]

Collections.reverse(list);
System.out.println(list);     // [Nicola, Marzia, Agata, Agata]

Collections.shuffle(list);
System.out.println(list);     // [Marzia, Agata, Agata, Nicola]

Collections.rotate(list, 1);
System.out.println(list);     // [Nicola, Marzia, Agata, Agata]
```

Algorithms

```java
ArrayList<String> list = new ArrayList<String>(
    List.of("Nicola", "Agata", "Marzia", "Agata")
);

Collections.sort(list);
System.out.println(ist);                        // [Agata, Agata, Marzia, Nicola]

Collections.binarySearch(list, "Nicola");       // 3
Collections.binarySearch(list, "Zuck"));        // -5
```

- Using binarySearch, the list must be sorted. If it is not sorted, the results are undefined.

Presentation Outline

1 Java Collection Framework

2 Lists

3 Sets

4 Queues

5 Maps

6 Collections and Iterators

7 Algorithms

- Comparable and Comparator

Sorting Collections of Objects

- For sorting collections of objects, the Comparable or Comparator interface must be implemented for making objects comparable to each other.
- The Comparable and Comparator Interface are implemented by default in common types in packages java.lang and java.util.

```java
public interface Comparable<T> {
    public int compareTo(T obj);
}

public interface Comparator<T> {
  public int compare(T left, T right);
}
```

The Comparable Interface

- A collection of T can be sorted if T implements Comparable. The compareTo() method compares the object with the object passed as a parameter. Return value must be:

  < 0 if this object precedes obj
  == 0 if this object has the same position as obj
  > 0 if this object follows obj

```java
class Person implements Comparable<Person> {
  protected String firstname;
  protected String lastname;
  protected int age;

  public int compareTo(Person person) {
    // order by lastname
    return lastname.compareTo(person.lastname);
  }
}
```

The Comparable Interface

```java
class Person implements Comparable<Person> {
  protected String firstname;
  protected String lastname;
  protected int age;

  public int compareTo(Person person) {
    // order by lastname
    compare = lastname.compareTo(person.lastname);
    if (compare == 0) {
      // if lastnames are equal, order by firstname
      compare = firstname.compareTo(person.firstname);
    }

    return compare;
  }
}
```

The Comparator Interface

- We can also sort objects using a Comparator<E>.
- Given a class already implementing Comparable<E>, we can sort it using alternative orders using a Comparator<E>.

```java
public class SortByAge implements Comparator<Person> {
  @Override
  public int compare(Person left, Person right) {
    return left.age - right.age;
  }
}

class Person implements Comparable<Person> {
  protected String firstname;
  protected String lastname;
  protected int age;

  public int compareTo(Person person) {
    return lastname.compareTo(person.lastname);
  }
}
```

The Comparator Interface

```java
public static void main(String[] args) {
  ArrayList<Person> list = new ArrayList<Person>();
  list.add(new Person("Mario", "Rossi", 68));
  list.add(new Person("Luca", "Bianchi", 28));
  list.add(new Person("Carlo", "Antoni", 34));

  // Natural ordering (Comparable)
  Collections.sort(list);

  // Special ordering (Comparator)
  Collections.sort(list, new SortByAge());

  // Comparator anonymous class
  Collections.sort(list, new Comparator<Person>() {
    @Override
    public int compare(Person left, Person right) {
      return left.age - right.age;
    }
  });
}
```

Comparator vs Comparable

- The Comparable interface is a good choice to use for defining the default ordering, or in other words, if it's the main way of comparing objects.
- So why use a Comparator if we already have Comparable? There are several reasons why:
  - Sometimes we can't modify the source code of the class whose objects we want to sort, thus making the use of Comparable impossible.
  - Using Comparators allows us to avoid adding additional code to our domain classes.
  - We can define multiple different comparison strategies, which isn't possible when using Comparable.

References

- Allen B. Downey, Chris Mayfield, ***Think Java***, (2016).
- Graham Mitchell, ***Learn Java the Hard Way***, 2nd Edition, (2016).
- Cay S. Horstmann, ***Big Java - Early Objects***, 7e-Wiley, (2019).
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, ***The Java Language Specification - Java SE 8 Edition***, (2015).
- Martin Fowler, ***UML Distilled - A Brief Guide To The Standard Object Modeling Language***, (2004).
- Richard Warburton, ***Object-Oriented vs. Functional Programming - Bridging the Divide Between Opposing Paradigms***, (2016).
- Naftalin, Maurice Wadler, Philip ***Java Generics and Collections***, O'Reilly Media, (2009).
- Eric Freeman, Elisabeth Robson ***Head First Design Patterns - Building Extensible and Maintainable Object***, Oriented Software-O'Reilly Media, (2020).
- Alexander Shvets ***Dive Into Design Patterns***, (2019).

Thank You!