Object-Oriented Programming and Design with Java

HaQT

HUS
VNU UNIVERSITY OF SCIENCE

Table of Contents

Part I Java Basics

Part II Object-Oriented Programming

Part III Collections Framework

1 The Java Collections Framework By Examples

1.1 Introduction to the Collections Framework

Although we can use an *array* as a *container* to store a group of elements of the same type (primitives or objects). 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.

The Java Collection Framework package (java.util) contains:

1. A set of interfaces,
2. Implementation classes, and
3. Algorithms (such as sorting and searching).

Similar Collection Framework is the C++ Standard Template Library (STL).

Prior to JDK 1.2, Java's data structures consist of array, Vector, and Hashtable that were designed in a non-unified way with inconsistent public interfaces. JDK 1.2 introduced the unified *collection framework*, and retrofits the legacy classes (**Vector** and **Hashtable**) to conform to this unified *collection framework*.

JDK 5 introduced *Generics* (which supports passing of types), and many related features (such as auto-boxing/auto-unboxing and for-each loop). The collection

framework is retrofitted to support generics and takes full advantages of these new features.

To understand this chapter, you have to be familiar with:

- Interfaces, abstract methods and their implementations.
- Inheritance and Polymorphism, especially the upcasting and downcasting operations.

You also need to be familar with these concepts introduced in JDK 5:

- Auto-Boxing and Auto-Unboxing between primitive types and their wrapper classes.
- The enhance fo-each loop.
- Generics

You need to refer to the JDK API specification while reading this chapter. The classes and interfaces for the *Collection Framework* are kept in package java.util.

1.2 Generic Collections Framework (JDK 5) by Examples

1.2.1 Example 1: List<String> (List of Strings) Implemented by ArrayList

The **java.util.List**<**E**> interface is the most commonly used data structure of the Collection Framework, which models a resizable (dynamically-allocated) array supporting numerical index access. The **java.util.ArrayList**<**E**> class is the most commonly used implementation of **List**<**E**>.

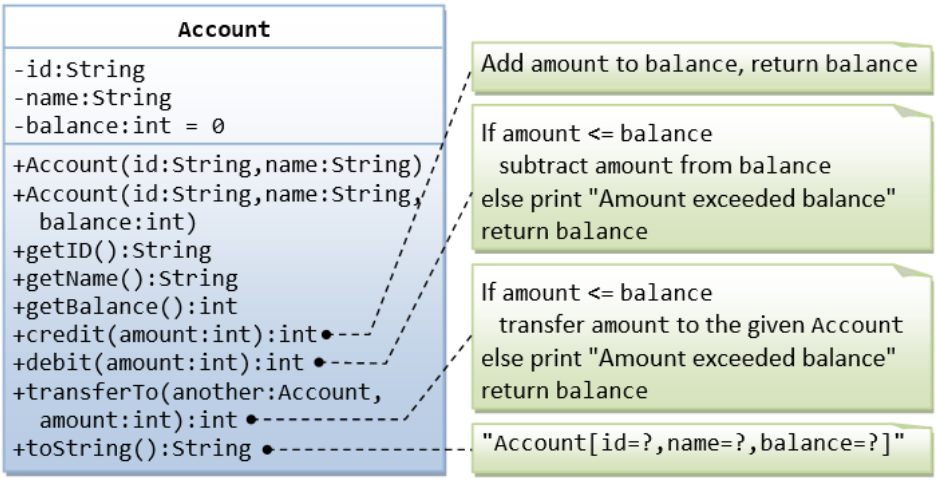
The <**E**> indicates that the interfaces are *generic* in design. When you construct an instance of these generic types, you need to provide the specific type of the objects contained in these collection, e.g., <*String*>, <**Integer**>. This allows the compiler to perform type-checking when elements are added into the collection at compile-time to ensure type-safety at runtime.

```
1 import java.util.List;
  import java.util.ArrayList;
3 import java.util.Iterator;

5 /**
   * TestJ5ListOfString.java
```

```
 * JDK 5 List<String> with Generics, implemented by an ArrayList
 */
public class TestJ5ListOfString {
   public static void main(String[] args) {
      // Declare a List<String> (List of String) implemented by an ArrayList<String>
      // The compiler is informed about the type of objects in the collection via <>.
      // It performs type checking when elements are added into the collection
      List<String> coffeeList = new ArrayList<>();
      // JDK 7 supports type inference on instantiation.
      // ArrayList<String> can be written as ArrayList<>,
      //  with type inferred from List<String>
      coffeeList.add("espresso");
      coffeeList.add(" latte ");
      coffeeList.add("cappuccino");
      System.out.println(coffeeList);   // [espresso, latte, cappuccino]

      // (1) Use an Iterator<String> to traverse through all the elements
      Iterator<String> iter = coffeeList.iterator();
      while (iter.hasNext()) {
         String str = iter.next();
         System.out.println(str);
      }
      // espresso
      //  latte
      // cappuccino

      // (2) Use an enhanced for-each loop (JDK 5) to traverse through the list
      for (String str : coffeeList) {
         System.out.println(str.toUpperCase());
      }
      // ESPRESSO
      // LATTE
      // CAPPUCCINO

      // A List supports numerical index access, where index begins at 0
      for (int i = 0; i < coffeeList.size(); ++i) {
         System.out.println(coffeeList.get(i).substring(0, 3));
      }
      // esp
      //  lat
      // cap

      // Compiler checks the type of the elements added or retrieved
      // coffeeList.add(new Integer(1234));
      // compilation error: incompatible types: Integer cannot be converted to String
      // Integer intObj = coffeeList.get(0);
      // compilation error: incompatible types: String cannot be converted to Integer
   }
}
```

Dissecting the Program

- Line 1-3 imports the collection framework classes and interfaces reside in the java.util package.

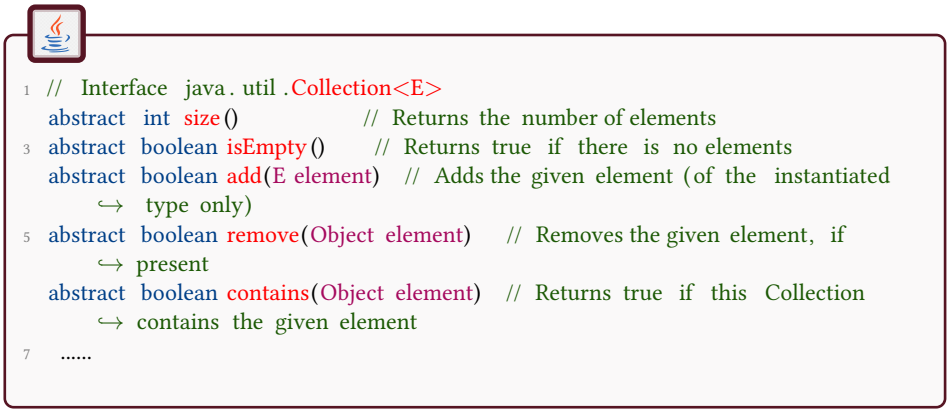


- The class hierarchy of the **ArrayList<E>** is shown above. We observe that **ArrayList<E>** implements **List<E>**, **Collection<E>** and **Iterable<E>** interfaces. The **Collection<E>** and **Iterable<E>** interfaces define the common behaviors of all the collection implementations.
- The interfaces/classes are designed (by the class designer) to take a generics type **E**. To construct an instance of an **ArrayList<E>**, we need to provide the actual type for generic type **E**. In this example, we pass the actual type *String* for the generic type **E**.
- In line 11, we construct an **ArrayList<String>** instance, and upcast it to the **List<String>** interface. This is possible as the **ArrayList<String>** is a subtype of **List<String>**. Remember that a good program operates on the specifications instead of an actual implementation. The Collection Framework provides a set of interfaces so that you can program on these interfaces instead of the actual implementation.
- JDK 7 supports type inference on instantiation to simplify the codes:

```
1 // Prior to JDK 7, you need to write
  List<String> coffeeLst = new ArrayList<String>();
3
  // JDK 7 can infer on type from the type of the assigned variable
5 List<String> coffeeLst = new ArrayList<>();
```

- Interface **Collection** defines how to add and remove an element into the collection. Interface Iterable defines a mechanism to iterate or traverse through all the elements of a collection.
- The super-interface **Collection<E>** interface defines the common behaviors expected from a **Collection<E>** object (such as getting the size, adding and removing element). Instead of using the interface **Collection<E>** directly, it is more common to use one of its specialized sub-interfaces: List (an ordered list supporting numerical indexed access), **Set** (models mathematical set with no duplicate elements) or **Queue** (FIFO, priority queues).

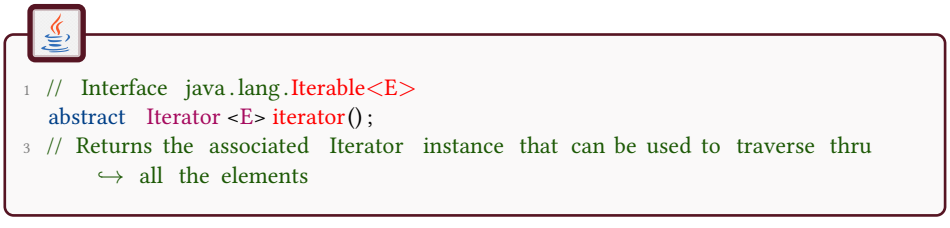
  The super-interface **Collection<E>** declares these abstract methods, implemented in concrete class **ArrayList<E>**:



```
1  // Interface java.util.Collection<E>
   abstract int size()              // Returns the number of elements
3  abstract boolean isEmpty()       // Returns true if there is no elements
   abstract boolean add(E element)  // Adds the given element (of the instantiated
       ↪ type only)
5  abstract boolean remove(Object element)    // Removes the given element, if
       ↪ present
   abstract boolean contains(Object element)  // Returns true if this Collection
       ↪ contains the given element
7  .....
```

  Line 14 - 16 adds elements (of the instantiated actual type *String*) to the **Collection**.

- There are a few ways of traversing through the elements of a **Collection**:
  - Via an associated **Iterator<E>** (Lines 21 - 25)
  - Use the new for-each loop introduced in JDK 5 (Line 31)
  - Use aggregate operations on **Stream** introduced in JDK 8. See "Example 4".
- The super-interface **Iterable<E>** defines a mechanism to iterate (or traverse) through all the elements of a **Collection<E>** object via a so-called **Iterator<E>** object. The **Iterable<E>** interface declares one abstract method to retrieve the **Iterator<E>** object associated with the **Collection<E>**.



```
1  // Interface java.lang.Iterable<E>
   abstract Iterator<E> iterator();
3  // Returns the associated Iterator instance that can be used to traverse thru
       ↪ all the elements
```

- The **Iterator<E>** interface declares the following abstract methods for traversing through the **Collection<E>**.

```
// Interface java.util.Iterator<E>
abstract boolean hasNext()    // Returns true if it has more elements
abstract E next()             // Returns the next element (of the actual type)
```

Lines 20 - 24 retrieve the **Iterator<String>** associated with this **ArrayList<String>**, and use a while-loop to iterate through all the elements of this **ArrayList<String>**. Take note that you need to specify the actual type.

- Line 30 uses the new for-each loop introduced in JDK 5 to iterate through all the elements of a **Collection**.
- **List** supports numerical index access via *get(index)* method, with index begins from 0, shown in Lines 36 - 38.
- With the use of generics, the compiler checks the type of elements added or retrieved and issue compilation error "incompatible type", as shown in lines 44 - 47. This is known as compiled-time type-safe.

1.2.2 Example 2: List<Integer> with Auto-Boxing/Auto-Unboxing

Collection can hold only objects, not primitives (such as *int*, *double*). JDK 5 introduces auto-boxing/unboxing to simplify the conversion between primitives and their wrapper classes (such as *int*/**Integer**, *double*/**Double**, etc.)

```
import java.util.List;
import java.util.ArrayList;

/**
 * TestJ5ListOfPrimitives.java
 * JDK 5 List of primitive wrapper objects with auto-boxing/auto-unboxing
 */
public class TestJ5ListOfPrimitives {
   public static void main(String[] args) {
      // Collection holds only objects, not primitives
      // JDK 5 supports auto-boxing/auto-unboxing of primitives to their wrapper class
      List<Integer> numLst = new ArrayList<>();   // List of Integer object
                                                  // (not int)
      numLst.add(111);   // auto-box primitive int to Integer object
      numLst.add(222);
      System.out.println(numLst);     // [111, 222]
```

```
17      int firstNum = numLst.get(0);   // auto-unbox Integer object to int primitive
        System.out.println(firstNum);   // 111
19
        // numLst.add(33.33);              // Only accept Integer or int (auto-box)
21      // compilation error: incompatible types: double cannot be converted to Integer
    }
23  }
```

Dissecting the Program

- In this example, we pass actual type Integer for the generic type **E**. In Line 9, we construct an **ArrayList<Integer>** and upcast to **List<Integer>**.
- In lines 10-11, we pass int value into the *add()*. The *int* value is auto-boxed into an **Integer** object, and added into the **List<Integer>**.
- In line 13, the *get()* returns an Integer object, which is auto-unboxed to an int value.

1.2.3 Example 3: Set<E> Implemented by HashSet

This example shows how to create a **Collection** of an user-defined objects.

The **Set<E>** interface models an unordered mathematical set without duplicate elements. **HashSet<E>** is the most common implementation of **Set<E>**.

We define a **Person** class with two private variables name and age, as follows:

```
1   /**
     * Person.java
3    * The Person class
     */
5   public class Person {
      private String name;  // private instance variables
7     private int age;

9     public Person(String name, int age) {   // constructor
        this.name = name; this.age = age;
11    }

13    public String getName() {   // getter for name
        return name;
15    }

17    public int getAge() {   // getter for age
        return age;
```

```
19    }

21    public String toString() {   // describe itself
        return name + "(" + age + ")";
23    }

25    public void sayHello() {   // for testing
        System.out.println(name + " says hello");
27    }

29    // Compare two Person objects by name strings, case insensitive
      @Override
31    public boolean equals(Object o) {
        return (o != null)
33        && (o instanceof Person)
          && this.name.equalsIgnoreCase(((Person)o).name);
35    }

37    // To be consistent with equals()
      // Two objects which are equal shall have the same hashcode.
39    @Override
      public int hashCode() {
41      return this.name.toLowerCase().hashCode();
      }
43  }
```

We shall test a **Set<Person>** is follows:

```
1   import java.util.Set;
    import java.util.HashSet;
3
    /**
5    * TestJ5SetOfPerson.java
     * JDK 5 Set of Person objects
7    */
    public class TestJ5SetOfPerson {
9     public static void main(String[] args) {
        Set<Person> personSet = new HashSet<>();
11      personSet.add(new Person("Peter", 21));
        personSet.add(new Person("Paul", 18));
13      personSet.add(new Person("John", 60));
        System.out.println(personSet);   // [John(60), Peter(21), Paul(18)]
15      // Unlike List, a Set is NOT ordered

17      for (Person p : personSet) {
          p.sayHello();
19      }
```

```
        // John says hello
21      // Peter says hello
        // Paul says hello
23
        // Set does not support duplicate elements
25      System.out.println(personSet.add(new Person("Peter", 21)));   // false
    }
27  }
```

Dissecting the Program

- We define our custom objects called **Person** in "Person.java".
- In this example, we pass actual type **Person** for the generic type **E**. In Line 7, we construct an instance of **HashSet<Person>**, and upcast to **Set<Person>**.
- Take note that the elements in a **Set** are not ordered, as shown in the output of Line 11.
- A **Set** does not duplicate elements, as shown in the output of Line 20. To compare two **Peson** objects, we override the $equal()$ and $hashCode()$ methods in "Person.java".

1.2.4 Example 4: JDK 8 Collection, Stream and Functional Programming (Preview)

JDK 8 greatly enhances the Collection Framework with the introduction of Stream API to support functional programming.

```
1   import java.util.List;
    import java.util.function.Predicate;
3   import java.util.stream.Collectors;

5   // JDK 8 Collection, Stream and Functional Programming
    public class J8StreamOfPersonTest {
7     public static void main(String[] args) {
        List<Person> personLst = List.of(   // JDK 9 instantiation of an unmodifiable List
9         new Person("Peter", 21),
          new Person("Paul", 18),
11        new Person("John", 60)
        );
13      System.out.println(personLst);   // [Peter(21), Paul(18), John(60)]

15      // Use a Predicate to filter the elements
        Predicate<Person> adult = p -> p.getAge() >= 21;
```

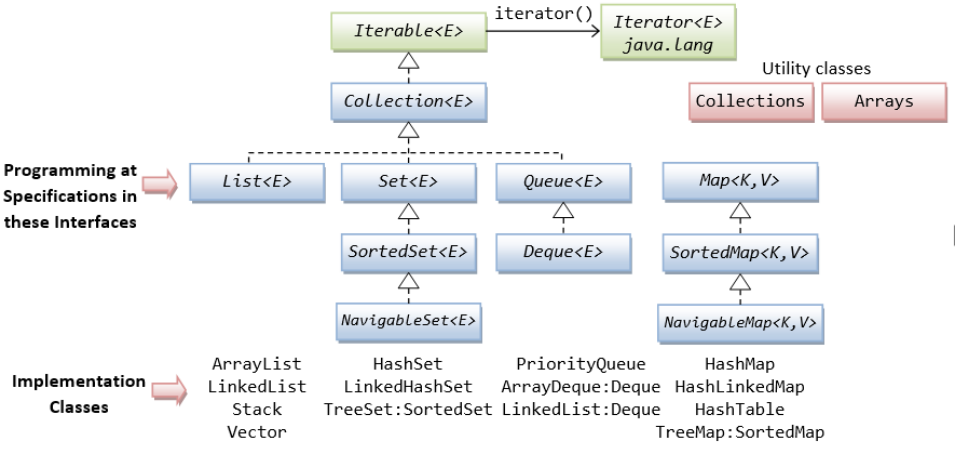
```
17
        // All adults say hello (filter-reduce(foreach))
19      personLst.stream()
          .filter(adult)
21        .forEach(Person::sayHello);
        // Peter says hello
23      // John says hello

25      // Use a parallel stream (reduce(foreach))
        personLst.parallelStream()
27        .forEach(p -> System.out.println(p.getName()));
        // Paul
29      // Peter
        // John
31
        // Get the average age of all adults (filter-map-reduce(aggregate))
33      double aveAgeAdults = personLst.stream()
          .filter(adult)
35        .mapToInt(Person::getAge)
          .average()
37        .getAsDouble();
        System.out.println(aveAgeAdults);
39      // 40.5

41      // Collect the sum of ages (reduce(aggregate))
        int sumAges = personLst.stream()
43        .collect(Collectors.summingInt(Person::getAge));
        System.out.println(sumAges);
45      // 99

47      // Collect the names starting with 'P' (filter-map-reduce(collect))
        List<String> nameStartWithP = personLst.stream()
49        .filter(p -> p.getName().charAt(0) == 'P')
          .map(Person::getName)
51        .collect(Collectors.toList());
        System.out.println(nameStartWithP);
53      // [Peter, Paul]
    }
55  }
```

1.3 Pre-JDK 5 vs. JDK 5 Collection Framework

JDK 5 introduces *Generics* to support parameterized type and retrofitted the *Collections Framework*.

1.4 The Collection Interfaces

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



- The base interface is **Collection**<**E**> (which is a subtype of **Iterable**<**E**>), that defines the common behaviors.
- We hardly program at the **Collection**<**E**>, but one of its specialized subtypes. The commonly-used subtypes of are **List**<**E**> (ordered elements supporting index access), **Set**<**E**> (unordered and non-duplicate) and **Queue**<**E**> (FIFO and priority queues).
- The popular implementation classes for **List**<**E**> are **ArrayList**<**E**> and **LinkedList**<**E**>; for Set are **HashSet**<**E**> and **TreeSet**<**E**>; for **Quene** is **PriorityQueue**<**E**>.
- The **Iterator**<**E**> is used to traverse (or iterate) through each element of a **Collection**<**E**>.
- **Map**<**K, V**> (or associative array), which supports key-value pair, is NOT a subtype of **Collection**<**E**>. The popular implementation class is **HashMap**<**K, V**>.
- The **Collections** (like **Arrays**) is a utility class, which contains static methods to support algorihtms like searching and sorting.

1.4.1 Iterable<E>/Iterator<E> Interfaces and for-each Loop

There are three ways to traverse through all the elements of a **Collection**:

1. Via the associated **Iterator**<**E**> object retrieved from the super-type **Iterable**<**E**>

2. Using the for-each loop (introduced in JDK 5)
3. Via the Stream API (introduced in JDK 8) (to be discussed in "Collection Framework, Part 2")

The Iterable<E> Interface

The **java.lang.Iterable<E>** interface, which takes a generic type <**E**> and read as **Iterable** of elements of type **E**, declares one abstract method called *iterator()* to retrieve the **Iterator<E>** object associated with the **Collection<E>** object. This **Iterator<E>** object can then be used to traverse through all the elements of the associated collection.

```
1 // Interface java.lang.Iterable<E>
  abstract Iterator<E> iterator();
3 // Returns the associated Iterator instance that can be used to
  //  traverse thru all the elements of the collection
```

All implementations of the **Collection** (e.g., **ArrayList**, **LinkedList**, **Vector**) must implement this method, which returns an object that implements **Iterator** interface, for traversal of the **Collection**.

The Iterator<E> Interface

The **Iterator<E>** interface, declares the following three abstract methods:

```
  // Interface java.util.Iterator<E>
2 abstract boolean hasNext()  // Returns true if it has more elements
  abstract E next()           // Returns the next element of generic type E
4 abstract void remove()      // Removes the last element returned by the iterator
```

You can use a while-loop to iterate through the elements with the **Iterator** as follows:

```
  List<String> list = new ArrayList<>();    // JDK 7 type inference
2 list.add("alpha");
  list.add("beta");
4 list.add("charlie");

6 // (1) Using the associated Iterator<E> to traverse through all elements
```

```
// Retrieve the Iterator associated with this List via the iterator() method
Iterator<String> iter = list.iterator();
// Transverse thru this List via the Iterator
while (iter.hasNext()) {
   // Retrieve each element and process
   String str = iter.next();
   System.out.println(str);
}

// (2) Using the for-each loop (JDK 5) to traverse through all elements
// JDK 5 introduces a for-each loop to simplify the above
for (str : list) {
   System.out.println(str);
}
```

The for-each Loop

JDK 5 introduces a new for-each loop to simplify traversal of a **Collection**, as shown in the above code.

for-each Loop vs. Iterator

The for-loop provides a convenience way to traverse through a collection of elements. But it hides the **Iterator**, hence, you CANNOT remove (via *Iterator.remove()*) or replace the elements.

On the other hand, as the loop variable receives a "cloned" copy of the object reference, the enhanced for-loop can be used to modify "mutable" elements (such as **StringBuilder**) via the "cloned" object references, but it cannot modify "immutable" objects (such as *String* and primitive wrapper classes) as new references are created.

Example: Using Enhanced for-each Loop on Collection of "Mutable" Objects (such as StringBuilder)

```
import java.util.List;
import java.util.ArrayList;

/**
 * TestForEachMutable.java
 * The test driver class to test enhanced for-each loop on Collection
 */
public class TestForEachMutable {
   public static void main(String[] args) {
      List<StringBuilder> list = new ArrayList<>();
      list.add(new StringBuilder("alpha"));
```

```
12        list.add(new StringBuilder("beta"));
          list.add(new StringBuilder("charlie"));
14        System.out.println(lst);   // [alpha, beta, charlie]

16        for (StringBuilder sb : list) {
            sb.append("123");          // can modify "mutable" objects
18        }
          System.out.println(lst);   // [alpha123, beta123, charlie123]
20    }
  }
```

Example: Using Enhanced for-each loop on Collection of "Immutable" Objects (such as String)

```
1  import java.util.List;
   import java.util.ArrayList;
3
   /**
5   * TestForEachImmutable.java
    * The test driver class to test enhanced for-each loop on Collection
7   */
   public class TestForEachImmutable {
9    public static void main(String[] args) {
       List<String> list = new ArrayList<>();
11      list.add("alpha");
        list.add("beta");
13      list.add("charlie");
        System.out.println(list);   // [alpha, beta, charlie]
15
        for (String str : list) {
17        str += "change!";          // cannot modify "immutable" objects
        }
19      System.out.println(list);   // [alpha, beta, charlie]
    }
21 }
```

1.4.2 Collection<E> Interface

The **Collection**<**E**>, which takes a generic type **E** and read as Collection of element of type **E**, is the root interface of the Collection Framework. It defines the common behaviors expected of all classes, such as how to add or remove an element, via the following abstract methods:

```
// 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)      // Removes the given element, if present
abstract boolean contains(Object element)    // Returns true if this Collection
    ↪ contains the given element

// "Bulk" (mutable) Operations
abstract void clear()                        // Removes all the elements
abstract boolean addAll(Collection<? extends E> c)   // Another Collection of E or
    ↪ E's subtypes
abstract boolean containsAll(Collection<?> c)  // Another Collection of any types
abstract boolean removeAll(Collection<?> c)
abstract boolean retainAll(Collection<?> c)

// Comparison – Objects that are equal shall have the same hashCode
abstract boolean equals(Object o)
abstract int hashCode()

// Array Operations
abstract Object[] toArray()          // Convert to an Object array
abstract <T> T[] toArray(T[] a)      // Convert to an array of the given type T
```

Take note that many of these operations are mutable, i.e., they modify the **Collection** object. (In the Functional Programming introduced in JDK 8, operations are immutable and should not modify the source collection.)

Collection of Primitives?

A **Collection<E>** can only contain objects, not primitives (such as int or double). Primitive values are to be wrapped into objects (via the respective wrapper classes such as Integer and Double). JDK 5 introduces auto-boxing and auto-unboxing to simplify the wrapping and unwrapping processes. Read "Auto-Boxing and Unboxing" section for example.

1.4.3 List<E>, Set<E> and Queue<E>: Specialized Sub-Interfaces of Collection<E>

In practice, we typically program on one of the specialized sub-interfaces of the **Collection<E>** interface: **List<E>**, **Set<E>**, or **Queue<E>**.

- **List**<**E**>: models a resizable linear array, which allows numerical indexed access, with index starting from 0. **List**<**E**> can contain duplicate elements. Implementations of **List**<**E**> include **ArrayList**<**E**>, **LinkedList**<**E**>, **Vector**<**E**> and **Stack**<**E**>.
- **Set**<**E**>: models a mathematical set, where no duplicate elements are allowed. Implementations of **Set**<**E**> include **HashSet**<**E**> and **LinkedHashSet**<**E**>. The sub-interface **SortedSet**<**E**> models an ordered and sorted set of elements, implemented by **TreeSet**<**E**>.
- **Queue**<**E**>: models queues such as First-in-First-out (FIFO) queue and priority queue. It sub-interface **Deque**<**E**> models queues that can be operated on both ends. Implementations include **PriorityQueue**<**E**>, **ArrayDeque**<**E**> and **LinkedList**<**E**>.

The details of these sub-interfaces and implementations will be covered later in the implementation section.

1.4.4 Map<K, V> Interface

In Java, a **Map** (also known as associative array) contains a collection of key-value pairs. It is similar to List and array. But instead of an numerical key 0, 1, 2, ..., a **Map**'s key could be any arbitrary objects.

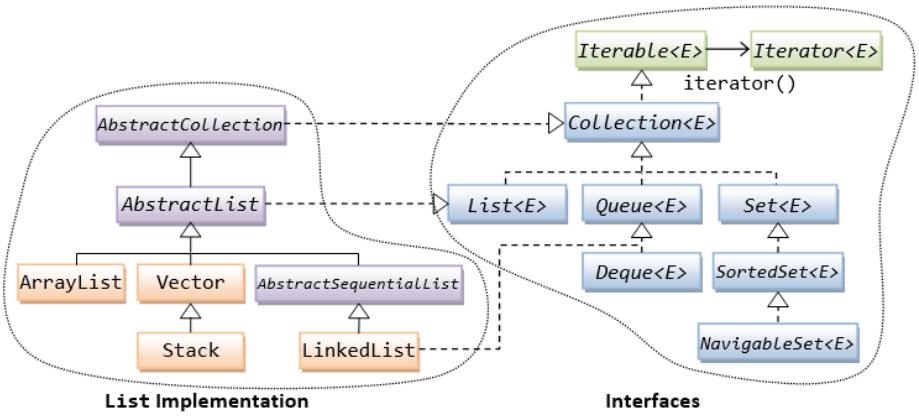
The interface **Map**<**K, V**>, which takes two generic types **K** and **V** (read as Map of key type **K** and value type **V**), is used as a collection of "key-value pairs". No duplicate key is allowed. Implementations include **HashMap**<**K, V**>, **Hashtable**<**K, V**> and **LinkedHashMap**<**K, V**>. Its sub-interface **SortedMap**<**K, V**> models an ordered and sorted map, based on its key, implemented in **TreeMap**<**K, V**>.

Take note that **Map**<**K, V**> is not a sub-interface of **Collection**<**E**>, as it involves a pair of objects for each element. The details will be covered later.

1.5 List Interfaces, Implementations and Algorithms

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 is the most commonly-used data structure, as a resizable array.



The **List<E>** interface declares the following abstract methods, in additional to its super-interfaces. Since List has a positional index. Operation such as *add()*, *remove()*, *set()* can be applied to an element at a specified index position.

```
// 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)
abstract int lastIndexOf(Object obj)

// Operations on a range fromIndex (inclusive) toIndex (exclusive)
abstract List<E> subList(int fromIndex, int toIndex)
```

The abstract superclass **AbstractList** provides implementations to many of the abstract methods declared in List and its supertypes **Collection** and **Iterable**. However, some methods such as *get(int index)* remains abstract. These methods are implemented by the concrete subclasses such as **ArrayList** and **Vector**.

[TODO] Example

1.5.1 ArrayList<E> and Vector<E>: Implementation Classes for List<E>

ArrayList<**E**> is the best all-around implementation of the **List**<**E**> interface. Many useful methods are already implemented in **AbstractList** but overridden for efficiency in **ArrayList** (e.g., $add()$, $remove()$, $set()$ etc.).

Vector<**E**> is a legacy class (since JDK 1.0), which is retrofitted to conform to the Collection Framework (in JDK 1.2). Vector is a synchronized thread-safe implementation of the List interface. It also contains additional legacy methods (e.g., $addElement()$, $removeElement()$, $setElement()$, $elementAt()$, $firstElement()$, $lastElement()$, $insertElementAt()$). There is no reason to use these legacy methods - other than to maintain backward compatibility.

ArrayList is not synchronized. The integrity of **ArrayList** instances is not guaranteed under multithreading. Instead, it is the programmer's responsibility to ensure synchronization. On the other hand, Vector is synchronized internally. Read "Synchronized Collection" if you are dealing with multi-threads.

Java Performance Tuning Tip: Synchronization involves overheads. Hence, if synchronization is not an issue, you should use **ArrayList** instead of **Vector** for better performance.

[TODO] Example

1.5.2 Stack<E>: Implementation Class for List<E>

Stack<**E**> is a last-in-first-out queue (LIFO) of elements. **Stack** extends **Vector**, which is a synchronized resizable array, with five additional methods:

```
// Class java.util.Stack<E>
E push(E element)   // pushes the specified element onto the top of the stack
E pop()             // removes and returns the element at the top of the stack
E peek()            // returns the element at the top of stack without removing

boolean empty()           // tests if this stack is empty
int search(Object obj)    // returns the distance of the specified object
   // from the top of stack (distance of 1 for TOS), or -1 if not found
```

[TODO] Example

1.5.3 LinkedList<E>: Implementation Class for List<E>

LinkedList<**E**> is a double-linked list implementation of the **List**<**E**> interface, which is efficient for insertion and deletion of elements, in the expense of more complex structure.

LinkedList<**E**> also implements **Queue**<**E**> and **Deque**<**E**> interfaces, and can be processed from both ends of the queue. It can serve as FIFO or LIFO queue.

[TODO] Example

1.5.4 Converting a List to an Array: toArray()

The super-interface **Collection**<**E**> defines a method called $toArray()$ to create a fixed-length array based on this list. The returned array is free for modification.

```
// Interface java.util.Collection<E>
abstract Object[] toArray()          // Object[] version
abstract <T> T[] toArray(T[] a)      // Generic type version
```

Example - List to array

```
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;

/**
 * TestListToArray.java
 * The test driver class to test creating an array based on a list
 */
public class TestListToArray {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>();
      list.add("alpha");
      list.add("beta");
      list.add("charlie");

      // Use the Object[] version
      Object[] strArray1 = list.toArray();
      System.out.println(Arrays.toString(strArray1));  // [alpha, beta, charlie]

      // Use the generic type version - Need to specify the type in the argument
      String[] strArray2 = list.toArray(new String[list.size()]);   // pass a String
          ↪ array of the same size
      System.out.println(strArray2.length);   // 3
```

```
23        strArray2[0] = "delta";     // modify the returned array
          System.out.println(Arrays.toString(strArray2)); // [delta, beta, charlie]
25        System.out.println(lst);    // [alpha, beta, charlie] (no change in the original list)
      }
27  }
```

1.5.5 Using an Array as a List: Arrays.asList()

The utility class **java.util.Arrays** provides a static method **Arrays**.*asList*() to convert an array into a **List**<**T**>. However, change to the list write-thru the array and vice versa. Take note that the name of the method is *asList()* and not *toList()*.

```
1  // Returns a fixed-size list backed by the specified array.
   // Change to the list write-thru to the array.
3  public static <T> List<T> asList(T[] a)
```

Example - Array as List

```
1  import java.util.List;
   import java.util.ArrayList;
3  import java.util.Arrays;

5  /**
    * TestArrayAsList.java
7   * The test driver class to test converting an array to list
    */
9  public class TestArrayAsList {
     public static void main(String[] args) {
11     String[] strs = {"alpha", "beta", "charlie"};
       System.out.println(Arrays.toString(strs)); // [alpha, beta, charlie]
13
       List<String> lst = Arrays.asList(strs);
15     System.out.println(lst);    // [alpha, beta, charlie]

17     // Changes in array or list write thru
       strs[0] += "88";
19     lst.set(2, lst.get(2) + "99");
       System.out.println(Arrays.toString(strs)); // [alpha88, beta, charlie99]
21     System.out.println(lst);    // [alpha88, beta, charlie99]

23     // Initialize a list using an array
```

```
        List<Integer> lstInt = Arrays.asList(22, 44, 11, 33);
25      System.out.println(lstInt);  // [22, 44, 11, 33]
    }
27 }
```

1.5.6 Comparison of ArrayList, Vector, LinkedList and Stack

[TODO] Example on benchmarking **ArrayList**, **Vector**, **LinkedList**, and **Stack**.

1.5.7 List's Algorithms

The utility class **java.util.Collections** provides many useful algorithms for collection. Some work for any Collections; while many work for **Lists** (with numerical index) only.

Mutating Operators

```
1  /* Utility Class java.util.Collections */
   // Swaps the elements at the specified indexes
3  static void swap(List<?> lst, int i, int j)

5  // Randomly permutes the List
   static void shuffle(List<?> lst)
7
   // Randomly permutes the List using the specified source or randomness
9  static void shuffle(List<?> lst, Ramdon rnd)

11 // Rotates the elements by the specified distance
   static void rotate(List<?> lst, int distnace)
13
   // Reverses the order of elements
15 static void reverse(List<?> lst)

17 // Replaces all elements with the specified object
   static <T> void fill(List<? super T>, T obj)
19
   // Copies all elements from src to dest
21 static <T> void copy(List<? super T> dest, List<? extends T> src)

23 // Replaces all occurrences
   static <T> boolean replaceAll(List<T> lst, T oldVal, T newVal)
```

[TODO] Example

Sub-List (Range-View) Operations

The **List**<**E**> supports range-view operation via $subList()$ as follows. The returned List is backup by the given **List**, so change in the returned List are reflected in the original **List**.

```
// Interface java.util.List<E>
List<E> subList(int fromIdx, int toIdx)
```

The Utility class **Collections** supports these sub-list operations:

```
// Utility Class java.util.Collections
static int indexOfSubList(List<?> src, List<?> target)
static int lastIndexOfSubList(List<?> src, List<?> target)
```

For example,

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

/**
 * TestSubList.java
 */
public class TestSubList {
   public static void main(String[] args) {
      List<Integer> lst = new ArrayList<>();
      for (int i = 0; i < 10; ++i) lst.add(i * 10);
      System.out.println(lst); // [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

      lst.subList(3, 6).clear(); // Remove the sublist
      System.out.println(lst);   // [0, 10, 20, 60, 70, 80, 90]

      System.out.println(lst.subList(2, 5).indexOf(60)); // 1 (of the subList)

      List<Integer> lst2 = List.of(20, 60, 70); // JDK 9
      System.out.println(Collections.indexOfSubList(lst, lst2));     // 2
      System.out.println(Collections.lastIndexOfSubList(lst, lst2)); // 2
   }
}
```

Searching, Sorting and Ordering

The utility class **Collections** provides these static methods for searching, sorting and ordering (max, min) of **List**:

```
/* Utility Class java.util.Collections */
static <T> int binarySearch(List<? extends T> lst, T key, Comparator<? super T>
    ↪ comp)
static <T> int binarySearch(List<? extends Comparable<? super T>> lst, T key)
// Searches for the specified key using binary search

static <T> void sort(List<T> lst, Comparator<? super T> comp)
static <T extends Comparable<? super T>> void sort(List<T> lst)

static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T>
    ↪ coll)
static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T>
    ↪ coll)
```

Each of these algorithms has two versions:

1. Requires a **Comparator** object with a $compare()$ method to determine the order of the elements.
2. Requires a **List** which implement **Comparable** interface, with a method *compareTo()* to determine the order of elements.

We shall elaborate in the next section.

1.6 List Ordering / Searching / Sorting with Comparable<T> / Comparator<T>

Ordering is needed in these two situations:

1. To sort a **Collection** or an array (using the **Collections**.$sort()$ or **Arrays**.$sort()$ methods), an ordering specification is needed.
2. Some **Collections**, in particular, **SortedSet** (**TreeSet**) and **SortMap** (**TreeMap**), are ordered. That is, the objects are stored according to a specified order.

There are two ways to specify the ordering of objects:

1. Create a special **java.util.Comparator**<**T**> object, with a method *compare*() to specify the ordering of comparing two objects.

2. Make the objects implement the **java.lang.Comparable**<**T**> interface, and override the *compareTo*() method to specify the ordering of comparing two objects.

1.6.1 Comparable<T> Interface

A **java.lang.Comparable**<**T**> interface specifies how two objects are to be compared for ordering. It defines one abstract method:

```
/* Interface java.lang.Comparable<T> */

abstract int compareTo(T o)
// Returns a negative integer, zero, or a positive integer
// if this object is less than, equal to, or greater than the given object
```

This ordering is referred to as the class's *natural ordering*.

It is strongly recommended that *compareTo*() be consistent with *equals*() and *hashCode*() (inherited from **java.lang.Object**):

1. If *compareTo*() returns a zero, *equals*() should return *true*.

2. If *equals*() returns *true*, *hashCode*() shall produce the same *int*.

All the eight primitive wrapper classes (**Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character** and **Boolean**) implement **Comparable**<**T**> interface, with the *compareTo*() uses the numerical order.

The *String* class also implements **Comparable**<**String**> interface, which compares two strings lexicographically based on the Unicode value of each character in the strings. The uppercase letters are smaller than the lowercase counterparts.

Example 1. Searching/Sorting String and Primitive Wrapper Types, which implement Comparable<T>

The utility class **java.util.Arrays** and **java.util.Collections** provide many static method for the various algorithms such as sorting and searching.

In this example, we use the **Arrays**.*sort*() and **Collections**.*sort*() methods to sort an array of Strings and a **List** of Integers, based on their **Comparable**<**T**>'s *compareTo*() method.

```java
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

/**
 * TestStringPrimitiveComparable.java
 * The test driver class to test string and primitive Comparable
 */
public class TestStringPrimitiveComparable {
   public static void main(String[] args) {
      // Sort/search an "array" of Strings using Arrays.sort() and
      // Arrays.binarySearch() using ordering specified in compareTo()
      String[] strArray = {"Hello", "hello", "Hi", "HI", "Hello"};  // has duplicate
            ↪ elements

      Arrays.sort(strArray);  // sort in place and mutable
      System.out.println(Arrays.toString(strArray));  // [HI, Hello, Hello, Hi, hello]

      // The array must be sorted for binarySearch()
      System.out.println(Arrays.binarySearch(strArray, "Hello")); // 2
      System.out.println(Arrays.binarySearch(strArray, "HELLO")); // -1 (insertion
            ↪ at index 0)

      // Sort/search a List<Integer> using Collections.sort() and Collections.binarySearch()
      List<Integer> list = new ArrayList<>();
      list.add(22);  // int auto-box to Integer
      list.add(11);
      list.add(44);
      list.add(11);  // duplicate element
      Collections.sort(list);      // sort in place and mutable
      System.out.println(list);  // [11, 11, 22, 44]
      System.out.println(Collections.binarySearch(list, 22)); // 2
      System.out.println(Collections.binarySearch(list, 35)); // -4 (insertion at
            ↪ index 3)
   }
}
```

Example 2. Custom Implementation of Comparable<T>

Let's create a subclass of **Person** (see "Person.java" above), called **ComparablePerson** which implements **Comparable**<**Person**> interface, and try out the **Collections**.*sort*() and **Collections**.*binarySearch*() methods.

```java
/**
 * ComparablePerson.java
```

```
 * The ComparablePerson class
 */
public class ComparablePerson extends Person implements Comparable<Person> {
   public ComparablePerson(String name, int age) {    // constructor
      super(name, age);
   }

   // Order by the name strings, case insensitive
   @Override
   public int compareTo(Person p) {
      return this.getName().compareToIgnoreCase(p.getName());   // via String's
            ↪ compareToIgnoreCase()
   }
}
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

/**
 * TestComparablePerson.java
 * The test driver class to test Person Comparable
 */
public class TestComparablePerson {
   public static void main(String[] args) {
      List<ComparablePerson> personList = new ArrayList<>();
      personList.add(new ComparablePerson("Peter", 21));
      personList.add(new ComparablePerson("Paul", 18));
      personList.add(new ComparablePerson("John", 60));
      System.out.println(personList);   // [Peter(21), Paul(18), John(60)]

      // Use compareTo() for ordering
      Collections.sort(personList);             // sort in place, mutable
      System.out.println(personList);       // [John(60), Paul(18), Peter(21)]

      // Use compareTo() too
      System.out.println(Collections.binarySearch(personList,
         new ComparablePerson("PAUL", 18)));   // 1

      System.out.println(Collections.binarySearch(personList,
         new ComparablePerson("PAUL", 16)));   // 1

      System.out.println(Collections.binarySearch(personList,
         new ComparablePerson("Kelly", 18)));  // -2
   }
}
```

1.6.2 Comparator<T> Interface

Besides the **java.lang.Comparable<T>** for the natural ordering, you can pass a **java.util.Comparator<T>** object into the sorting methods (*Collections.sort()* or *Arrays.sort()*) to provide precise control over the ordering. The **Comparator<T>** will override the **Comparable<T>**, if available.

The **Comparator<T>** interface declares one abstract method (known as Functional Interface in JDK 8):

```
/* java.util.Comparator<T> */
/*
 * Returns a negative integer, zero, or a positive integer as the first
 *   argument is less than, equal to, or greater than the second.
 */
abstract int compare(T o1, T o2)
```

Take note that you need to construct an instance of **Comparator<T>**, and invoke *compare()* to compare o1 and o2. [In the earlier Comparable<T>, the method is called *compareTo()* and it takes only one argument, i.e., this object compare to the given object.]

Example 3. Using Customized Comparator<T> for *String* and Integer

In this example, instead of using the natural **Comparable<T>**, we define our customized **Comparator<T>** for Strings and Integers. We can do this via any of the following ways:

1. A named inner class
2. An anonymous inner class
3. Lambda Expressions (JDK 8)

```
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;

/**
 * TestStringPrimitiveComparator.java
 * The test driver class to test string and primitive comparator
 */
public class TestStringPrimitiveComparator {
```

```
12    // Define an "named inner class" implements Comparator<String>
      //    to order strings in case-insensitive manner
14    public static class StringComparator implements Comparator<String> {
        @Override
16      public int compare(String s1, String s2) {
          return s1.compareToIgnoreCase(s2);
18      }
      }
20
      public static void main(String[] args) {
22      // Use a customized Comparator for Strings
        Comparator<String> strComp = new StringComparator();
24
        // Sort and search an "array" of Strings
26      String[] array = {"Hello", "Hi", "HI", "hello", "Hello"};   // with duplicate
        Arrays.sort(array, strComp);
28      System.out.println(Arrays.toString(array));
        // [Hello, hello, Hello, Hi, HI]
30      System.out.println(Arrays.binarySearch(array, "Hello", strComp));
        // 2
32      System.out.println(Arrays.binarySearch(array, "HELLO", strComp));
        // 2 (case-insensitive)
34
        // Use an "anonymous inner class" to implement Comparator<Integer>
36      Comparator<Integer> intComp = new Comparator<Integer>() {
          @Override
38        public int compare(Integer i1, Integer i2) {
            return i1%10 - i2%10;
40        }
        };
42
        // Sort and search a "List" of Integers
44      List<Integer> list = new ArrayList<Integer>();
        list.add(42);   // int auto-box Integer
46      list.add(21);
        list.add(34);
48      list.add(13);
        Collections.sort(lst, intComp);
50      System.out.println(list);   // [21, 42, 13, 34]
        System.out.println(Collections.binarySearch(list, 22, intComp));
52      // 1
        System.out.println(Collections.binarySearch(list, 35, intComp));
54      // -5 (insertion at index 4)
      }
56  }
```

Try: Modify the **Comparator** to sort in A, a , B, b, C, c ... (uppercase letter before the lowercase).

Notes: You can use Lambda Expressions (JDK 8) to shorten this code, as follows:

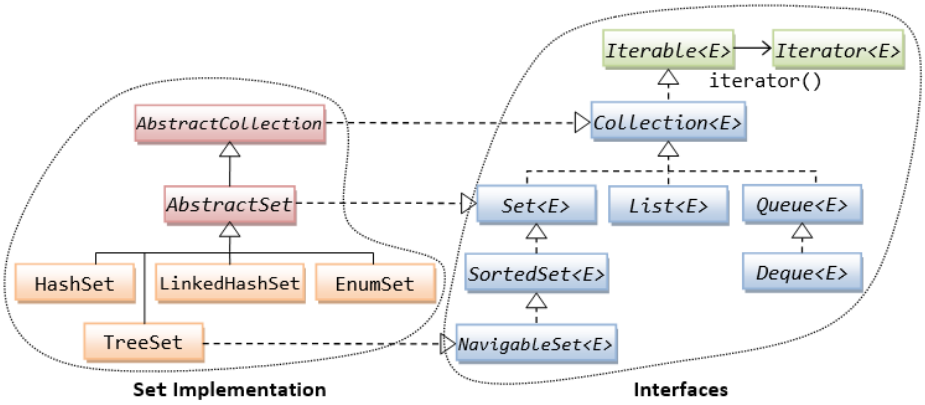
```
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

/**
 * TestStringPrimitiveComparatorJ8.java
 * The test driver class to test string and primivite comparator J8
 */
public class TestStringPrimitiveComparatorJ8 {   // JDK 8
   public static void main(String[] args) {
      // Use a customized Comparator for Strings
      Comparator<String> strComp = (s1, s2) -> s1.compareToIgnoreCase(s2);
      /*
       * The lambda expression create an instance of an anonymous inner class implements
       *  Comparator<String> with the body of the single-abstract-method compare()
       */

      // Sort and search an "array" of Strings
      String[] array = {"Hello", "Hi", "HI", "hello", "Hello"};   // with duplicate
      Arrays.sort(array, strComp);
      System.out.println(Arrays.toString(array));
      // [Hello, hello, Hello, Hi, HI]
      System.out.println(Arrays.binarySearch(array, "Hello", strComp));
      // 2
      System.out.println(Arrays.binarySearch(array, "HELLO", strComp));
      // 2 (case-insensitive)

      // Use a customized Comparator for Integers
      Comparator<Integer> intComp = (i1, i2) -> i1%10 - i2%10;

      // Sort and search a "List" of Integers
      List<Integer> list = new ArrayList<Integer>();
      list.add(42);   // int auto-box Integer
      list.add(21);
      list.add(34);
      list.add(13);
      Collections.sort(list, intComp);
      System.out.println(list);   // [21, 42, 13, 34]
      System.out.println(Collections.binarySearch(list, 22, intComp));
      // 1
      System.out.println(Collections.binarySearch(list, 35, intComp));
      // -5 (insertion at index 4)
   }
}
```

Example 4. Using Customized Comparator<Person> for Person Objects

```
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Collections;

/**
 * The test driver class to test Person Comparator
 */
public class TestComparatorPerson {
   public static void main(String[] args) {
      List<Person> personList = new ArrayList<>();
      personList.add(new Person("Peter", 21));
      personList.add(new Person("Paul", 18));
      personList.add(new Person("John", 60));
      System.out.println(personList);   // [Peter(21), Paul(18), John(60)]

      // Using an anonymous inner class
      Comparator<Person> comp = new Comparator<>() {
         @Override
         public int compare(Person p1, Person p2) {
            return p1.getName().compareToIgnoreCase(p2.getName());
         }
      };

      // Use compareTo() for ordering
      Collections.sort(personList, comp);
      System.out.println(personList);   // [John(60), Paul(18), Peter(21)]

      // Use compareTo() too
      System.out.println(Collections.binarySearch(personList,
         new Person("PAUL", 18), comp));   // 1
      System.out.println(Collections.binarySearch(personList,
         new Person("PAUL", 16), comp));   // 1
      System.out.println(Collections.binarySearch(personList,
         new Person("Kelly", 18), comp)); // -2

      // Using JDK 8 Lambda Expression to create an instance of anonymous
      //   inner class implements Comparator<Person>
      personList.add(new Person("Janes", 30));
      System.out.println(personList);   // [John(60), Paul(18), Peter(21), Janes(30)]

      Collections.sort(personList,
         (p1, p2) -> p1.getName().toLowerCase().compareTo(p2.getName().toLowerCase()));
      System.out.println(personList);   // [Janes(30), John(60), Paul(18), Peter(21)]
   }
}
```

1.7 Set<E> Interfaces, Implementations and Algorithms

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.



The **Set**<**E**> interface declares the following abstract methods. The insertion, deletion and inspection methods returns false if the operation fails, instead of throwing an exception.

```
/* Interface java.util.Set<E> */
// Adds the specified element if it is not already present
abstract boolean add(E o)

// removes the specified element if it is present
abstract boolean remove(Object o)

// returns true if it contains o
abstract boolean contains(Object o)

// Set operations
// Set union
abstract boolean addAll(Collection<? extends E> c)

// Set intersection
abstract boolean retainAll(Collection<?> c)
```

The implementations of **Set**<**E**> interface include:

- **HashSet<E>**: Stores the elements in a hash table (hashed via the *hashcode()*). **HashSet** is the best all-round implementation for Set.
- **LinkedHashSet<E>**: Stores the elements in a linked-list hash table for better efficiency in insertion and deletion. The element are hashed via the *hashCode()* and arranged in the linked list according to the insertion-order.
- **TreeSet<E>**: Also implements sub-interfaces **NavigableSet** and **SortedSet**. Stores the elements in a red-black tree data structure, which are sorted and navigable. Efficient in search, add and remove operations (in O(log(n))).

1.7.1 HashSet<E> By Example

Let's write a **Book** class, and create a **Set** of **Book** objects.

```
/**
 * Book.java
 */
public class Book {
   private int id;
   private String title;

   public Book(int id, String title) {   // constructor
      this.id = id;
      this.title = title;
   }

   @Override
   public String toString() {   // describle itself
      return id + ": " + title;
   }

   // Two books are equal if they have the same id
   @Override
   public boolean equals(Object o) {
      return o != null && o instanceof Book && this.id == ((Book)o).id;
   }

   // To be consistent with equals(). Two objects which are equal have the same hash code.
   @Override
   public int hashCode() {
      return id;
   }
}
```

We need to provide an $equals()$ method, so that the Set implementation can test for equality and duplication. In this example, we choose the id as the distinguishing

feature. We override $equals()$ to return true if two books have the same id. We also override the $hashCode()$ to be consistent with $equals()$.

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

/**
 * TestHashSet.java
 * The test driver class to test HashSet
 */
public class TestHashSet {
  public static void main(String[] args) {
    Book book1 = new Book(1, "Java for Dummies");
    Book book1Dup = new Book(1, "Java for the Dummies"); // same id as above
    Book book2 = new Book(2, "Java for more Dummies");
    Book book3 = new Book(3, "more Java for more Dummies");

    Set<Book> set1 = new HashSet<Book>();
    set1.add(book1);
    set1.add(book1Dup); // duplicate id, not added
    set1.add(book1);    // added twice, not added
    set1.add(book3);
    set1.add(null);     // Set can contain a null
    set1.add(null);     // but no duplicate
    set1.add(book2);
    System.out.println(set1);
    // [null, 1: Java for Dummies, 2: Java for more Dummies, 3: more Java for more
        Dummies]

    set1.remove(book1);
    set1.remove(book3);
    System.out.println(set1); // [null, 2: Java for more Dummies]

    Set<Book> set2 = new HashSet<Book>();
    set2.add(book3);
    System.out.println(set2); // [3: more Java for more Dummies]

    set2.addAll(set1);          // "union" with set1
    System.out.println(set2); // [null, 2: Java for more Dummies, 3: more Java for
        ↪ more Dummies]

    set2.remove(null);
    System.out.println(set2); // [2: Java for more Dummies, 3: more Java for more
        ↪ Dummies]

    set2.retainAll(set1);       // " intersection " with set1
    System.out.println(set2); // [2: Java for more Dummies]
  }
}
```

- A **Set** cannot hold duplicate element. The elements are check for duplication via the overridden $equal()$.
- A **Set** can hold a null value as its element (but no duplicate too).
- The $addAll()$ and $retainAll()$ perform set union and set intersection operations, respectively.

Take note that the arrangement of the elements is arbitrary, and does not correspond to the order of $add()$.

1.7.2 LinkedHashSet<E> By Example

Unlike **HashSet**, **LinkedHashSet** builds a link-list over the hash table for better efficiency in insertion and deletion (in the expense of more complex structure). It maintains its elements in the insertion-order (i.e., order of $add()$).

```
1  import java.util.LinkedHashSet;
   import java.util.Set;
3
   /**
5   * TestLinkedHashSet.java
    * The test driver class to test LinkedHashSet
7   */
   public class TestLinkedHashSet {
9    public static void main(String[] args) {
       Book book1 = new Book(1, "Java for Dummies");
11     Book book1Dup = new Book(1, "Java for the Dummies"); // same id as above
       Book book2 = new Book(2, "Java for more Dummies");
13     Book book3 = new Book(3, "more Java for more Dummies");

15     Set<Book> set = new LinkedHashSet<Book>();
       set.add(book1);
17     set.add(book1Dup); // duplicate id, not added
       set.add(book1);    // added twice, not added
19     set.add(book3);
       set.add(null);     // Set can contain a null
21     set.add(null);     // but no duplicate
       set.add(book2);
23     System.out.println(set);
       // [1: Java for Dummies, 3: more Java for more Dummies,
25     //  null, 2: Java for more Dummies]
     }
27 }
```

The output clearly shows that the set is ordered according to the order of $add()$.

1.7.3 SortedSet<E> and NavigableSet<E> Interfaces

Elements in a **SortedSet<E>** are sorted during *add()*, either using the natural ordering in the **Comparable<T>**, or given a **Comparator<T>** object. Read "Ordering, Sorting and Searching" for details on **Comparable<T>** and **Comparator<T>**.

The **NavigableSet<E>** is a sub-interface of **SortedSet<E>**, which declares these additional navigation methods:

```
1  // Interface java.util.NavigableSet<E>
   // Returns an iterator in ascending order.
3  abstract Iterator<E> iterator()

5  // Returns an iterator in descending order.
   abstract Iterator<E> descendingIterator()
7
   // Per-Element operation
9  // Returns the greatest element less than or equal to the
   //  given element, or null if there is no such element.
11 abstract E floor(E e)

13 // Returns the least element greater than or equal to the
   //  given element, or null
15 abstract E ceiling(E e)

17 // Returns the greatest element strictly less than the
   //  given element, or null
19 abstract E lower(E e)

21 // Returns the least element strictly greater than the
   //  given element, or null
23 abstract E higher(E e)

25 // Subset operation
   // Returns a view whose elements are strictly less than toElement.
27 abstract SortedSet<E> headSet(E toElement)

29 // Returns a view whose elements are greater than or equal to fromElement.
   abstract SortedSet<E> tailSet(E fromElement)
31
   // Returns a view whose elements range from fromElement (inclusive) to toElement
       (exclusive)
33 abstract SortedSet<E> subSet(E fromElement, E toElement)
```

1.7.4 TreeSet<E> by Example

TreeSet<E> is an implementation to **NavigableSet<E>** and **SortedSet<E>**.

Example 1. TreeSet with Comparable<E>

```java
/**
 * AddressBookEntry.java
 */
public class AddressBookEntry implements Comparable<AddressBookEntry> {
   private String name;
   private String address;
   private String phone;

   public AddressBookEntry(String name) {
      this.name = name;
   }

   @Override
   public String toString() {   // describe itself
      return name;
   }

   @Override
   public int compareTo(AddressBookEntry other) {  // Interface Comparable<T>
      return this.name.compareToIgnoreCase(other.name);
   }

   @Override
   public boolean equals(Object o) {
      return (o != null}
         && (o instanceof AddressBookEntry)
         && this.name.equalsIgnoreCase(((AddressBookEntry)o).name);
   }

   // Two objects which are equals() shall have the same hash code
   @Override
   public int hashCode() {
      return name.toLowerCase().hashCode();
   }
}
```

This **AddressBookEntry** class implements **Comparable**, in order to be used in **TreeSet**. It overrides $compareTo()$ to compare the name in a case insensitive manner. It also overrides $equals()$ and $hashCode()$, so as they are consistent with the $compareTo()$.

```java
import java.util.TreeSet;

/**
 * TestTreeSetComparable.java
```

```
 * The test driver class to test TreeSet Comparable
 */
public class TestTreeSetComparable {
  public static void main(String[] args) {
    AddressBookEntry address1 = new AddressBookEntry("peter");
    AddressBookEntry address2 = new AddressBookEntry("PAUL");
    AddressBookEntry address3 = new AddressBookEntry("Patrick");

    TreeSet<AddressBookEntry> set = new TreeSet<>();
    set.add(address1);
    set.add(address2);
    set.add(address3);
    System.out.println(set); // [Patrick, PAUL, peter]

    System.out.println(set.floor(address2));     // PAUL
    System.out.println(set.lower(address2));     // Patrick
    System.out.println(set.headSet(address2)); // [Patrick]
    System.out.println(set.tailSet(address2)); // [PAUL, peter]
  }
}
```

Observe that the **AddressBookEntry** objects are sorted and stored in the order depicted by the **Comparable<T>** during $add()$ operation.

Example 2. TreeSet with Comparator<T>

Let's rewrite the previous example to use a **Comparator** object instead of **Comparable**. We shall set the **Comparator** to order in descending order of name for illustration.

```
/**
 * PhoneBookEntry.java
 */
public class PhoneBookEntry {
  public String name;
  public String address;
  public String phone;

  public PhoneBookEntry(String name) {   // constructor, ignore address and phone
    this.name = name;
  }

  @Override
  public String toString() {
    return name;
  }
```

```
}
```

The **PhoneBookEntry** class does not implement **Comparator**. You cannot add a **PhoneBookEntry** object into a **TreeSet** as in the above example. Instead, we define a **Comparator** class, and use an instance of **Comparator** to construct a **TreeSet**.

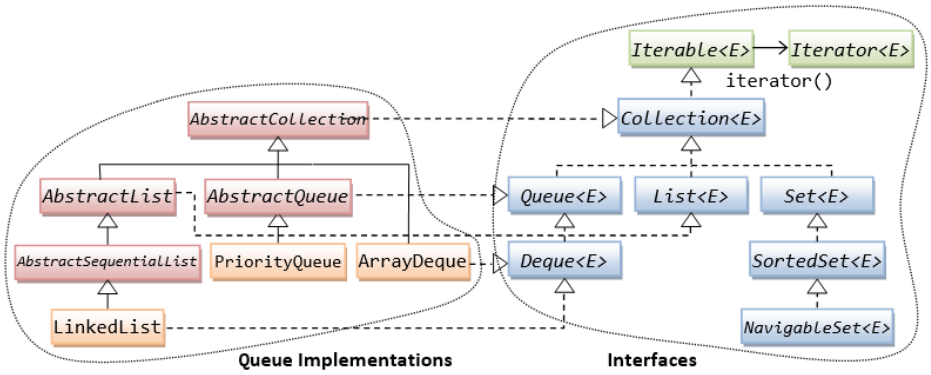
The **Comparator** orders the **PhoneBookEntry** objects in descending name and case insensitive.

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Comparator;

/**
 * TestTreeSetComparable.java
 * The test driver class to test TreeSet Comparator
 */
public class TestTreeSetComparator {
   // Using a named inner class to implement Comparator<T>
   public static class PhoneBookComparator implements Comparator<
         ↪ PhoneBookEntry> {
      @Override
      public int compare(PhoneBookEntry p1, PhoneBookEntry p2) {
         return p2.name.compareToIgnoreCase(p1.name); // descending name
      }
   }

   public static void main(String[] args) {
      PhoneBookEntry addr1 = new PhoneBookEntry("peter");
      PhoneBookEntry addr2 = new PhoneBookEntry("PAUL");
      PhoneBookEntry addr3 = new PhoneBookEntry("Patrick");

      Comparator<PhoneBookEntry> comp = new PhoneBookComparator();
      TreeSet<PhoneBookEntry> set = new TreeSet<PhoneBookEntry>(comp);
      set.add(addr1);
      set.add(addr2);
      set.add(addr3);
      System.out.println(set);      // [peter, PAUL, Patrick]

      Set<PhoneBookEntry> newSet = set.descendingSet();   // Reverse the order
      System.out.println(newSet); // [Patrick, PAUL, peter]
   }
}
```

In the test program, we construct a **TreeSet** with the **BookComparator**. We also tried the $descendingSet()$ method to obtain a new **Set** in reverse order.

1.8 Queue<E> Interfaces, Implementations and Algorithms

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.



Besides basic **Collection**<**E**> operations, **Queue**<**E**> provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operations). The latter form of the insert operation is designed specifically for use with capacity-restricted **Queue** implementations

```
1 /* Interface java.util.Queue<E> */
  // Insertion at the end of the queue
3 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
5
  // Extract element at the head of the queue
7 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
9
```

```
// 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
```

Deque<E> declares additional methods to operate on both ends (head and tail) of the queue.

```
/* Interface java.util.Deque<E> */
// Insertion
abstract void addFirst(E e)
abstract void addLast(E e)
abstract boolean offerFirst(E e)
abstract boolean offerLast(E e)

// Retrieve and Remove
abstract E removeFirst()
abstract E removeLast()
abstract E pollFirst()
abstract E pollLast()

// Retrieve but does not remove
abstract E getFirst()
abstract E getLast()
abstract E peekFirst()
abstract E peekLast()
```

A **Deque** can be used as FIFO queue (via methods $add(e)$, $remove()$, $element()$, $offer(e)$, $poll()$, $peek()$) or LIFO queue (via methods $push(e)$, $pop()$, $peek()$).

The **Queue<E>** and **Deque<E>** implementations include:

- **PriorityQueue<E>**: A queue where the elements are ordered based on an ordering you specify, instead of FIFO.
- **ArrayDeque<E>**: A queue and deque implemented as a dynamic array, similar to **ArrayList<E>**.
- **LinkedList<E>**: The **LinkedList<E>** also implements the **Queue<E>** and **Deque<E>** interfaces, in additional to **List<E>** interface, providing a queue or deque that is implemented as a double-linked list data structure.

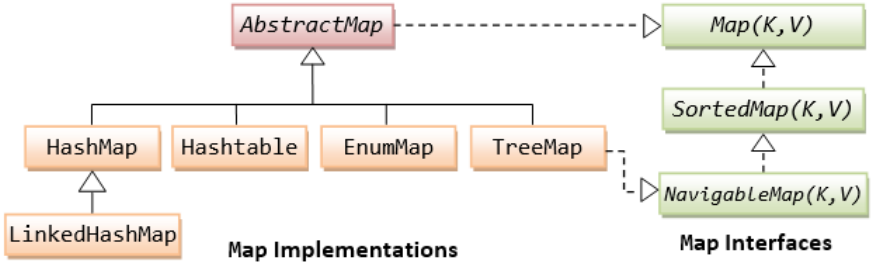
The basic operations of **Queue<E>** include adding an element, polling the queue to retrieve the next element, or peeking at the queue to see if there is an element

available in the queue. The **Deque**<**E**> operations are similar except element can be added, polled, or peeked at both ends of the deque.

[TODO] Example

1.9 Map<K, V> Interfaces, Implementations and Algorithms

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).



The implementations of **Map**<**K, V**> interface include:

- **HashMap**<**K, V**>: Hash table implementation of the **Map**<**K, V**> interface. The best all-around implementation. Methods in **HashMap** is not synchronized.
- **TreeMap**<**K, V**>: Red-black tree implementation of the **SortedMap**<**K, V**> interface.
- **LinkedHashMap**<**K,V**>: Hash table with link-list to facilitate insertion and deletion.
- **Hashtable**<**K, V**>: Retrofitted legacy (JDK 1.0) implementations. A synchronized hash table implementation of the **Map**<**K, V**> interface that does not allow null key or values, with legacy methods.

Basic Operations

The **Map**<**K, V**> interface declares the following abstract methods for basic operations:

```
/* Interface java.util.Map<K, V> */
// Returns the number of key-value pairs
abstract int size()

// Returns true if this map contain no key-value pair
abstract boolean isEmpty()

// Returns the value of the specified key
abstract V get(Object key)

// Associates the specified value with the specified key
abstract V put(K key, V value)

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

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

// Removes all key-value pairs
abstract void clear()

// Removes the specified key
abstract void V remove(object key)
```

Collection Views

The **Map<K,V>** provides these method to allow a map to be viewed as a **Collection**:

```
// java.util.Map(K, V)
abstract Set<K> keySet()              // Returns a set view of the keys
abstract Collection<V> values()       // Returns a collection view of the values
abstract Set<Map.Entry<K, V>> entrySet()  // Returns a set view of the key-value
```

The nested class **Map.Entry<K,V>** contains these methods:

```
// Nested Class Map.Entry<K, V>
K getKey()      // Returns the key of this map entry
V getValue()    // Returns the value of this map entry
V setValue()    // Replaces the value of this map entry
```

The **Map** does not have List-like iterator. The **Collection** views provides the means to iterate over a map.

Example 1. Iterating through a Map using entrySet() and keySet()

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;

/**
 * TestMapView.java
 * The test driver class to test Map
 */
public class MapViewTest {
   public static void main(String[] args) {
      Map<String, Double> map = new HashMap<>();
      map.put("espresso", 1.1);
      map.put("latte", 2.2);
      map.put("cappuccino", 3.3);
      System.out.println(map);  // {espresso=1.1, cappuccino=3.3, latte=2.2}

      // Using .entrySet() which returns a Set<Map.Entry> to iterate through the map
      for (Map.Entry<String, Double> e : map.entrySet()) {
         e.setValue(e.getValue() + 10.0);   // modify value
         System.out.println(e.getKey() + ":" + e.getValue());
      }
      // espresso:11.1
      // cappuccino:13.3
      // latte:12.2

      // Using for-each loop on .keySet() which returns a Set to iterate through the map
      // .keySet() returns a Set of keys
      System.out.println(map.keySet());   // [espresso, cappuccino, latte]
      for (String key : map.keySet()) {
         System.out.println(key + "=" + map.get(key));
      }
      // espresso=11.1
      // cappuccino=13.3
      // latte=12.2

      // Using Iterator on .keySet() to iterate through the map
      Iterator<String> iter = map.keySet().iterator();
      while (iter.hasNext()) {
         String key = iter.next();
         System.out.println(key + ":" + map.get(key));
      }
      // espresso:11.1
      // cappuccino:13.3
      // latte:12.2
```

```
46      // .values() returns a Collection of values
        System.out.println(map.values());  // [21.1, 23.3, 22.2]
48    }
  }
```

Example 2. Word Count using HashMap<String, Integer>

```
1  import java.util.Map;
   import java.util.HashMap;
3  import java.util.Scanner;
   import java.io.File;
5
   /**
7   * WordCount.java
    * Counts the frequency of each of the words in a file given in
9   *  the command-line, and saves in a map of {word, freq}.
    */
11 public class WordCount {
     public static void main(String[] args) throws Exception {
13     Scanner in = new Scanner(new File(args[0]));

15     Map<String, Integer> map = new HashMap<>();
       while (in.hasNext()) {
17       String word = in.next();
         int freq = (map.get(word) == null) ? 1 : map.get(word) + 1;    // type-safe
19       map.put(word, freq);          // auto-box int to Integer and upcast, type-check
       }
21     System.out.println(map);

23     in.close();
     }
25 }
```

1.10 Utilities Class java.util.Arrays

The Collection Framework provides two utility classes: **java.util.Arrays** and **java.util.Collections**, which provide some useful algorithms, such as sort, shuffle, reverse, and search, on arrays and Collections.

Array is a reference type in Java. It can hold primitives, as well as objects. On the other hand, a Collection holds only object.

1.10.1 Array Sorting/Searching: Arrays.sort() and Arrays.binarySearch()

Sorting of Primitive and Object Arrays

There is a pair of *sort()* methods for each of the primitive types (except *boolean*) and **Object**.

For example, for *int*[ ]:

```
/* Utility Class java.util.Arrays */
// Sorting of primitive arrays (except boolean[])
static sort(int[] a) -> void
static sort(int[] a, int fromIdx, int toIdx) -> void
// Similar methods for byte[], short[], long[], float[], double[] and char[]

// Sorting of Object[]
static sort(Object[] a) -> void
static sort(Object[] a, int fromIdx, int toIdx) -> void
```

Similar *sort()* methods are available for primitive arrays *byte*[ ], *short*[ ], *long*[ ], *float*[ ], *double*[ ], *char*[ ] (except *boolean*[ ]), and *Object*[ ]. For *Object*[ ], the objects must implement **Comparable**<T> interface so that the ordering can be determined via the *compareTo()* method.

Soriting for Generic Arrays

A pair of methods is also defined for generic, to be sorted based on the given **Comparator** (instead of **Comparable**).

```
// Sorting of specified array according to the order induced by the comparator.
static sort(T[] a, Comparator<Ts> c) -> void
static sort(T[] a, int fromIdx, int toIdx, Comparator<? super T> c) -> void
```

Notes: Suppose that you wish to sort an array of **Integer** (where **T** is **Integer**), you could use a **Comparator**<**Integer**>. You can also use **Comparator**<**Number**> or **Comparator**<**Object**>, as **Object** and **Number** are superclass of **Integer**.

Seaching of Primitive, Object and Generic Arrays

Similarly, there is a pair of searching method for each of the primitive arrays (except *boolean*[ ]) and **Object**. The arrays must be sorted before you can apply the *binarySearch()* method.

```
/* Utility Class java.util.Arrays */
// Methods search the specified array of the given data type
//  for the specified value using the binary search algorithm
static binarySearch(int[] a, int key) -> int
static binarySearch(int[] a, int fromIdx, int toIdx, int key) -> int
// Similar methods for byte[], short[], long[], float[], double[] and char[]

// Searching object[], which implements Comparable
static binarySearch(Object[] a, Object key) -> int
static binarySearch(Object[] a, int fromIdx, int toIdx, Object key) -> int

// Searching generic array, order by the given Comparator
static binarySearch(T[] a, T key, Comparator<? super T> c) -> int
static binarySearch(T[] a, T key, int fromIdx, int toIdx, Comparator<? super T> c)
    ↪ -> int
```

[TODO] Examples

1.10.2 Equality Comparison: Arrays.equals()

```
// Method checks whether two arrays are equal or not.
static equals(int[] a, int[] b) -> boolean
// Similar methods for byte[], short[], long[], float[], double[], char[], boolean[] and
    Object[]
```

1.10.3 Copying: Arrays.copyOf() and Arrays.copyOfRange()

```
// Copies the given array, truncating or padding with zeros (if necessary)
//  so the copy has the specified length
static copyOf(int[] src, int length) -> int[]

static copyOfRange(int[] src, int fromIdx, int toIdx) -> int[]
// Similar methods for byte[], short[], long[], float[], double[], char[] and boolean[]

static copyOf(T[] src, int length) -> T[]
static copyOfRange(T[] src, int fromIdx, int toIdx) -> T[]
static copyOf(U[] src, int length, Class<? extends T[]> newType) -> T[]
static copyOfRange(U[] src, int fromIdx, int toIdx, Class<? extends T[]>
    ↪ newType) -> T[]
```

1.10.4 Filling: Arrays.fill()

```
// Makes all elements of a[] equal to "$value$"
static fill(int[] a, int value) -> void

// Makes elements from fromIndex (inclusive) to toIndex
//   (exclusive) equal to "$value$"
static fill(int[] a, int fromIdx, int toIdx, int value) -> void

/*
 * Similar methods for byte[], short[], long[], float[], double[],
 *  char[] and boolean[] and Object[]
 */
```

1.10.5 Description: Arrays.toString()

```
// Returns a string representation of the contents of the specified array.
static toString(int[] a) -> String

/*
 * Similar methods for byte[], short[], long[], float[], double[],
 *  char[] and boolean[] and Object[]
 */
```

1.10.6 Converting to List: Arrays.asList()

```
// Returns a fixed-size list backed by the specified array.
// Change to the list write-thru to the array.
static asList(T[] a) -> List<T>
```

1.11 Utilities Class java.util.Collections

The Collection Framework provides two utility classes: **java.util.Arrays** and **java.util.Collections**, which provide some useful algorithms, such as sort, shuffle, reverse, and search, on arrays and Collections. Take note that the interface is called Collection, while the utility class is called **Collections** with a 's'.

1.11.1 List Searching/Sorting: Collations.sort()/binarySearch()

```
// Utility Class java.util.Collections
// Sorts the specified list into ascending order. The objects shall implement Comparable.
static sort(List<T> list) -> void

// Sorts the specified list according to the order induced by the specified comparator.
static sort(List<T> list, Comparator<? super T> c) -> void

// Returns index of key in sorted list sorted in ascending order
static binarySearch(List<? extends Comparable<? super T>> list, T key) -> int

// Returns index of key in sorted list sorted in
//  order defined by Comparator c.
static binarySearch(List<? extends T> list, T key, Comparator<? super T> c) -> int
```

1.11.2 Maximum and Minimum: Collections.max()|min()

```
// Returns the maximum/minimum element of the given collection,
//  according to the natural ordering of its elements.
static max(Collection<? extends T> c) -> T
static min(Collection<? extends T> c) -> T

// Returns the maximum/minimum element of the given collection,
//  according to the order induced by the specified comparator.
static max(Collection<? extends T> c, Comparator<? super T> comp) -> T
static min(Collection<? extends T> c, Comparator<? super T> comp) -> T
```

1.11.3 Synchronized Wrapper: Collections.synchronizedXxx()

Most of the **Collection** implementations such as **ArrayList**, **HashSet** and **HashMap** are NOT synchronized for multi-threading, except the legacy **Vector** and **HashTable**, which are retrofitted to conform to the Collection Framework and synchronized. Instead of using the synchronized **Vector** and **HastTable**, you can create a synchronized **Collection**, **List**, **Set**, **SortedSet**, **Map** and **SortedMap**, via the static **Collections**.$synchronizedXxx()$ methods:

```
/* Class java.util.Collections */
// Returns a synchronized (thread-safe) collection backed by the specified collection.
```

```
static synchronizedCollection(Collection<T> c) -> Collection<T>
static synchronizedList(List<T> list) -> List<T>
static synchronizedSet(Set<T> set) -> Set<T>
static synchronizedSortedSet(SortedSet<T> set) -> SortedSet<T>
static synchronizedMap(Map<K, V> map) -> Map<K, V>
static synchronizedSortedMap(SortedMap<K, V> map) -> SortedMap<K, V>
```

According to the JDK API specification, "to guarantee serial access, it is critical that all access to the backing list is accomplished through the returned list, and that user manually synchronize on the returned list when iterating over it". For example,

```
List list = Collections.synchronizedList(new ArrayList());
  ......
  ......

synchronized(list) {   // must be enclosed in a synchronized block
   Iterator iter = list.iterator();
   while (iter.hasNext())
   iter.next();
     ......
}
```

1.11.4 Unmodifiable Wrappers: Collections.unmodifiableXxx()

Unmodifiable wrappers prevent modification of the collection by intercepting all the operations that would modify the collection and throwing an **UnsupportedOperationException**.

The **Collections** class provides six static methods to wrap the interfaces **Collection**, **List**, **Set**, **SortedSet**, **Map** and **SortedMap**.

```
/* Class java.util.Collections */
// Methods return an unmodifiable view of the specified collection.
static unmodifiableCollection(Collection<? extends T> c) -> Collection<T>
static unmodifiableList(List<? extends T> list) -> List<T>
static unmodifiableSet(Set<? extends T> s) -> Set<T>
static unmodifiableSortedSet(SortedSet<? extends T> s) -> SortedSet<T>
static unmodifiableMap(Map<? extends K, ? extends V> m) -> Map<K, V>
static unmodifiableSortedMap(SortedMap<K, ? extends V> m) -> SortedMap<K, V>
```

1.12 Utility Class java.util.Objects (JDK 7)

Notes: By JDK convention, Object is a regular class modeling objects, and **Objects** is an utility class containing static methods for **Object**.

The utility class **java.util.Objects** contains static methods for operating on **Objects** (such as comparing two objects and computing hash code), and checking certain conditions (such as checking for *null*, checking if indexes are out of bounds) before operations.

```
// java.util.Objects
static compare(T a, T b, Comparator<T> c) -> int
static equals(Object a, Object b) -> boolean
static deepEquals(Object a, Object b) -> boolean
static hashCode(Object o) -> int
static hash(Object ... values) -> int
static toString(Object o) -> String
static toString(Object o, String nullDefault) -> String

// Check index bound
static checkIndex(int idx, int length) -> int
static checkFromToIndex(int fromIdx, int toIdx, int length) -> int
static checkFromIndexSize(int fromIdx, int size, int length) -> int

// Check null
static isNull(Object o) -> boolean
static nonNull(Object o) -> boolean
static requireNonNull(T obj) -> T
static requireNonNull(T obj, String errmsg) -> T
static requireNonNull(T obj, Supplier<String> msgSupplier) -> T
static requireNonNullElse(T obj, T defaultObj) -> T
static requireNonNullElseGet(T obj, Supplier<T> supplier) -> T
```

2 The Java Collections Framework

2.1 Storing Data Using the Collections Framework

2.1.1 Introducing the Collections Framework

The Collections Framework is the most widely used API of the JDK. Whatever the application you are working on is, odds are that you will need to store and process data in memory at some point. The Java Collections Framework that provides developers with a unified architecture for representing and manipulating collections, enabling them to be manipulated independently of the details of their representation.

The history of data structures goes back nearly as far back as computing itself. The Collections Framework is an implementation of the concepts on how to store, organize, and access data in memory that were developed long before the invention of Java. The Collections Framework does this in a very efficient way, as you are going to see.

The Collections Framework was first introduced in Java SE 2, in 1998 and was rewritten twice since then:

- in Java SE 5 when generics were added;
- in Java 8 when lambda expressions were introduced, along with default methods in interfaces.

These two are the most important updates of the Collections Framework that have been made so far. But in fact, almost every version of the JDK has its set of changes to the Collections Framework.

You will learn in this part are the most useful data structures the Collections Framework has to offer, along with the patterns you will be using to manipulate this data in your application.

The first thing you need to know is that, from a technical point of view, the Collections Framework is a set of interfaces that models different way of storing data in different types of containers. Then the Framework provides at least one implementation for each interface. Knowing these implementations is as important as the interfaces, and choosing the right one depends on what you need to do with it.

2.1.2 Finding Your Way in the Collections Framework

The amount of interfaces and classes in the Collections Framework may be overwhelming at first. Indeed, there are many structures available, both classes and interfaces. Some have self-explanatory names, like **LinkedList**, some are carrying behavior, like **ConcurrentHashMap**, some may sound weird, like **ConcurrentSkipListMap**.

You are going to use some of these elements much more often than others. If you are already familiar with the Java language, you probably came across **List**, **ArrayList** and **Map** already. We focus on the most widely used structures of the Collections Framework, the ones you are going to use daily as a Java developer, and that you need to know and understand best.

That being said, you need to have the big picture of what is there for you in the Collections Framework.

First, the framework consists of interfaces and implementations. Choosing the right interface means you need to know what functions you want to bring to your application. Is what you need consists in:

- storing objects and iterating over them?
- pushing your object to a queue and popping them?
- retrieving them with the use of a key?
- accessing them by their index?
- sorting them?
- preventing the presence of duplicates, or null values?

Choosing the right implementation means you need to know how you are going to use these functionalities:

- Will accessing your objects be done by iterating, or random, indexed access?
- Will the objects be fixed at the launch of your application, and will not change much over its lifetime?
- Will the amount of objects be important with a lot of checking for the presence of certain objects?
- Is the structure you need to store your objects in will be accessed concurrently?

The Collections Framework can give you the right solution to all of these problems.

There are two main categories of interfaces in the Collections Framework: collections and maps.

Collections are about storing objects and iterating over them. The **Collection** interface is the root interface of this category. In fact, the **Collection** interface extends the **Iterable** interface, but this interface is not part of the Collections Framework.

A **Map** stores an object along with a key, which represents that object, just as a primary key represents an object in a database, if you are familiar with this concept. Sometimes you will hear that maps store key/value pairs, which exactly describes what a map does. The **Map** interface is the root interface of this category.

There is no direct relationship between the interfaces of the **Collection** hierarchy and the **Map** hierarchy.

Along with these collections and maps, you also need to know that you can find interfaces to model queues and stacks in the **Collection** hierarchy. **Queues** and stacks are not really about iterating over collections of objects, but since they have been added to the **Collection** hierarchy, it turns out you can do that with them.

There is one last hierarchy that you also need to know, which is the **Iterator** hierarchy. An iterator is an object that can iterate over a collection of objects, and it is part of the Collections Framework.

That makes two main categories, **Collection** and **Map**, a subcategory, **Queue**, and a side category, **Iterator**.

2.1.3 Avoiding Using Old Interfaces and Implementations

The Collections Framework was only introduced in Java 2, meaning that there was a life before it. This life consisted in several classes and interfaces that are still in the JDK, to preserve backward compatibility, but that you should not use anymore in your applications.

Those classes and interfaces are the following:

- **Vector** and **Stack**. The **Vector** class has been retrofitted to implement the List interface. If you are using a vector in a non-concurrent environment, then you can safely replace it with **ArrayList**. The **Stack** class extends **Vector** and should be replaced by **ArrayDeque** in non-concurrent environments.
- The **Vector** class uses the **Enumeration** interface to model its iterator. This interface should not be used anymore: the preferred interface is now the **Iterator** interface.
- **HashTable**: This class has been retrofitted to implement the Map interface. If you are using instances of this class in a non-concurrent environment, then you can safely replace its use with **HashMap**. In a concurrent environment, **ConcurrentHashMap** can be used as a replacement.

2.1.4 Why Choose a Collection Over an Array?

You may be wondering why you should bother learn the Collection Frameworks when you may have the feeling that putting your data in a good old array does the job.

The fact is, in any case, if you have a solution that is simple, that you master well, and that fits your needs, then you should definitely stick with it!

What can a collection do for you, that an array cannot?

- A collection tracks the number of elements it contains
- The capacity of a collection is not limited: you can add (almost) any amount of elements in a collection
- A collection can control what elements you may store in it. For instance, you can prevent null elements to be added
- A collection can be queried for the presence of a given element
- A collection provides operations like intersecting or merging with another collection.

This is just a small sample of what a collection can do for you. In fact, since a collection is an object and given that an object is extensible, you can add any operation you need on most of the collections provided by the JDK. It is not possible to do this with an array, because an array is not an object in Java.

2.2 Getting to Know the Collection Hierarchy

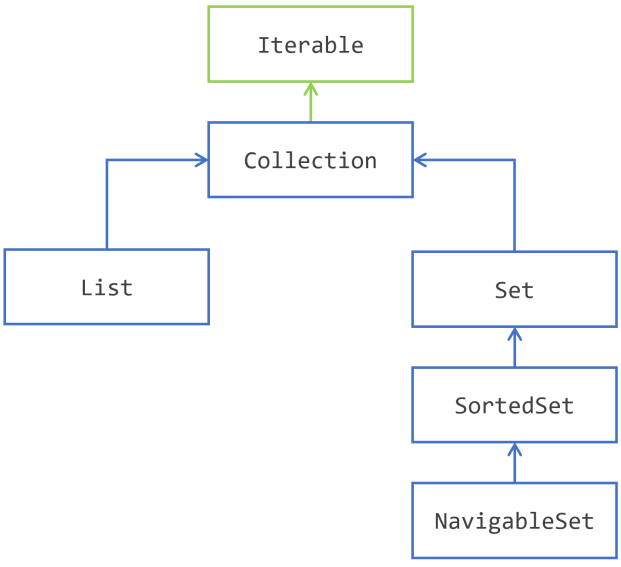
2.2.1 Avoiding Getting Lost in the Collection Hierarchy

The Collection interfaces are divided into two groups. The most basic interface, *java.util*.**Collection**, has the following descendants:

- *java.util*.**Set**
- *java.util*.**SortedSet**
- *java.util*.**NavigableSet**
- *java.util*.**List**
- *java.util*.**Queue**

- *java.util.concurrent.***BlockingQueue**
- *java.util.concurrent.***TransferQueue**
- *java.util.***Deque**
- *java.util.concurrent.***BlockingDeque**

The Collections Framework is divided in several hierarchies of interfaces and classes. The first one you need to understand is the following: the Collection interface hierarchy.



The Collection Interface Hierarchy

Note that some interfaces have been omitted.

2.2.2 The Iterable Interface

The first interface of this hierarchy is the **Iterable** interface, and it is in fact not part of the Collections Framework. It is still worth mentioning here because it is the super interface of the **Collection** interface, and thus of all the interfaces of this hierarchy.

The **Iterable** interface is an addition of Java SE 5 (2004). An object that implements Iterable is an object that you can iterate over. It has been added in Java SE 5 along with the for each pattern of code.

You may be already familiar with this way of iterating over the elements of a **Collection**:

```
Collection<String> collection = ...;

for (String element: collection) {
  // do someting with element
}
```

You may already know that you can iterate over any collection using this pattern, or any array. It turns out that in fact any instance of **Iterable** may be used here.

To implement the **Iterable** interface is really easy: all you need to do is provide an instance of another interface, Iterator, that you are going to see in the following.

2.2.3 Storing Elements in a Container with the Collection Interface

All the other interfaces are about storing elements in containers.

The two interfaces **List** and **Set** both share a common behavior, which is modeled by the Collection interface. The **Collection** interface models several operations on containers of elements. Without diving in the technical details (yet!), here is what you can do with a **Collection**:

- add or remove elements;
- test for the presence of a given element;
- ask for the number of elements contained, or if this collection is empty;
- clear this content.

Since a Collection is a set of elements, there also set operations defined on the **Collection** interface:

- testing for the inclusion of a set in another set;
- union;
- intersection;
- complement.

Lastly, the **Collection** interface also models different ways of accessing its elements:

- you can iterate over the elements of a collection, through the use of an iterator;
- you can create a stream on these elements, that can be parallel.

Of course, all these operations are also available on **List** and **Set**. So what does make the difference between a plain instance of **Collection** and an instance of **Set** or an instance of **List**?

2.2.4 Extending Collection with List

The difference between a **List** of elements and a **Collection** of elements, is that a **List** remembers in what order its elements have been added.

The first consequence is that if you iterate over the elements of a list, the first element you will get is the first that has been added. Then you will get the second one, and so on until all the elements have been seen. So the order you will iterate over the elements is always the same, it is fixed by the order in which these elements have been added. You do not have this guarantee with a plain **Collection** nor for a **Set**.

> It turns out that some implementations of **Set** provided by the Collections Framework happen to always iterate over the elements in the same order. This is an accidental effect, and your code should not rely on this behavior.

There is a second consequence, maybe not as clear as the first one, which is that the elements of a list have an index. Querying a collection for its first element does not make sense. Querying a list for its first element does make sense, since a list does remember that.

How are those indexes handled? Well, once again, this is the responsibility of the implementation. The first role of an interface is to specify a behavior, not to tell how an implementation should achieve that.

As you will see it, the List interface adds new operations to the **Collection** interface. As the elements of a list have an index, you can do the following with that index.

- Get an element at a specific index, or delete it Insert an element or replace an element at a specific position Get a range of elements between two indexes.

2.2.5 Extending Collection with Set

The difference between a Set of elements and a Collection of elements, is that you cannot have duplicates in a Set. You can have several instances of the same class that are equal in a Collection, or even the same instance more than once. This is not allowed in a Set. How is this enforced is the responsibility of the implementation, you will see that later in this tutorial.

One of the consequences of this behavior, is that adding an element to a **Set** may fail.

Then you may ask yourself: can I have a container that prevents having duplicates, and in which elements have an index? The answer is not that simple. The Collections Framework gives you an implementation of Set with which you will iterate over the elements always in the same order, but these elements do not have an index, so this class does not implement **List**.

This difference in behavior does not bring any new operations in the **Set** interface.

2.2.6 Sorting the element of a Set with SortedSet and NavigableSet

The **Set** interface has itself two extensions: **SortedSet** and **NavigableSet**.

The **SortedSet** interface maintains its elements sorted in the ascending order. Once again, how this is enforced is the responsibility of the implementation, as you will see it later.

To be able to sort them, a **SortedSet** needs to compare your elements. How can it achieve that? Well, there are two standard mechanisms defined in the Java language for that.

- Your elements may implement the **Comparable** interface, and provide a $compareTo()$ method
- You give a **Comparator** to the **SortedSet** so that it can compare them.

Even if your elements are **Comparable**, you can still provide a **Comparator** when building a **SortedSet**. This may prove useful if you need to sort your elements in an order that is different from the one implemented in the $compareTo()$ method.

> What is the difference between sorting and ordering? A List keeps its elements in the order they have been added, and a **SortedSet** keeps them sorted. Sorting elements means that the first element you get while traversing a set will be the lowest one, in the sense of a given comparison logic. Ordering elements means that the order in which you added them in a list is kept throughout the life of this list. So the first element you get while traversing a list is the first that has been added to it.

The **SortedSet** adds several operations to **Set**. Here is what you can do with a **SortedSet**.

- You can get the lowest element, and the largest element of the set
- You can extract a headSet and a tailSet of all the elements lesser than, or greater than a given element.

Iterating over the elements of a **SortedSet** will be made from the lowest element to the greatest.

The **NavigableSet** does not change the behavior of a **SortedSet**. It adds several vey useful operations on **SortedSet**, among them the possibility to iterate over the elements in the descending order. You will see more details ont that later.

2.3 Storing Elements in a Collection

2.3.1 Exploring the Collection Interface

The first interface you need to know is the **Collection** interface. It models a plain collection, which can store elements and gives you different ways to retrieve them.

If you want to run the examples in this part, you need to know how to create a collection. We have not covered the **ArrayList** class yet, we will do that later.

2.3.2 Methods That Handle Individual Elements

Let us begin by storing and removing an element from a collection. The two methods involved are $add()$ and $remove()$.

- $add(element)$: adds an element in the collection. This method returns a boolean in case the operation failed. You saw in the introduction that it should not fail for a List, whereas it may fail for a Set, because a set does not allow duplicates.
- $remove(element)$: removes the given element from the collection. This method also returns a $boolean$, because the operation may fail. A remove may fail, for instance, when the item requested for removal is not present in the collection

You can run the following example. Here, you create an instance of the Collection interface using the **ArrayList** implementation. The generics used tells the Java compiler that you want to store String objects in this collection. **ArrayList** is not the only implementation of **Collection** you may use. More on that later.

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
System.out.println("strings = " + strings);
strings.remove("one");
System.out.println("strings = " + strings);
```

Running the previous code should print the following:

Command window

```
strings = [one, two]
strings = [two]
```

You can check for the presence of an element in a collection with the *contains*() method. Note that you can check the presence of any type of element. For instance, it is valid to check for the presence of a User object in a collection of *String*. This may seem odd, since there is no chance that this check returns true, but it is allowed by the compiler. If you are using an IDE to test this code, your IDE may warn about testing for the presence of a User object in a collection of *String* objects.

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
if (strings.contains("one")) {
  System.out.println("one is here");
}
if (!strings.contains("three")) {
  System.out.println("three is not here");
}

User rebecca = new User("Rebecca");
if (!strings.contains(rebecca)) {
  System.out.println("Rebecca is not here");
}
```

Running this code produces the following:

Command window

```
one is here
three is not here
Rebecca is not here
```

2.3.3 Methods That Handle Other Collections

This first set of methods you saw allows you to handle individual elements.

There are four such methods: *containsAll*(), *addAll*(), *removeAll*() and *retainAll*(). They define the four fundamental operations on set of objects.

- *containsAll*(): defines the inclusion

- $addAll()$: defines the union
- $removeAll()$: defines the complement
- $retainAll()$: defines the intersection.

The first one is really simple: $containsAll()$ takes another collection as an argument and returns true if all the elements of the other collections are contained in this collection. The collection passed as an argument does not have to be the same type as this collection: it is legal to ask if a collection of $String$, of type **Collection<String>** is contained in a collection of **User**, of type **Collection<User>**.

Here is an example of the use of this method:

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Collection<String> first = new ArrayList<>();
strings.add("one");
strings.add("two");

Collection<String> second = new ArrayList<>();
strings.add("one");
strings.add("four");

System.out.println("Is first contained in strings? "
  + strings.containsAll(first));
System.out.println("Is second contained in strings? "
  + strings.containsAll(second));
```

Running this code produces the following:

Command window

```
Is first contained in strings? true
Is second contained in strings? false
```

The second one is $addAll()$. It allows you to add all the elements of a given collection to this collection. As with the $add()$ method, this may fail for some elements in some cases. This method returns true if this collection has been modified by this call. This is an important point to understand: getting a true value does not mean that all the elements of the other collection have been added; it means that at least one has been added.

You can see *removeAll()* in action on the following example:

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Collection<String> first = new ArrayList<>();
first.add("one");
first.add("four");

boolean hasChanged = strings.addAll( first );

System.out.println("Has strings changed? " + hasChanged);
System.out.println(" strings = " + strings);
```

Running this code produces the following result:

Command window

```
Has strings changed? true
strings = [one, two, three, one, four]
```

You need to be aware that running this code will produce a different result if you change the implementation of Collection. This result stands for **ArrayList**, as you will see in the following, it would not be the same for **HashSet**.

The third one is *removeAll()*. It removes all the elements of this collection that are contained in the other collection. Just as it is the case for *contains()* or *remove()*, the other collection can be defined on any type; it does not have to be compatible with the one of this collection.

You can see *addAll()* in action on the following example:

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Collection<String> toBeRemoved = new ArrayList<>();
toBeRemoved.add("one");
toBeRemoved.add("four");
```

```
boolean hasChanged = strings.removeAll(toBeRemoved);

System.out.println("Has strings changed? " + hasChanged);
System.out.println(" strings = " + strings);
```

Running this code produces the following result:

Command window

```
Has strings changed? true
 strings = [two, three]
```

The last one is *retainAll()*. This operation retains only the elements from this collection that are contained in the other collection; all the others are removed. Once again, as it is the case for *contains()* or *remove()*, the other collection can be defined on any type.

You can see *retainAll()* in action on the following example:

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Collection<String> toBeRetained = new ArrayList<>();
toBeRetained.add("one");
toBeRetained.add("four");

boolean hasChanged = strings.retainAll(toBeRetained);

System.out.println("Has strings changed? " + hasChanged);
System.out.println(" strings = " + strings);
```

Running this code produces the following result:

Command window

```
Has strings changed? true
 strings = [one]
```

2.3.4 Methods That Handle The Collection Itself

Then the last batch of methods deal with the collection itself.

You have two methods to check the content of a collection.

- $size()$: Returns the number of elements in a collection, as an int.
- $isEmpty()$: Tells you if the given collection is empty or not.

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
if (!strings.isEmpty()) {
  System.out.println("Indeed strings is not empty!");
}
System.out.println("The number of elements in strings is "
  + strings.size());
```

Command window

```
Indeed strings is not empty!
The number of elements in strings is 2
```

Then you can delete the content of a collection by simply calling $clear()$ on it.

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
System.out.println("The number of elements in strings is "
  + strings.size());
strings.clear();
System.out.println("After clearing it, this number is now "
  + strings.size());
```

Command window

```
The number of elements in strings is 2
After clearing it, this number is now 0
```

2.3.5 Getting an Array of the Elements of a Collection

Even if storing your elements in a collection may make more sense in your application than putting them in an array, there are still cases where getting them in an array is something you will need.

The Collection interface gives you three patterns to get the elements of a collection in an array, in the form of three overloads of a $toArray()$ method.

The first one is a plain $toArray()$ call, with no arguments. This returns your elements in an array of plain objects.

This may not be what you need. If you have a **Collection<String>**, what you could prefer is an array of String. You can still cast Object[] to String[], but there is no guarantee that this cast will not fail at runtime. If you need type safety, then you can call either of the following methods.

- $toArray(T[]\ tab)$ returns an array or T: T[]
- $toArray(IntFunction<T[]>\ generator)$, returns the same type, with a different syntax.

What are the differences between the last two patterns? The first one is readability. Creating an instance of **IntFunction<T[]>** may look weird at first, but writing it with a method reference is really a no brainer.

Here is the first pattern. In this first pattern, you need to pass an array of the corresponding type.

```
// Suppose you have 15 elements in that collection
Collection<String> strings = ...;

// You can pass an empty array
String[] tabString1 = strings.toArray(new String[] {});

// or an array of the right size
String[] tabString2 = strings.toArray(new String[15]);
```

What is the use of this array passed as an argument? If it is big enough to hold all the elements of the collection, then these elements will be copied in the array, and it will be returned. If there is more room in the array than needed, then first of the unused cell of the array will be set to null. If the array you pass is too small, then a new array of the exact right size is created to hold the elements of the collection.

Here is this pattern in action:

```
Collection<String> strings = List.of("one", "two");

String[] largerTab = {"three", "three", "three", "I", "was", "there"};
System.out.println("largerTab = " + Arrays.toString(largerTab));

String[] result = strings.toArray(largerTab);
System.out.println("result = " + Arrays.toString(result));

System.out.println("Same arrays? " + (result == largerTab));
```

Running the previous code will give you:

Command window

```
largerTab = [three, three, three, I, was, there]
result = [one, two, null, I, was, there]
Same arrays? true
```

You can see that the array was copied in the first cells of the argument array, and null was added right after it, thus leaving the last elements of this array untouched. The returned array is the same array as the one you gave as an argument, with a different content.

Here is a second example, with a zero-length array:

```
Collection<String> strings = List.of("one", "two");

String[] zeroLengthTab = {};
String[] result = strings.toArray(zeroLengthTab);

System.out.println("zeroLengthTab = " + Arrays.toString(zeroLengthTab));
System.out.println("result = " + Arrays.toString(result));
```

Command window

```
zeroLengthTab = []
result = [one, two]
```

A new array has been created in this case.

The second pattern is written using a constructor method reference to implement **IntFunction**<**T**[]>:

```
Collection<String> strings = ...;

String[] tabString3 = strings.toArray(String[]::new);
```

In that case, a zero-length array of the right type is created with this function, and this method then calls to $toArray()$ with this array passed as an argument.

This pattern of code was added in JDK 8 to improve the readability of the $toArray()$ calls.

2.3.6 Filtering out Elements of a Collection with a Predicate

Java SE 8 added a new feature the Collection interface: the possibility to filter out elements of a collection with a predicate.

Suppose you have a **List**<**String**> and you need to remove all the null strings, the empty strings and the strings longer than 5 characters. In Java SE 7 and earlier, you can use the **Iterator**.$remove()$ method to do that, calling it in an if statement. You will see this pattern along with the Iterator interface. With $removeIf()$, your code becomes much simpler:

```
Predicate<String> isNull = Objects::isNull;
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNullOrEmpty = isNull.or(isEmpty);

Collection<String> strings = new ArrayList<>();
strings.add(null);
strings.add("");
strings.add("one");
strings.add("two");
strings.add("");
strings.add("three");
strings.add(null);

System.out.println("strings = " + strings);
strings.removeIf(isNullOrEmpty);
System.out.println("filtered strings = " + strings);
```

Running this code produces the following result:

```
Command window
strings = [null, , one, two, , three, null]
filtered strings = [one, two, three]
```

Once again, using this method will greatly improve the readability and expressiveness of your application code.

2.3.7 Choosing an Implementation for the Collection Interface

In all these examples, we used **ArrayList** to implement the **Collection** interface.

The fact is: the Collections Framework does not provide a direct implementation of the **Collection** interface. **ArrayList** implements **List**, and because **List** extends **Collection**, it also implements **Collection**.

If you decide to use the **Collection** interface to model the collections in your application, then choosing **ArrayList** as you default implementation is your best choice, most of the time. You will see more discussions on the right implementation to choose in later in this tutorial.

2.4 Iterating over the Elements of a Collection

2.4.1 Using the for-each Pattern

Your simplest choice to iterate over the elements of a collection is to use the for-each pattern.

```
Collection<String> strings = List.of("one", "two", "three");

for (String element: strings) {
  System.out.println(string);
}
```

This pattern is very efficient, as long as you only need to read the elements of your collection. The Iterator pattern allows to remove some of the elements of your collection while you are iterating over them. If you need to do that, then you want to use the Iterator pattern.

2.4.2 Using an Iterator on a Collection

Iterating over the elements of a collection uses a special object, an instance of the Iterator interface. You can get an **Iterator** object from any extension of the **Collection** interface. The $iterator()$ method is defined on the **Iterable** interface, extended by the **Collection** interface, and further extended by all the interfaces of the collection hierarchy.

Iterating over the elements of a collection using this object is a two-steps process.

1. First you need to check if there are more elements to be visited with the $hasNext()$ method
2. Then you can advance to the next element with the $next()$ method.

If you call the $next()$ method but there are no more elements in the collection, you will get a **NoSuchElementException**. Calling $hasNext()$ is not mandatory, it is there to help you to make sure that there is indeed a next element.

Here is the pattern:

```
Collection<String> strings = List.of("one", "two", "three", "four");
for (Iterator<String> iterator = strings.iterator(); iterator.hasNext();) {
  String element = iterator.next();
  if (element.length() == 3) {
    System.out.println(element);
  }
}
```

This code produces the following result:

Command window

```
one
two
```

The **Iterator** interface has a third method: $remove()$. Calling this method removes the current element from the collection. There are cases though where this method is not supported, it will throw an **UnsupportedOperationException**. Quite obviously, calling $remove()$ on an immutable collection cannot work, so this is one of the cases. The implementation of Iterator you get from **ArrayList**, **LinkedList** and **HashSet** all support this remove operation.

2.4.3 Updating a Collection While Iterating over It

If you happen to modify the content of a collection while iterating over it, you may get a **ConcurrentModificationException**. Getting this exception may be a little confusing, because this exception is also used in concurrent programming. In the context of the Collections Framework, you may get it without touching multithreaded programming.

The following code throws a ConcurrentModificationException.

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Iterator<String> iterator = strings.iterator();
while (iterator.hasNext()) {

    String element = iterator.next();
    strings.remove(element);
}
```

If what you need is to remove the elements of a collection that satisfy a given criteria, you may use the $removeIf()$ method.

2.4.4 Implementing the Iterable Interface

Now that you saw what an iterator is in the Collection Framework, you can create a simple implementation of the **Iterable** interface.

Suppose you need to create a Range class that models a range of integers between two limits. All you need to do is iterate from the first integer to the last one.

You can implement the **Iterable** interface with a record, a feature introduced in Java SE 16:

```
record Range(int start, int end) implements Iterable<Integer> {
    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            private int index = start;

            @Override
            public boolean hasNext() {
```

```
        return index < end;
      }

      @Override
      public Integer next() {
        if (index > end) {
          throw new NoSuchElementException("" + index);
        }
        int currentIndex = index;
        index++;
        return currentIndex;
      }
    };
  }
}
```

You can do the same with a plain class, in case your application does not support Java SE 16 yet. Note that the code of the implementation of **Iterator** is exactly the same.

```
class Range implements Iterable<Integer> {
  private final int start;
  private final int end;

  public Range(int start, int end) {
    this.start = start;
    this.end = end;
  }

  @Override
  public Iterator<Integer> iterator() {
    return new Iterator<>() {
      private int index = start;

      @Override
      public boolean hasNext() {
        return index < end;
      }

      @Override
      public Integer next() {
        if (index > end) {
          throw new NoSuchElementException("" + index);
        }
        int currentIndex = index;
        index++;
```

```
            return currentIndex;
        }
    };
  }
}
```

In both cases, you can use an instance of Range in a for-each statement, since it implements **Iterable**:

```
for (int i : new Range(0, 5)) {
    System.out.println("i = " + i);
}
```

This code produces the following result:

Command window

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

2.5 Extending Collection with List

2.5.1 Exploring the List Interface

The List interface brings two new functionalities to plain collections.

- This order in which you iterate over the elements of a list is always the same, and it respects the order in which the elements have been added to this list.
- The elements of a list have an index.

2.5.2 Choosing your Implementation of the List Interface

While the Collection interface as no specific implementation in the Collections Framework (it relies on the implementations of its sub-interfaces), the List interface

has 2: **ArrayList** and **LinkedList**. As you may guess, the first one is built on an internal array, and the second on a doubly-linked list.

Is one of these implementation better than the other? If you are not sure which one to choose, then your best choice is probably **ArrayList**.

What was true for linked lists when computing was invented in the 60's does not hold anymore, and the capacity of linked lists to outperform arrays on insertion and deletion operations is greatly diminished by modern hardware, CPU caches, and pointer chasing. Iterating over the elements of an ArrayList is much faster that over the elements of a **LinkedList**, mainly due to pointer chasing and CPU cache misses.

There are still cases where a linked list is faster than an array. A doubly-linked list can access its first and last element faster than an **ArrayList** can. This is the main use case that makes LinkedList better than **ArrayList**. So if your application needs a Last In, First Out (LIFO, covered later in this tutorial) stack, or a First In, First Out (FIFO, also covered later) waiting queue, then choosing a linked list is probably your best choice.

On the other hand, if you plan to iterate through the elements of your list, or to access them randomly by their index, then the ArrayList is probably your best bet.

2.5.3 Accessing the Elements Using an Index

The **List** interface brings several methods to the **Collection** interface, that deal with indexes.

Accessing a Single Object

- $add(index, element)$: inserts the given object at the index, adjusting the index if there are remaining elements
- $get(index)$: returns the object at the given index
- $set(index, element)$: replaces the element at the given index with the new element
- $remove(index)$: removes the element at the given index, adjusting the index of the remaining elements.

Calling these methods work only for valid indexes. If the given index is not valid then an **IndexOutOfBoundsException** exception will be thrown.

Finding the Index of an Object

The methods $indexOf(element)$ and $lastIndexOf(element)$ return the index of the given element in the list, or -1 if the element is not found.

Getting a SubList

The *subList*(*start*, *end*) returns a list consisting of the elements between indexes start and end - 1. If the indexes are invalid then an **IndexOutOfBoundsException** exception will be thrown.

Note that the returned list is a view on the main list. Thus, any modification operation on the sublist is reflected on the main list and vice-versa.

For instance, you can clear a portion of the content of a list with the following pattern:

```
1 List<String> strings = new ArrayList<>(List.of("0", "1", "2", "3", "4", "5"));
  System.out.println(strings);
3 strings.subList(2, 5).clear();
  System.out.println(strings);
```

This code produces the following result:

Command window

```
  [0, 1, 2, 3, 4, 5]
2 [0, 1, 5]
```

Inserting a Collection

The last pattern of this list is about inserting a collection at a given indexes: *addAll*(*int index*, *Collection collection*).

2.5.4 Sorting the Elements of a List

A list keeps its elements in a known order. This is the main difference with a plain collection. So it makes sense to sort the elements of a list. This is the reason why a *sort*() method has been added to the **List** interface in JDK 8.

In Java SE 7 and earlier, you could sort the elements of your **List** by calling **Collections**.*sort*() and pass you list as an argument, along with a comparator if needed.

Starting with Java SE 8 you can call *sort*() directly on your list and pass your comparator as an argument. There is no overload of this method that does not take any argument. Calling it with a null comparator will assume that the elements of your **List** implement **Comparable**, you will get a **ClassCastException** if this is not the case.

If you do not like calling methods will null arguments (and you are right!), you can still call it with **Comparator**.*naturalOrder*() to achieve the same result.

2.5.5 Iterating over the Elements of a List

The **List** interface gives you one more way to iterate over its elements with the **ListIterator**. You can get such an iterator by calling *listIterator*(). You can call this method with no argument, or pass an integer index to it. In that case, the iteration will start at this index.

The **ListIterator** interface extends the regular **Iterator** that you already know. It adds several methods to it.

- *hasPrevious*() and *previous*(): to iterate in the descending order rather than the ascending order
- *nextIndex*() and *previousIndex*(): to get the index of the element that will be returned by the next *next*() call, or the next *previous*() call
- *set*(*element*): to update the last element returned by *next*() or *previous*(). If neither of these methods have been called on this iterator then an **IllegalStateException** is raised.

Let us see this *set*() method in action:

```
// Create list of strings
List<String> numbers = Arrays.asList("one", "two", "three");
// Using set() method to update an element
for (ListIterator<String> it = numbers.listIterator(); it.hasNext();) {
   String nextElement = it.next();
   if (Objects.equals(nextElement, "two")) {
      it.set("2");
   }
}

System.out.println("numbers = " + numbers);
```

This code produces the following result:

Command window

```
numbers = [one, 2, three]
```

2.6 Extending Collection with Set, SortedSet and NavigableSet

2.6.1 Exploring the Set Interface

The **Set** interface does not bring any new method to the **Collection** interface. The Collections Framework gives you one plain implementation of the **Set** interface: **HashSet**. Internally, a **HashSet** wraps an instance of **HashMap**, a class that will be covered later, that acts as a delegate for **HashSet**.

As you already saw, what a Set brings to a **Collection** is that it forbids duplicates. What you lose over the **List** interface is that your elements are stored in no particular order. There is very little chance that you will iterate over them in the same order as you added them to your set.

You can see this in the following example:

```
List<String> strings = List.of("one", "two", "three", "four", "five");
Set<String> set = new HashSet<>();
set.addAll(strings);
set.forEach(System.out::println);
```

Command window

```
four
one
two
three
five
```

Some implementations of **Set** give you the same order when you iterate over their elements, but since this is not guaranteed, your code should not rely on that.

2.6.2 Extending Set with SortedSet

The first extension of Set is the **SortedSet** interface. The **SortedSet** interface keeps its elements sorted according to a certain comparison logic. The Collections Framework gives you one implementation of **SortedSet**, called **TreeSet**.

As you already saw, either you provide a comparator when you build a **TreeSet**,

or you implement the Comparable interface for the elements you put in the **TreeSet**. If you do both, then the comparator takes precedence.

The **SortedSet** interface adds new methods to **Set**.

- $first()$ and $last()$ returns the lowest and the largest elements of the set
- $headSet(toElement)$ and $tailSet(fromElement)$ returns you subsets containing the elements lower than $toElement$ or greater than $fromElement$
- $subSet(fromElement, toElement)$ gives you a subset of the element between $fromElement$ and $toElement$.

The $toElement$ and $fromElement$ do not have to be elements of the main set. If they are, then $toElement$ is not included in the result and $fromElement$ is, following the usual convention.

Consider the following example:

```
SortedSet<String> strings = new TreeSet<>(Set.of("a", "b", "c", "d", "e", "f"));
SortedSet<String> subSet = strings.subSet("aa", "d");
System.out.println("sub set = " + subSet);
```

Command window

```
sub set = [b, c]
```

The three subsets that these methods return are views on the main set. No copy is made, meaning that any change you make to these subsets will be reflected in the set, and the other way round.

You can remove or add elements to the main set through these subsets. There is one point you need to keep in mind though. These three subsets remember the limits on which they have been built. For consistency reasons, it is not legal to add an element through a subset outside its limits. For instance, if you take a $headSet$ and try to add an element greater than $toElement$, then you will get an **IllegalArgumentException**.

2.6.3 Extending SortedSet with NavigableSet

Java SE 6 saw the introduction of an extension of **SortedSet** with the addition of more methods. It turns out that the **TreeSet** class was retrofitted to implement **NavigableSet**. So you can use the same class for both interfaces.

Some methods are overloaded by **NavigableSet**.

- $headSet()$, $tailSet()$, and $subSet()$ may take a further boolean arguments to specify whether the limits ($toElement$ or $fromElement$) are to be included in the resulting subset.

Other methods have been added.

- $ceiling(element)$, and $floor(element)$ return the greatest element lesser or equal than, or the lowest element greater or equal than the provided element. If there is no such element then null is returned
- $lower(element)$, and $higher(element)$ return the greater element lesser than, or the lowest element greater than the provided element. If there is no such element then null is returned.
- $pollFirst()$, and $pollLast()$ return and removes the lowest or the greatest element of the set.

Furthermore, **NavigableSet** also allows you to iterate over its elements in descending order. There are two ways to do this.

- You can call $descendingIterator()$: it gives you a regular Iterator that traverses the set in the descending order.
- You can also call $descendingSet()$. What you get in return is another **NavigableSet** that is a view on this set and that makes you think you have the same set, sorted in the reversed order.

The following example demonstrates this.

```
1 NavigableSet<String> sortedStrings = new TreeSet<>(Set.of("a", "b", "c", "d", "e",
      ↪ "f"));
  System.out.println("sorted strings = " + sortedStrings);
3 NavigableSet<String> reversedStrings = sortedStrings.descendingSet();
  System.out.println("reversed strings = " + reversedStrings);
```

Command window

```
  sorted strings = [a, b, c, d, e, f]
2 reversed strings = [f, e, d, c, b, a]
```

2.7 Creating and Processing Data with the Collections Factory Methods

2.7.1 Creating Immutable Collections

Java SE 9 saw the addition of a set of factory methods to the **List** and **Set** interfaces to create lists and sets. The pattern is very simple: just call the **List**.*of*() or **Set**.*of*() static method, pass the elements of your list and set, and that's it.

```
List<String> stringList = List.of("one", "two", "three");
Set<String> stringSet = Set.of("one", "two", "three");
```

Several points are worth noting though.

- The implementation you get in return may vary with the number of elements you put in your list or set. None of them is **ArrayList** or **HashSet**, so your code should not rely on that.
- Both the list and the set you get are immutable structures. You cannot add or modify elements in them, and you cannot modify these elements. If the objects of these structures are mutable, you can still mutate them.
- These structures do not accept null values. If you try to add a null value in such a list or set, you will get an exception.
- The Set interface does not allow duplicates: this is what a set is about. Because it would not make sense to create such a set with duplicate values, it is assumed that writing such a code is a bug. So you will get an exception if you try to do that.
- The implementations you get are **Serializable**.

These *of*() methods are commonly referred to as convenience factory methods for collections.

2.7.2 Getting an Immutable Copy of a Collection

Following the success of the convenience factory methods for collections, another set of convenience methods have been added in Java SE 10 to create immutable copies of collections.

There are two of them: **List**.*copyOf*() and **Set**.*copyOf*(). They both follow the same pattern:

```
Collection<String> strings = Arrays.asList("one", "two", "three");

List<String> list = List.copyOf(strings);
Set<String> set = Set.copyOf(strings);
```

In all cases, the collection you need to copy should not be null and should not contain any null elements. If this collection has duplicates, only one of these elements will be kept in the case of **Set**.*copyOf*().

What you get in return is an immutable copy of the collection passed as an argument. So modifying this collection will not be reflected in the list or set you get as a copy.

None of the implementations you get accept null values. If you try to copy a collection with null values, you will get a **NullPointerException**.

2.7.3 Wrapping an Array in a List

The Collections Framework has a class called Arrays with about 200 methods to handle arrays. Most of them are implementing various algorithms on arrays, like sorting, merging, searching, and are not covered in this section.

There is one though that is worth mentioning: **Arrays**.*asList*(). This method takes a *vararg* as an argument and returns a List of the elements you passed, preserving their order. This method is not part of the convenience factory methods for collections but is still very useful.

This **List** acts as a wrapper on an array, and behaves in the same way, which maybe a little confusing at first. Once you have set the size of an array, you cannot change it. It means that you cannot add an element to an existing array, nor can you remove an element from it. All you can do is replace an existing element with another one, possibly null.

The List you get by calling **Arrays**.*asList*() does exactly this.

- If you try to add or remove an element, you will get an UnsupportedOperationException, whether you do that directly or through the iterator.
- Replacing existing elements is OK.

So this list is not immutable, but there are restrictions on how you can change it.

2.7.4 Using the Collections Factory Class to Process a Collection

The Collections Framework comes with another factory class: **Collections**, with a set of method to manipulate collections and their content. There are about 70

methods in this class, it would be tedious so examine them one-by-one, so let us present a subset of them.

Extracting the Minimum or the Maximum from a Collection

The **Collections** class give you two methods for that: the min() and the $max()$. Both methods take the collection as an argument from which the min or the max is extracted. Both methods have an overload that also takes a comparator as a further argument.

If no comparator is provided then the elements of the collection must implement Comparable. If not, a **ClassCastException** will be raised. If a comparator is provided, then it will be used to get the min or the max, whether the elements of the collection are comparable or not.

Getting the min or the max of an empty collection with this method will raise a NoSuchMethodException.

Finding a Sublist in a List

Two methods locate a given sublist in a bigger list:

- $indexOfSublist(List<?>\ source, List<?>\ target)$: returns the first index of the first element of the target list in the source list, or -1 if it does not exist;
- $lastIndexOfSublist(List<?>\ source, List<?>\ target)$: return the last of these indexes.

Changing the Order of the Elements of a List

Several methods can change the order of the elements of a list:

- $sort()$ sorts the list in place. This method may take a comparator as an argument. As usual, if no comparator is provided, then the elements of the list must be comparable. If a comparator is provided, then it will be used to compare the elements. Starting with Java SE 8, you should favor the $sort()$ method from the **Listinterface**.
- $shuffle()$ randomly shuffles the elements of the provided list. You can provide yout instance of **Random** if you need a random shuffling that you can repeat.
- $rotate()$ rotates the elements of the list. After a rotation the element at index 0 will be found at index 1 and so on. The last elements will be moved to the first place of the list. You can combine $subList()$ and $rotate()$ to remove an element at a given index and to insert it in another place in the list. This can be done with the following code:

```
List<String> strings = Arrays.asList("0", "1", "2", "3", "4");
System.out.println(strings);
int fromIndex = 1, toIndex = 4;
Collections.rotate(strings.subList(fromIndex, toIndex), -1);
System.out.println(strings);
```

Command window

```
[0, 1, 2, 3, 4]
[0, 2, 3, 1, 4]
```

The element at index $fromIndex$ has been removed from its place, the list has been reorganized accordingly, and the element has been inserted at index $toIndex$ - 1.

- $reverse()$: reverse the order of the elements of the list.
- $swap()$: swaps two elements from the list. This method can take a list as an argument, as well as a plain array.

Wrapping a Collection in an Immutable Collection

The **Collections** factory class gives you several methods to create immutable wrappers for your collections or maps. The content of the structure is not duplicated; what you get is a wrapper around your structure. All the attempts to modify it will raise exceptions.

All these methods starts with unmodifiable, followed by the name of the type of your structure. For instance, to create an immutable wrapper of a list, you can call:

```
List<String> strings = Arrays.asList("0", "1", "2", "3", "4");
List<String> immutableStrings = Collections.unmodifiableList(strings);
```

Just a word of warning: it is not possible to modify your collection through this wrapper. But this wrapper is backed by your collection, so if you modify it by another means, this modification will be reflected in the immutable collection. Let us see that in the following code:

```java
List<String> strings = new ArrayList<>(Arrays.asList("0", "1", "2", "3", "4"));
List<String> immutableStrings = Collections.unmodifiableList(strings);
System.out.println(immutableStrings);
strings.add("5");
System.out.println(immutableStrings);
```

Command window

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
```

If you plan to create an immutable collection using this pattern, defensively copying it first may be a safe precaution.

Wrapping a Collection in a Synchronized Collection

In the same way as you can create immutable wrappers for your maps and collections, the **Collections** factory class can create synchronized wrappers for them. The patterns follow the same naming convention as the names for methods that create immutable wrappers: the methods are called synchronized followed by **Collection**, **List**, **Set**, etc...

There are two precautions you need to follow.

- All the accesses to your collection should be made through the wrapper you get
- Traversing your collection with an iterator or a stream should be synchronized by the calling code on the list itself.

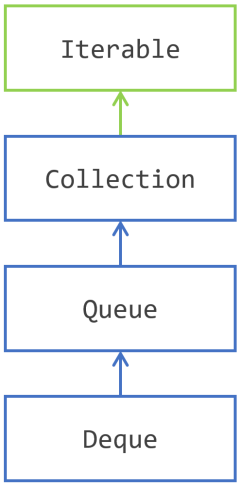
Not following these rules will expose your code to race conditions.

Synchronizing collections using the **Collections** factory methods may not be your best choice. The Java Util Concurrent framework has better solutions to offer.

2.8 Storing Elements in Stacks and Queues

2.8.1 Finding Your Way in the Queue Hierarchy

Java SE 5 saw the addition of a new interface in the Collections Framework: the **Queue** interface, further extended in Java SE 6 by the **Deque** interface. The **Queue** interface is an extension of the **Collection** interface.



The Queue Interface Hierarchy

2.8.2 Pushing, Popping and Peeking

The stack and queue structures are classic data structures in computing. Stacks are also called LIFO stacks, where LIFO stands for Last In, First Out. Queues are known as FIFO: First In First Out.

These structures are very simple and gives you three main operations.

- $push(element)$: adds an element to the queue, or the stack
- $pop()$: removes an element from the stack, that is, the youngest element added
- $poll()$: removes an element from the queue, that is, the oldest element added
- $peek()$: allows you to see the element you will get with a $pop()$ or a $poll()$, but without removing it from the queue of the stack.

There are two reasons to explain the success of these structures in computing. The first one is their simplicity. Even in the very early days of computing, implementing these was simple. The second one is their usefulness. Many algorithms use stacks for their implementations.

2.8.3 Modeling Queues and Stacks

The Collections Framework gives you two interfaces to model queues and stacks:

- the **Queue** interface models a queue;

- the **Deque** interface models a double ended queue (thus the name). You can push, pop, poll and peek elements on both the tail and the head of a **Deque**, making it both a queue and a stack.

Stacks and queues are also widely used in concurrent programming. These interfaces are further extended by more interfaces, adding methods useful in this field. These interfaces, **BlockingQueue**, **BlockingDeque** and **TransferQueue**, are at the intersection of the Collections Framework and concurrent programming in Java, which is outside the scope of this tutorial.

Both the **Queue** and the **Deque** interfaces add behavior to these three fundamental operations to deal with two corner cases.

- A queue may be full and not able to accept more elements
- A queue may be empty and cannot return an element with a *pop*, *poll*, nor the *peek* operation.

In fact this question needs to be answered: how does an implementation should behave in these two cases?

2.8.4 Modeling FIFO Queues with Queue

The **Queue** interface gives you two ways of dealing with these corner cases. An exception can be thrown, or a special value can be returned.

Here is the table of the methods **Queue** gives you.

Operation	Method	Behavior when the queue is full or empty
push	*add*(*element*)	throws an IllegalStateException
	offer(*element*)	returns false
poll	*remove*()	throws a NoSuchElementException
	poll()	returns false
peek	*element*()	throws a NoSuchElementException
	peek()	returns null

2.8.5 Modeling LIFO Stacks and FIFO Queues with Deque

Java SE 6 added the **Deque** interface as an extension of the **Queue** interface. Of course, the methods defined in **Queue** are still available in **Deque**, but **Deque** brought a new naming convention. So these methods have been duplicated in **Deque**, following this new naming convention.

Here is the table of the methods defined in **Deque** for the FIFO operations.

Operation	Method	Behavior when the queue is full or empty
push	$addLast(element)$	throws an IllegalStateException
	$offerLast(element)$	returns false
poll	$removeFirst()$	throws a NoSuchElementException
	$pollFirst()$	returns null
peek	$getFirst()$	throws a NoSuchElementException
	$peekFirst()$	returns null

And here is the table of the methods defined in **Deque** for the LIFO operations.

Operation	Method	Behavior when the queue is full or empty
push	$addFirst(element)$	throws an IllegalStateException
	$offerFirst(element)$	returns false
poll	$removeFirst()$	throws a NoSuchElementException
	$pollFirst()$	returns null
peek	$getFirst()$	throws a NoSuchElementException
	$peekFirst()$	returns null

The **Deque** naming convention is straightforward and is the same as the one followed in the **Queue** interface. There is one difference though: the peek operations are named $getFirst()$ and $getLast()$ in **Deque**, and $element()$ in **Queue**.

Moreover, **Deque** also defines the methods you would expect in any queue or stack class:

- $push(element)$: adds the given element to the head of the double ended queue
- $pop()$: removes and return the element at the head of the double ended queue
- $poll()$: does the same at the tail of the double ended queue
- $peek()$: shows you the element at the tail of the double ended queue.

In case there is no element to *pop*, *poll*, or *peek*, then a null value is returned by these methods.

2.8.6 Implementing Queue and Deque

The Collections Framework gives you three implementations of **Queue** and **Deque**, outside the concurrent programming space:

- **ArrayDeque**: which implements both. This implementation is backed by an array. The capacity of this class automatically grows as elements are added. So this implementation always accepts new elements.
- **LinkedList**: which also implements both. This implementation is backed by a linked list, making the access to its first and last element very efficient. A **LinkedList** will always accept new elements.
- **PriorityQueue**: that only implements **Queue**. This queue is backed by an array that keeps its elements sorted by their natural order or by an order specified by a **Comparator**. The head of this queue is always the least element of the queue with respect to the specified ordering. The capacity of this class automatically grows as elements are added.

2.8.7 Staying Away from the Stack Class

It may seem tempting to use the **Stack** class offered by the JDK. This class is simple to use and to understand. It has the three expected methods $push(element)$, $pop()$, and $peek()$, and seeing this class referenced in your code makes it perfectly readable.

It turns out that this class is an extension of the **Vector** class. Back in the days before the Collections Framework was introduced, **Vector** was your best choice to work with a list. Although **Vector** is not deprecated, its usage is discouraged. So is the usage of the **Stack** class.

The **Vector** class is thread safe, and so is **Stack**. If you do not need the thread safety, then you can safely replace its usage with **Deque** and **ArrayDeque**. If what you need is a thread-safe stack, then you should explore the implementations of the **BlockingQueue** interface.

2.9 Using Maps to Store Key Value Pairs

2.9.1 Introducing the Map Hierarchy

The second main structure offered by the Collections Framework is an implementation of a very classic data structure: the hashmap structure. This concept is not new and is fundamental in structuring data, whether in-memory or not. How does it work and how has it been implemented in the Collections Framework?

A hashmap is a structure able to store key-value pairs. The value is any object your application needs to handle, and a key is something that can represent this object.

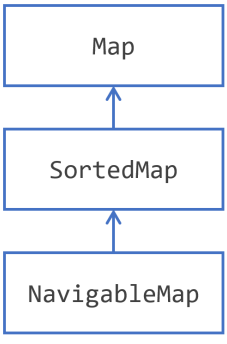
Suppose you need to create an application that has to handle invoices, represented by instances of an Invoice class. Then your values are these **Invoice** instances, and your keys could be the invoice numbers. Each invoice has a number, and that number is unique among all your invoices.

Generally speaking, each value is bound to a key, just as an invoice is bound to its invoice number. If you have a given key, you can retrieve the value. Usually a key is a simple object: think of a string of several characters or a number. The value, on the other hand, can be as complex as you need. This is what hashmaps have been made for: you can manipulate keys, move them from one part of your application to another, transmit them over a network, and when you need the full object, then you can retrieve it with its key.

Before you see all the details of the Map interface, here are the notions you need to have in mind.

- A hashmap can store key-value pairs
- A key acts as a symbol for a given value
- A key is a simple object, a value can be as complex as needed
- A key is unique in a hashmap, a value does not have to be unique
- Every value stored in a hashmap has to be bound to a key, a key-value pair in a map forms an entry of that map
- A key can be used to retrieve its bound value.

The Collections Framework gives you a Map interface that implements this concept, along with two extensions, **SortedMap** and **NavigableMap**, as shown on the following figure.



The Map Interface Hierarchy

This hierarchy is very simple and looks like the **Set** hierarchy, with **SortedSet** and **NavigableSet**. Indeed, a **SortedMap** shares the same kind of semantics as the **SortedSet**: a **SortedMap** is a map that keeps its key-value pairs sorted by their keys. The same goes for **NavigableMap**: the methods added by this interface are the same kind of methods than the ones added by **NavigableSet** to **SortedSet**.

The JDK gives you several implementations of the **Map** interface, the most widely used is the **HashMap** class.

Here are the two other implementations.

- **LinkedHashMap** is a **HashMap** with an internal structure to keep the key-value pairs ordered. Iterating on the keys or the key-value pairs will follow the order in which you have added your key-value pairs.
- **IdentityHashMap** is a specialized Map that you should only be used in very precise cases. This implementation is not meant to be generally used in application. Instead of using $equals()$ and $hashCode()$ to compare the key objects, this implementation only compares the references to these keys, with an equality operator (==). Use it with caution, only if you are sure this is what you need.

You may have heard of multimaps. Multimap is a concept where a single key can be associated to more than one value. This concept is not directly supported in the Collections Framework. This feature may be useful though, and you will see later in this tutorial how you can create maps with values that are in fact lists of values. This pattern allows you to create multimap-like structures.

2.9.2 Using the Convenience Factory Methods for Collections to Create Maps

As you already saw, Java SE 9 added methods to the List and Set interfaces to create immutable lists and sets.

There are such methods on the Map interface that create immutable maps and immutable entries.

You can create a Map easily with the following pattern.

```
Map<Integer, String> map = Map.of(
    1, "one",
    2, "two",
    3, "three"
);
```

There is one caveat though: you can only use this pattern it if you have no more than 10 key-value pairs.

If you have more, then you need to use another pattern:

```
Map.Entry<Integer, String> e1 = Map.entry(1, "one");
Map.Entry<Integer, String> e2 = Map.entry(2, "two");
Map.Entry<Integer, String> e3 = Map.entry(3, "three");

Map<Integer, String> map = Map.ofEntries(e1, e2, e3);
```

You can also write this pattern in this way, and use static imports to further improve its readability.

```
Map<Integer, String> map3 =
Map.ofEntries(
    entry(1, "one"),
    entry(2, "two"),
    entry(3, "three")
);
```

There are restrictions on these maps and entries created by these factory methods, as for the sets:

- The map and the entries you get are immutable objects
- Null entries, null keys, and null values are not allowed
- Trying to create a map with duplicate keys in this way does not make sense, so as a warning you will get an **IllegalArgumentException** at map creation.

2.9.3 Storing Key/Value Pairs in a Map

The relationship between a key and its bound value follows these two simple rules.

- A key can be bound to only one value
- A value can be bound to several keys.

This leads to several consequences for the content of the map.

- The set of all the keys cannot have any duplicates, so it has the structure of a **Set**
- The set of all the key/value pairs cannot have duplicates either, so it also has the structure of a **Set**
- The set of all the values may have duplicates, so it has the structure of a plain Collection.

Then, you can define the following operations on a map:

- Putting a key/value pair in the map. This may fail if the key is already defined in the map
- Getting a value from a key
- Removing a key from a map, along with its value.

You can also define the classic, set-like operations:

- Checking if the map is empty or not
- Getting the number of key-value pairs contained in the map
- Putting all the content of another map in this map
- Clearing the content of a map.

All these operations and concepts are implemented in the Map interface, along with some others that you are going to see in the following.

2.9.4 Exploring the Map interface

The Map interface is the base type that models the notion of map in the JDK.

You should be extremely careful when choosing the type of the keys for your maps. In a nutshell, choosing a mutable key is not prohibited but is dangerous and discouraged. Once a key has been added to a map, mutating it may lead to changing its hash code value, and its identity. This may make your key-value pair unrecoverable or may get you a different value when querying your map. You will see this later on an example.

The **Map** defines a member interface: **Map**.$Entry$ to model a key-value pair. This interface defines three methods to access the key and the values:

- $getKey()$: to read the key;
- $getValue()$ and $setValue(value)$: to read and update the value bound to that key.

The **Map**.$Entry$ objects you can get from a given map are views on the content of the map. Modifying the value of an entry object is thus reflected in the map and the other way round. This is the reason why you cannot change the key in this object: it could corrupt your map.

2.10 Managing the Content of a Map

2.10.1 Adding a Key Value Pair to a Map

You can simply add a key/value pair in a map with $put(key, value)$. If the key is not already present in the map, then the key/value pair is simply added to the map. If it is, then the existing value is replaced with the new one.

In both cases, the $put()$ method returns the existing value currently bound to the key. This means that if this a new key, a call to $put()$ will return $null$.

Java SE 8 introduces the $putIfAbsent()$ method. This method can also add a key/value pair to the map, only if the key is not already present and not associated to a null value. This may seem a bit confusing at first, but $putIfAbsent()$ will replace a null value with the new value provided.

This method is very handy if you need to get rid of faulty null values in your map. For instance the following code will fail with a **NullPointerException** because you cannot auto-unbox a null **Integer** to an int value.

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", null);
map.put("three", 3);
map.put("four", null);
map.put("five", 5);

for (int value : map.values()) {
  System.out.println("value = " + value);
}
```

If you take a close look at this code, you will see that $map.values()$ is a **Collection<Integer>**. So iterating on this collection produces instances of Integer. Because you declared value as an int, the compiler will auto-unbox this Integer to an int value. This mechanism fails with a **NullPointerException** if the instance of **Integer** is $null$.

You may fix this map with the following code, which replaces the faulty null values with a default value, -1, that will not generate any **NullPointerException** anymore.

```
for (String key : map.keySet()) {
  map.putIfAbsent(key, -1);
```

```
}
```

Running the previous code will print the following. As you can see this map does not contain any null values anymore:

Command window

```
value = -1
value = 1
value = -1
value = 3
value = 5
```

2.10.2 Getting a Value From a Key

You can get a value bound to a given key simply by calling the $get(key)$ method.

Java SE 8 introduced the $getOrDefault()$ method that takes a key and a default value which is returned if the key is not in the map.

Let us see this method in action in an example:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");

List<String> values = new ArrayList<>();
for (int i = 0; i < 5; i++) {
  values.add(map.getOrDefault(key,"UNDEFINED"));
}

System.out.println("values = " + values);
```

Or, if you are familiar with streams (which are covered later in this tutorial):

```
List<String> values = IntStream.range(0, 5)
  .mapToObj(key -> map.getOrDefault(key, "UNDEFINED"))
  .collect(Collectors.toList());
```

```
System.out.println("values = " + values);
```

Both codes print out the same result:

Command window

```
values = [UNDEFINED, one, two, three, UNDEFINED]
```

2.10.3 Removing a Key from a Map

Removing a key/value pair is made by calling the $remove(key)$ method. This method returns the value that was bound to that key, so it may return null.

It may be risky to blindly remove a key/value pair from a map if you do not know the value that is bound to that key. Thus, Java SE 8 added an overload that takes a value as a second argument. This time, the key/value pair is removed only if it fully matches the key/value pair in the map.

This $remove(key, value)$ method returns a boolean value, true if the key/value pair was removed from the map.

2.10.4 Checking for the Presence of a Key or a Value

You have two methods to check for the presence of a given key or a given value: $containsKey(key)$ and $containsValue(value)$. Both methods return true if the map contains the given key or value.

2.10.5 Checking for the Content of a Map

The **Map** interface also brings methods that look like the ones you have in the **Collection** interface. These methods are self-explanatory: $isEmpty()$ returns true for empty maps, $size()$ returns the number of key/value pairs, and $clear()$ removes all the content of the map.

There is also a method to add the content of a given map to the current map: $putAll(otherMap)$. If some keys are present in both maps, then the values of $otherMap$ will erase those of this map.

2.10.6 Getting a View on the Keys, the Values or the Entries of a Map

You can also get different sets defined on a map.

- *keySet*(): returns an instance of **Set**, containing the keys defined in the map
- *entrySet*(): returns an instance of **Set<Map.Entry>**, containing the key/value pairs contained in the map
- *values*(): returns an instance of **Collection**, containing the values present in the map.

The following examples show these three methods in action:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.put(4, "four");
map.put(5, "five");
map.put(6, "six");

Set<Integer> keys = map.keySet();
System.out.println("keys = " + keys);

Collection<String> values = map.values();
System.out.println("values = " + values);

Set<Map.Entry<Integer, String>> entries = map.entrySet();
System.out.println("entries = " + entries);
```

Running this code produces the following result:

Command window

```
keys = [1, 2, 3, 4, 5]
values = [one, two, three, four, five]
entries = [1=one, 2=two, 3=three, 4=four, 5=five]
```

These sets are views backed by the current map. Any change made to the map is reflected in those views.

Removing a Key From the Set of Keys

Modifying one of these sets will also be reflected in the map: for instance, removing a key from the set returned by a call to *keySet*() removes the corresponding key/value pair from the map.

For instance, you can run this code on the previous map:

```
keys.remove(3);
entries.forEach(System.out::println);
```

It will produce the following resut:

Command window

```
1=one
2=two
4=four
5=five
6=six
```

Removing a Value From the Collection of Values

Removing a value is not as simple because a value can be found more than once in a map. In that case, removing a value from the collection of values just removes the first matching key/value pair.

You can see that on the following example.

```
Map<Integer, String> map = Map.ofEntries(
    Map.entry(1, "one"),
    Map.entry(2, "two"),
    Map.entry(3, "three"),
    Map.entry(4, "three")
);

map = new HashMap<>(map);
map.values().remove("three");
System.out.println("map = " + map);
```

Running this code will produce the following result.

Command window

```
map before = {1=one, 2=two, 3=three, 4=three}
map after  = {1=one, 2=two, 4=three}
```

As you can see, only the first key/value pair has been removed in this example. You need to be careful in this case because if the implementation you chose is a **HashMap**, you cannot tell in advance what key/value pair will be found.

You do not have access to all the operations on these sets though. For instance, you cannot add an element to the set of keys, or to the collection of values. If you try to do that, you will get an **UnsupportedOperationException**.

If what you need is to iterate over the key/value pairs of a map, then your best choice is to iterate directly on the set of key/value pairs. It is much more efficient to do that, rather than iterating on the set of keys, and getting the corresponding value. The best pattern you can use is the following:

```
for (Map.Entry<Integer, String> entry : map.entrySet()) {
  System.out.println("entry = " + entry);
}
```

2.11 Handling Map Values with Lambda Expressions

2.11.1 Consuming the Content of a Map

The **Map** interface has a $forEach()$ method that works in the same way as the $forEach()$ method on the **Iterable** interface. The difference is that this $forEach()$ method takes a **BiConsumer** as an argument instead of a simple **Consumer**.

Let us create a simple map and print out its content.

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

This code produces the following result:

```
Command window

1 :: one
2 :: two
3 :: three
```

2.11.2 Replacing Values

The Map interface gives you three methods to replace a value bound to a key with another value.

The first one is $replace(key, value)$, which replaces the existing value with the new one, blindly. This is the equivalent of a put-if-present operation. This method returns the value that was removed from your map.

If you need finer control, then you can use an overload of this method, which takes the existing value as an argument: $replace(key, existingValue, newValue)$. In this case, the existing value is replaced only if it matches the new value. This method returns true if the replacement occurred.

The **Map** interface has also a method to replace all the values of your map using a **BiFunction**. This **BiFunction** is a remapping function, which takes the key and the value as arguments, and returns a new value, which will replace the existing value. A call to this method iterates internally on all the key/value pairs of your map.

The following example shows how you can use this $replaceAll()$ method:

```
Command window

1 :: ONE
2 :: TWO
3 :: THREE
```

2.11.3 Computing Values

The **Map** interface gives you a third pattern to add key-value pairs to a map or modify a map's existing values in the form of three methods: $compute()$, $computeIfPresent()$, and $computeIfAbsent()$.

These three methods take the following arguments:

- the key on which the computation is made
- the value bound to that key, in the case of $compute()$ and $computeIfPresent()$

- a **BiFunction** that acts as a remapping function, or a mapping function in the case of *computeIfAbsent*().

In the case of *compute*(), the remapping bi-function is called with two arguments. The first one is the key, and the second one is the existing value if there is one, or null if there is none. Your remapping bifunction can be called with a null value.

For *computeIfPresent*(), the remapping function is called if there is a value bound to that key and if it is not null. If the key is bound to a null value, then the remapping function is not called. Your remapping function cannot be called with a null value.

For *computeIfAbsent*(), beacause there is no value bound to that key, the remapping function is in fact a simple Function that takes the key as an argument. This function is called if the key is not present in the map or if it is bound to a null value.

In all cases, if your bifunction (or function) returns a null value, then the key is removed from the map: no mapping is created for that key. No key/value pair with a null value can be put in the map using one of these three methods.

In all cases, the value returned is the new value bound to that key in the map or null if the remapping function returned null. It is worth pointing out that this semantic is different from the *put*() methods. The *put*() methods return the previous value, whereas the *compute*() methods return the new value.

A very interesting use case for the *computeIfAbsent*() method is the creation of maps with lists as values. Suppose you have the following list of strings: [one two three four five six seven]. You need to create a map, where the keys are the length of the words of that list, and the values are the lists of these words. What you need to create is the following map:

Command window

```
3 :: [one, two, six]
4 :: [four, five]
5 :: [three, seven]
```

Without the *compute*() methods, you would probably write this:

```
List<String> strings = List.of("one", "two", "three", "four", "five", "six", "
    ↪ seven");
Map<Integer, List<String>> map = new HashMap<>();
for (String word: strings) {
  int length = word.length();
  if (!map.containsKey(length)) {
    map.put(length, new ArrayList<>());
```

```
    }
    map.get(length).add(word);
}

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

Running this code produces the expected result:

Command window

```
3 :: [one, two, six]
4 :: [four, five]
5 :: [three, seven]
```

By the way, you could use a *putIfAbsent()* to simplify this for loop:

```
for (String word: strings) {
    int length = word.length();
    map.putIfAbsent(length, new ArrayList<>());
    map.get(length).add(word);
}
```

But using *computeIfAbsent()* can make this code even better:

```
for (String word: strings) {
    int length = word.length();
    map.computeIfAbsent(length, key -> new ArrayList<>())
       .add(word);
}
```

How does this code work?

- If the key is not in the map, then the mapping function is called, which creates an empty list. This list is returned by the *computeIfAbsent()* method. This is the empty list in which the code adds word.

- If the key is in the map, the mapping function is not called, and the current value bound to that key is returned. This is the partially filled list in which you need to add word.

This code is much more efficient than the $putIfAbsent()$ one, mostly because in that case, the empty list is created only if needed. The $putIfAbsent()$ call requires an existing empty list, which is used only if the key is not in the map. In cases where the object you add to the map has to be created on demand, then using $computeIfAbsent()$ should be preferred over $putIfAbsent()$.

2.11.4 Merging Values

The $computeIfAbsent()$ pattern works well if your map has values that are aggregations of other values. But there is a restriction on the structure that is supporting this aggregation: it has to be mutable. This is the case for **ArrayList**, and this is what the code you wrote does: it adds your values to an **ArrayList**.

Instead of creating lists of words, suppose you need to create a concatenation of words. The String class is seen here as an aggregation of other strings, but it is not a mutable container: you cannot use the $computeIfAbsent()$ pattern to do that.

This is where the $merge()$ pattern comes to the rescue. The $merge()$ method takes three arguments:

- a key
- a value, that you need to bind to that key
- a remapping BiFunction.

If the key is not in the map or bound to a null value, then the value is bound to that key. The remapping function is not called in this case.

On the contrary, if the key is already bound to a non-null value, then the remapping function is called with the existing value, and the new value passed as an argument. If this remapping function returns null, then the key is removed from the map. The value it produces is bound to that key otherwise.

You can see this $merge()$ pattern in action on the following example:

```
1 List<String> strings = List.of("one", "two", "three", "four", "five", "six", "
      ↪ seven");
  Map<Integer, String> map = new HashMap<>();
3 for (String word: strings) {
    int length = word.length();
5   map.merge(length, word,
```

```
    (existingValue, newWord) -> existingValue + ", " + newWord);
}

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

In this case, if the length key is not in the map, then the $merge()$ call just adds it and binds it to word. On the other hand, if the length key is already in the map, then the bifunction is called with the existing value and word. The result of the bifunction then replaces the current value.

Running this code produces the following result:

Command window

```
3 :: one, two, six
4 :: four, five
5 :: three, seven
```

In both patterns, $computeIfAbsent()$ and $merge()$, you may be wondering why the lambda created takes an argument that is always available in the context of this lambda, and that could be captured from that context. The answer is: you should favor non-capturing lambdas over capturing ones, for performance reasons.

2.12 Keeping Keys Sorted with SortedMap and NavigableMap

2.12.1 Methods Added by SortedMap

The JDK provides two extensions of the Map interface: SortedMap and **NavigableMap**. **NavigableMap** is an extension of **SortedMap**. Both interfaces are implemented by the same class: **TreeMap**. The **TreeMap** class is a red-black tree, a well-known data structure.

SortedMap and **NavigableMap** keep their key/value pairs sorted by key. Just as for **SortedSet** and **NavigableSet**, you need to provide a way to compare these keys. You have two solutions to do this: either the class of your keys implements **Comparable**, or you provide a Comparator for your keys when creating your **TreeMap**. If you provide a **Comparator**, it will be used even if your keys are comparable.

If the implementation you chose for your **SortedMap** or **NavigableMap** is **TreeMap**, then you can safely cast the set returned by a call to $keySet()$ or $entrySet()$ to **SortedSet** or **NavigableSet**. **NavigableMap** has a method, $navigableKeySet()$ that returns an instance of **NavigableSet** that you can use instead of the plain $keySet()$ method. Both methods return the same object.

The **SortedMap** interface adds the following methods to **Map**:

- $firstKey()$ and $lastKey()$: returns the lowest and the greatest key of your map;
- $headMap(toKey)$ and $tailMap(fromKey)$: returns a **SortedMap** whose keys are strictly less than $toKey$, or greater than or equal to $fromKey$;
- $subMap(fromKey, toKey)$: returns a **SortedMap** whose keys are strictly lesser than $toKey$, or greater than or equal to $fromKey$.

These maps are instances of **SortedMap** and are views backed by this map. Any change made to this map will be seen in these views. These views can be updated, with a restriction: you cannot insert a key outside the boundaries of the map you built.

You can see this behavior on the following example:

```
SortedMap<Integer, String> map = new TreeMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.put(5, "five");
map.put(6, "six");

SortedMap<Integer, String> headMap = map.headMap(3);
headMap.put(0, "zero"); // this line is ok
headMap.put(4, "four"); // this line throws an IllegalArgumentException
```

2.12.2 Methods Added by NavigableMap

Accessing to Specific Keys or Entries

The **NavigableMap** adds more methods to **SortedMap**. The first set of methods gives you access to specific keys and entries in your map.

- $firstKey()$, $firstEntry()$, $lastEntry()$, and $lastKey()$: return the lowest or greatest key or entry from this map.
- $ceilingKey(key)$, $ceilingEntry(key)$, $higherKey(key)$, $higherEntry(key)$: return the lowest key or entry greater than the provided key. The ceiling methods

may return a key that is equal to the provided key, whereas the key returned by the higher methods is strictly greater.

- $floorKey(key), floorEntry(key), lowerKey(key), lowerEntry(key)$: return the greatest key or entry lesser than the provided key. The floor methods may return a key that is equal to the provided key, whereas the key returned by the higher methods is strictly lower.

Accessing your Map with Queue-Like Features

The second set gives you queue-like features:

- $pollFirstEntry()$: returns and removes the lowest entry
- $pollLastEntry()$: returns and removes the greatest entry.

Traversing your Map in the Reverse Order

The third set reverses your map, as if it had been built on the reversed comparison logic.

- $navigableKeySet()$ is a convenience method that returns a **NavigableSet** so that you do not have to cast the result of $keySet()$
- $descendingKeySet()$: returns a **NavigableSet** backed by the map, on which you can iterate in the descending order
- $descendingMap()$: returns a **NavigableMap** with the same semantic.

Both views support element removal, but you cannot add anything through them.

Here is an example to demonstrate how you can use them.

```
NavigableMap<Integer, String> map = new TreeMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.put(4, "four");
map.put(5, "five");

map.keySet().forEach(key -> System.out.print(key + " "));
System.out.println();

NavigableSet<Integer> descendingKeys = map.descendingKeySet();
descendingKeys.forEach(key -> System.out.print(key + " "));
```

Running this code prints out the following result:

```
Command window
1 2 3 4 5
5 4 3 2 1
```

Getting Submap Views

The last set of methods give you access to views on portions of your map.

- $subMap(fromKey, fromInclusive, toKey, toInclusive)$: returns a submap where you can decide to include or not the boundaries
- $headMap(toKey, inclusive)$: same for the head map
- $tailMap(fromKey, inclusive)$: same for the tail map.

These maps are views on this map, which you can update by removing or adding key/value pairs. There is one restriction on adding elements though: you cannot add keys outside the boundaries on which the view has been created.

2.13 Choosing Immutable Types for Your Key

2.13.1 Avoiding the Use of Mutable Keys

Using mutable key is an antipattern, and you should definitely avoid doing that. The side effects you may get if you do are terrible: you may end up making the content of your map unreachable.

It is quite easy to set up an example to show that. Here is a **Key** class, which is just a mutable wrapper on an $String$. Note that the $equals()$ and $hashCode()$ methods have been overridden by a code that your IDE could generate.

```
/*
 * This an example of an antipattern .
 * Do not do this in your production code !!!
 */
class Key {
  private String key;
```

```
    public Key(String key) {
        this.key = key;
    }

    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

    @Override
    public String toString() {
        return key;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Key key = (Key) o;
        return Objects.equals(key, key.key);
    }

    @Override
    public int hashCode() {
        return key.hashCode();
    }
}
```

You can use this wrapper to create a map in which to put key-value pairs in.

```
Key one = new Key("1");
Key two = new Key("2");

Map<Key, String> map = new HashMap<>();
map.put(one, "one");
map.put(two, "two");

System.out.println("map.get(one) = " + map.get(one));
System.out.println("map.get(two) = " + map.get(two));
```

So far this code is OK and prints out the following:

Command window

```
map.get(one) = one
map.get(two) = two
```

What will happen is someone mutates your key? Well, it really depends on the mutation. You can try the ones in the following example, and see what is happening when you try to get your values back.

```
one.setKey(5);
two.setKey(1);

System.out.println("map.get(one) = " + map.get(one));
System.out.println("map.get(two) = " + map.get(two));
```

The result is not what you want.

Command window

```
map.get(one) = null
map.get(two) = one
```

As you can see, even on a very simple example, things can go terribly wrong: the first key cannot be used to access the right value anymore, and the second one gives access to the wrong value.

In a nutshell: if you really cannot avoid using mutable keys, do not mutate them.

2.13.2 Diving in the Structure of HashSet

You may be wondering why would it be interesting to talk about the **HashSet** class in this section? Well, it turns out that the **HashSet** class is in fact built on an internal **HashMap**. So the two classes share some common features.

Here is the code of the add(element) of the **HashSet** class:

```
private transient HashMap<E,Object> map;
private static final Object PRESENT = new Object();
```

```
4 public boolean add(E e) {
      return map.put(e, PRESENT) == null;
6 }
```

What you can see is that in fact, a hashset stores your object in a hashmap (the transient keyword is not relevant). Your objects are the keys of this hashmap, and the value is just a placeholder, an object with no significance.

The important point to remember here is that if you mutate your objects after you have added them to a set, you may come across weird bugs in your application, that will be hard to fix.

Let us take the previous example again, with the mutable **Key** class. This time, you are going to add instances of this class to a set.

```
  Key one = new Key("1");
2 Key two = new Key("2");

4 Set<Key> set = new HashSet<>();
  set.add(one);
6 set.add(two);

8 System.out.println("set = " + set);

10 // You should never mutate an object once it has been added to a Set!
  one.setKey("3");
12 System.out.println("set.contains(one) = " + set.contains(one));
  boolean addedOne = set.add(one);
14 System.out.println("addedOne = " + addedOne);
  System.out.println("set = " + set);
```

Running this code produces the following result:

Command window

```
1 set = [1, 2]
  set.contains(one) = false
3 addedOne = true
  set = [3, 2, 3]
```

You can see that in fact the first element and the last element of the set are the same:

```
List<Key> list = new ArrayList<>(set);
Key key0 = list.get(0);
Key key2 = list.get(2);

System.out.println("key0 = " + key0);
System.out.println("key2 = " + key2);
System.out.println("key0 == key2 ? " + (key0 == key2));
```

If you run this last piece of code, you will get the following result:

Command window

```
key0 = 3
key2 = 3
key0 == key2 ? true
```

In this example, you saw that mutating an object once it has been added to a set can lead to having the same object more than once in this set. Simply said, do not do that!

Part IV Correctness, Robustness, Efficiency

Part V Design Patterns

Part VI Java Generics

Part VII Java Concurrency

Part VIII Java GUI

Part IX References