3.7. Collections and Generic Types

Software and Analysis Design

2nd Year, Computer Science

Universidad Autónoma de Madrid

Contents

- **Introduction**
- Interfaces
- Comparing objects
- Implementations
- Algorithms

- Generic types

Introduction

- A collection (or aggregation) is an object that groups multiple elements as a whole
- Collections are used to store, retrieve, manipulate and communicate aggregated data
- They allow representing data elements that form groups in a natural manner
  - A poker hand (a collection of cards)
  - An email folder (a collection of emails)
  - A phone book (a dictionary that maps names to phone numbers)

The Java Collection Framework

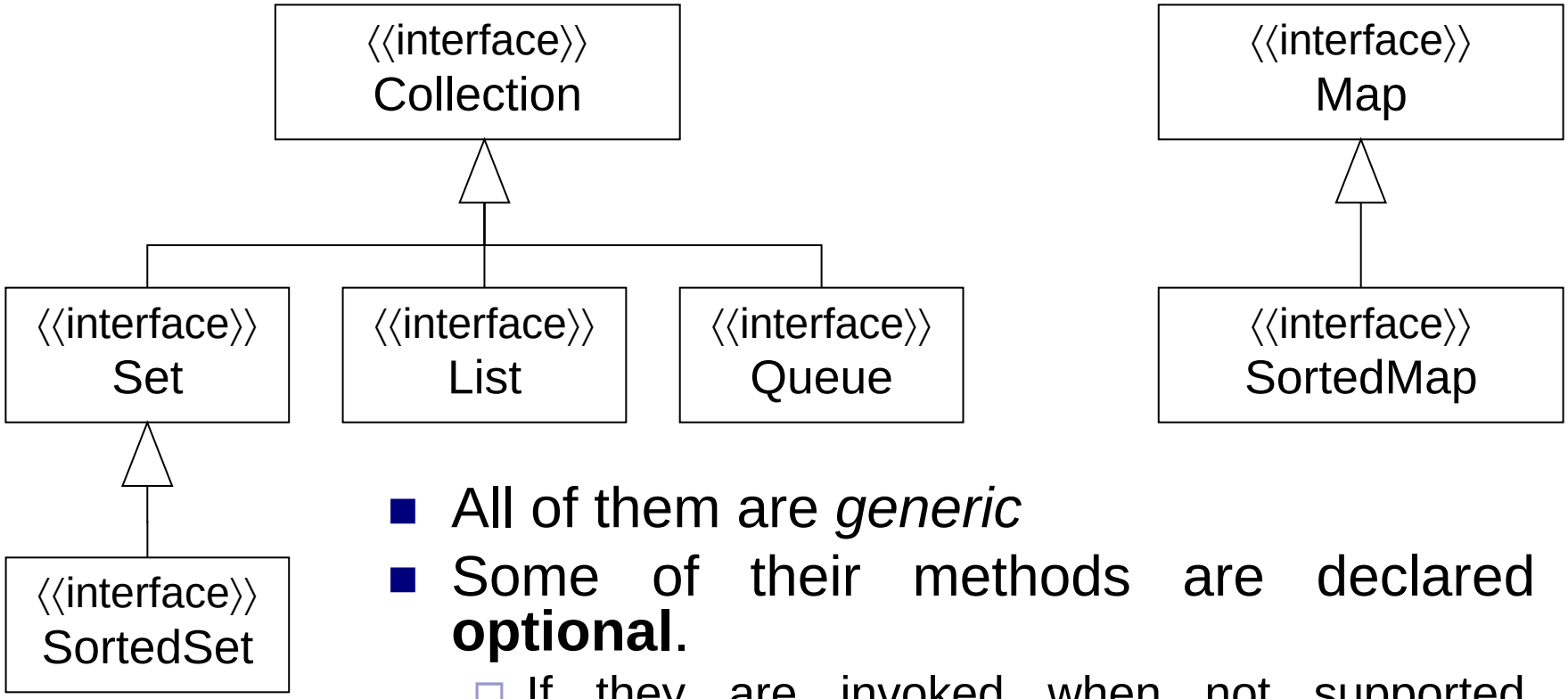
- A unified framework to represent and manipulate collections.
- Made of
  - **Interfaces**. Allow manipulating collections independently of their implementation
  - **Implementations**. Concrete implementations of the interfaces
  - **Algorithms**. Methods providing useful computations, such as searching and sorting, applied to any objects that implement any of the collection's interfaces
- Other similar frameworks: the STL of C++

Advantages of using collections

- Following common patterns reduces learning time and **increases** coding **productivity** by using software components
- **Efficiency**: the framework implementations are optimized for the typical uses of collections
- **Interoperability** among software libraries.
  - Different libraries use collections through the same interfaces, facilitating their integration

Interfaces of the collections

The most important interfaces

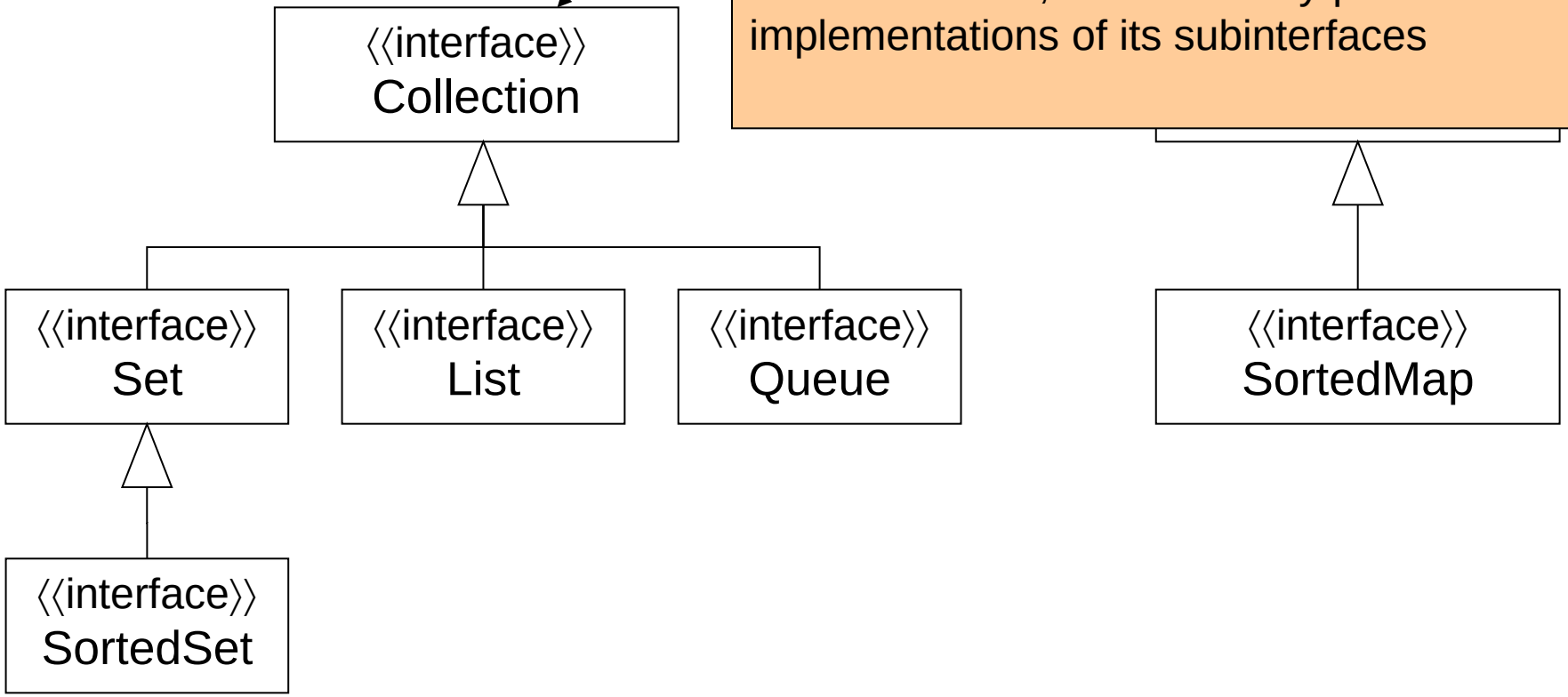


- All of them are *generic*
- Some of their methods are declared as **optional**.
  - If they are invoked when not supported, the collection throws an exception of the class UnsupportedOperationException

Interfaces of the C

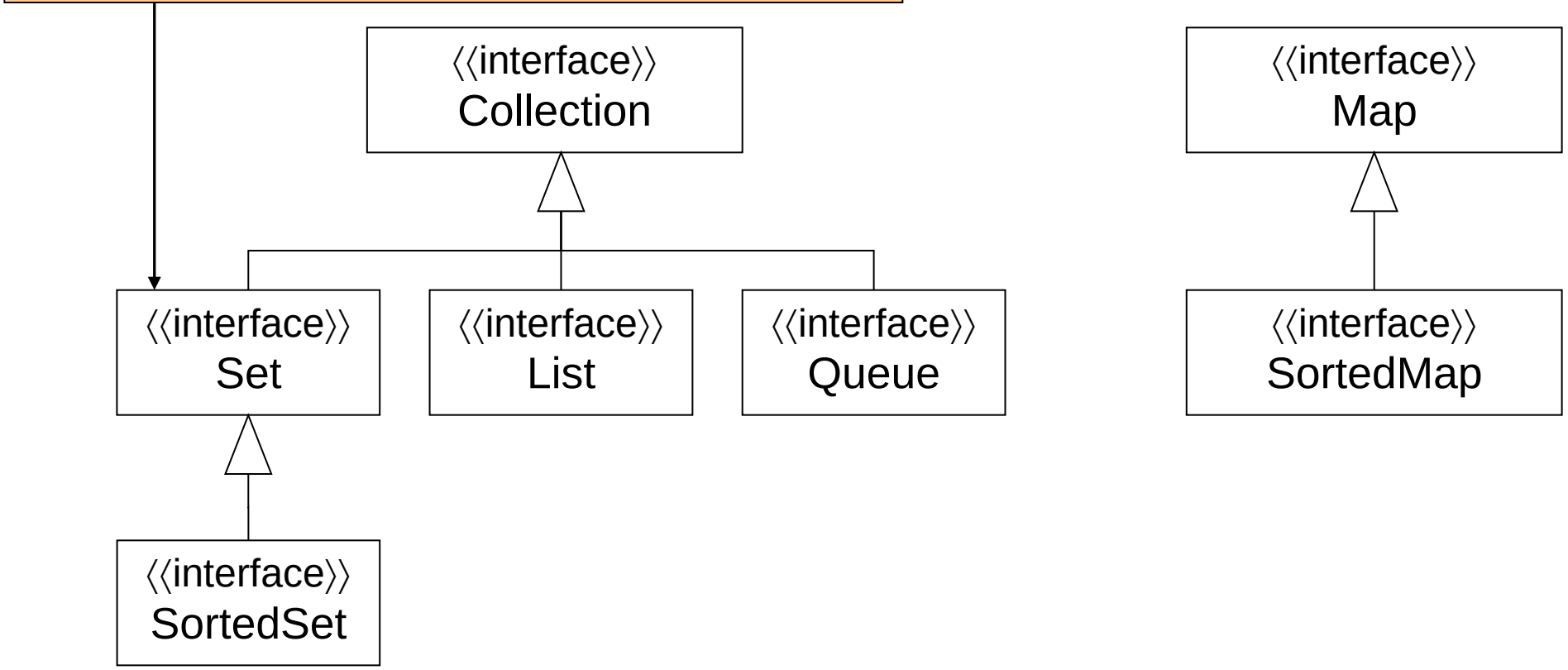
The most important interfaces

- The root of the collection hierarchy
- Represents the smallest group of features that all collections must implement
- Useful for collection sharing when the goal is to maximize generality
- Java does not provide a direct implementation of this interface, but the library provides implementations of its subinterfaces



Interfaces of the collections

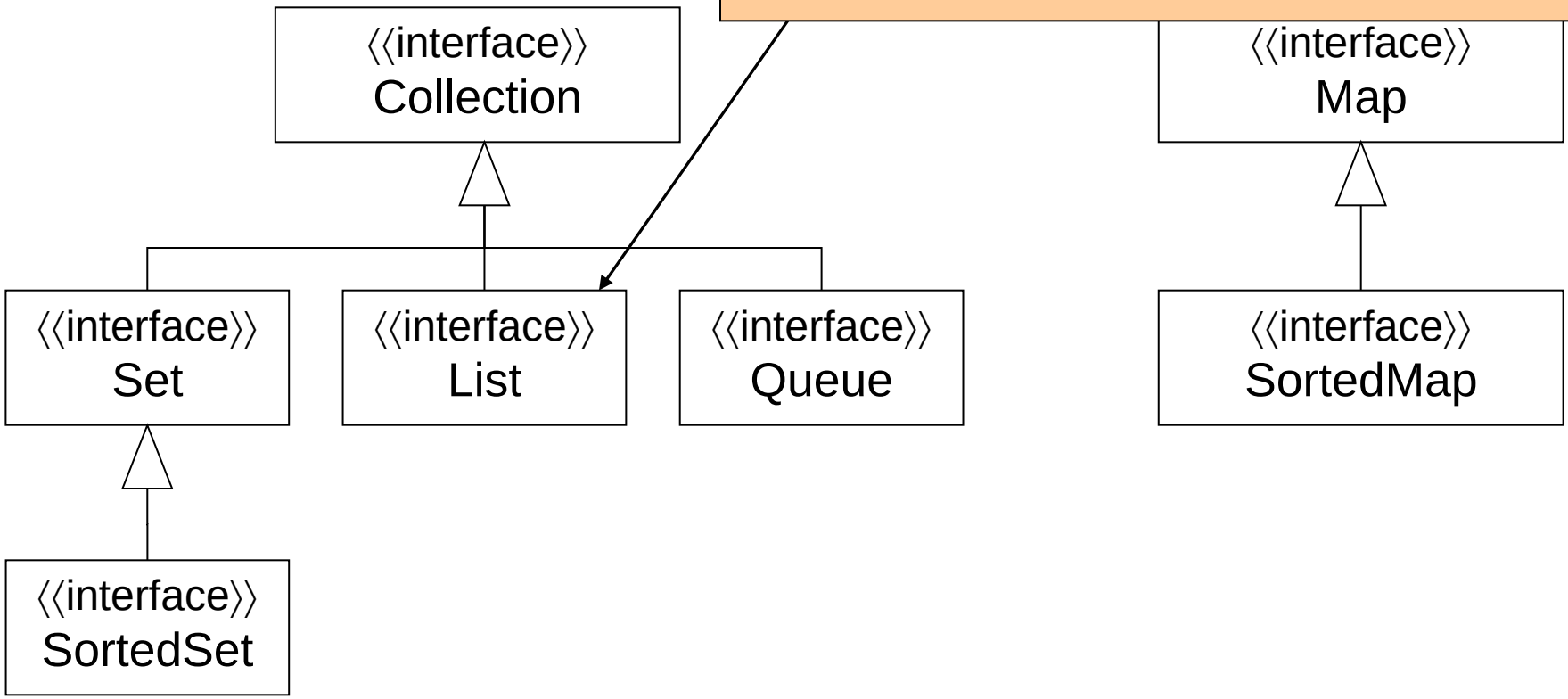
- A collection that excludes duplicated elements.
- Models the mathematical abstraction of “set”.



Interfaces of the collections

The most important interfaces

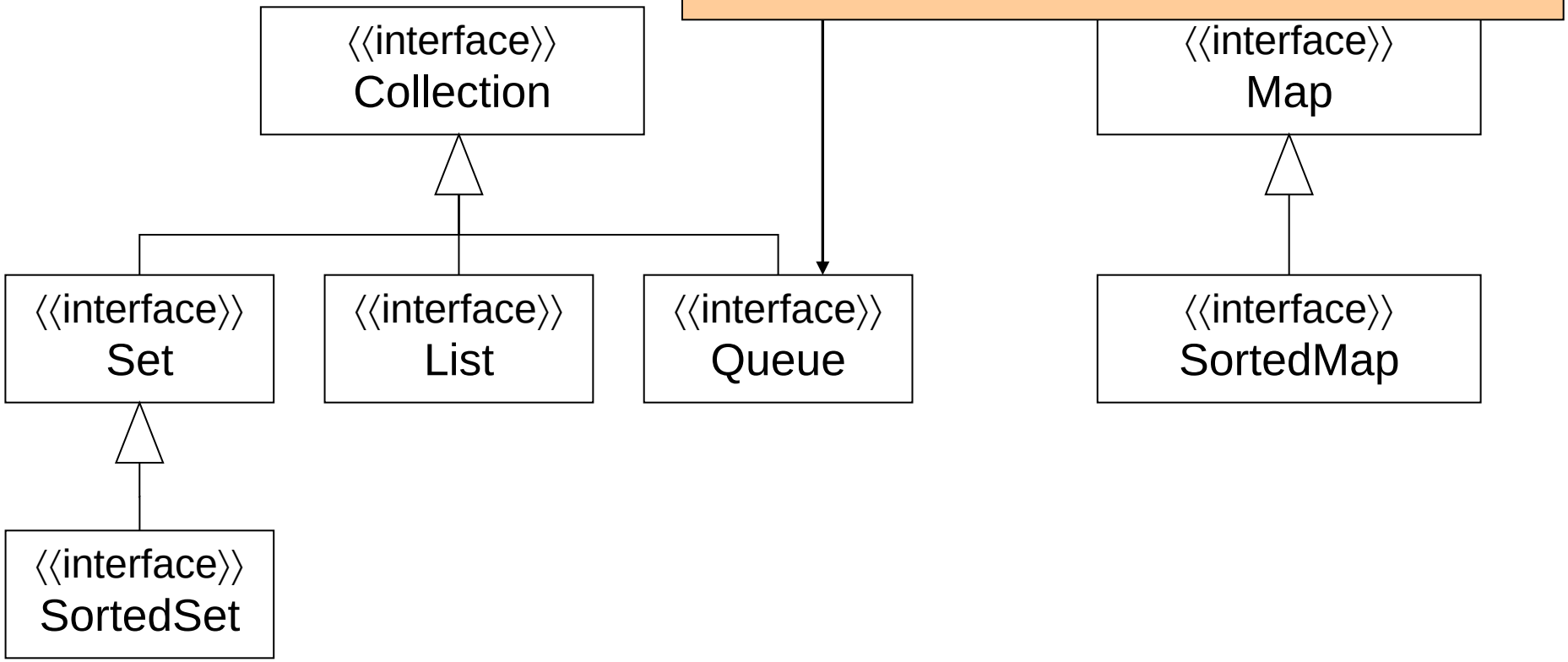
- An ordered collection, also known as “sequence”.
- May contain duplicated elements.



Interfaces of the collections

The most important interfaces

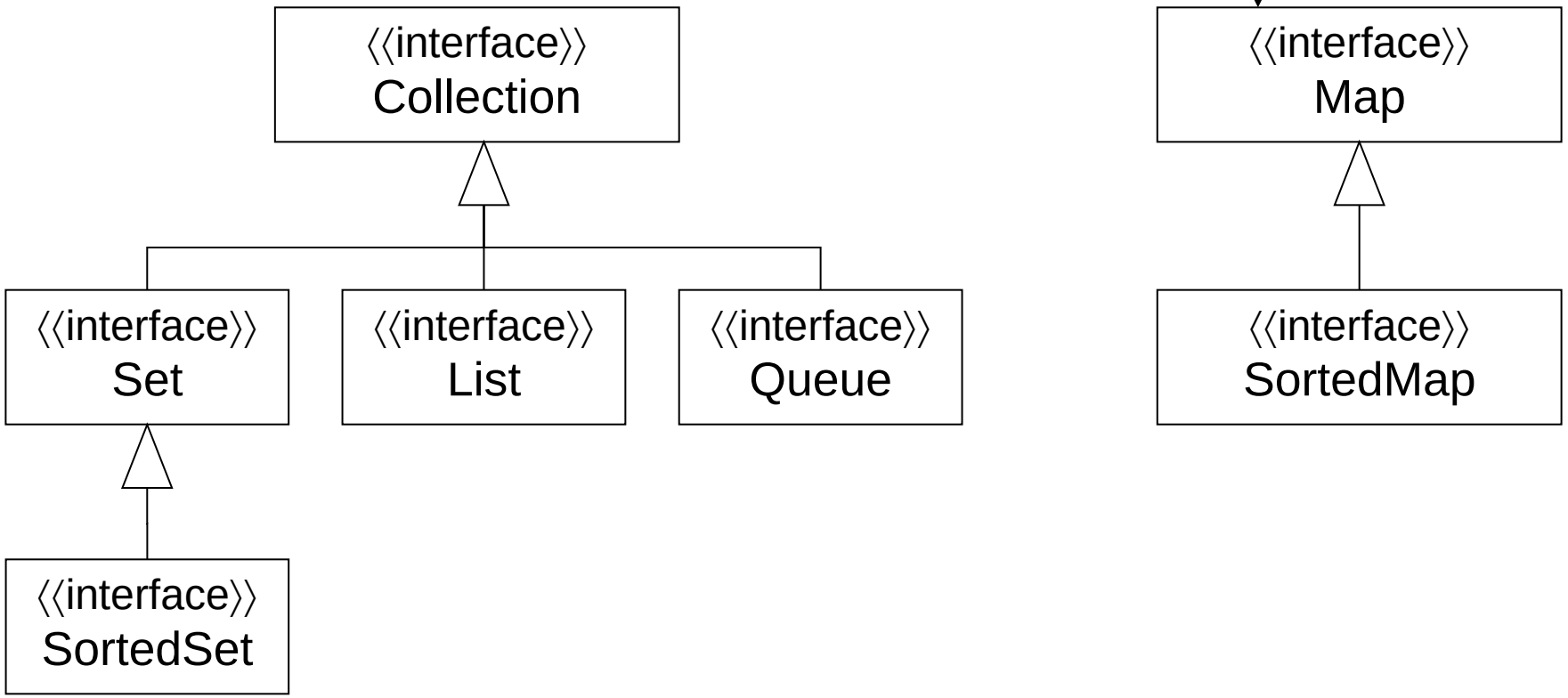
- A queue typically offers its elements ordered by a FIFO criterion.
- There are also priority queues.



Interfaces of the

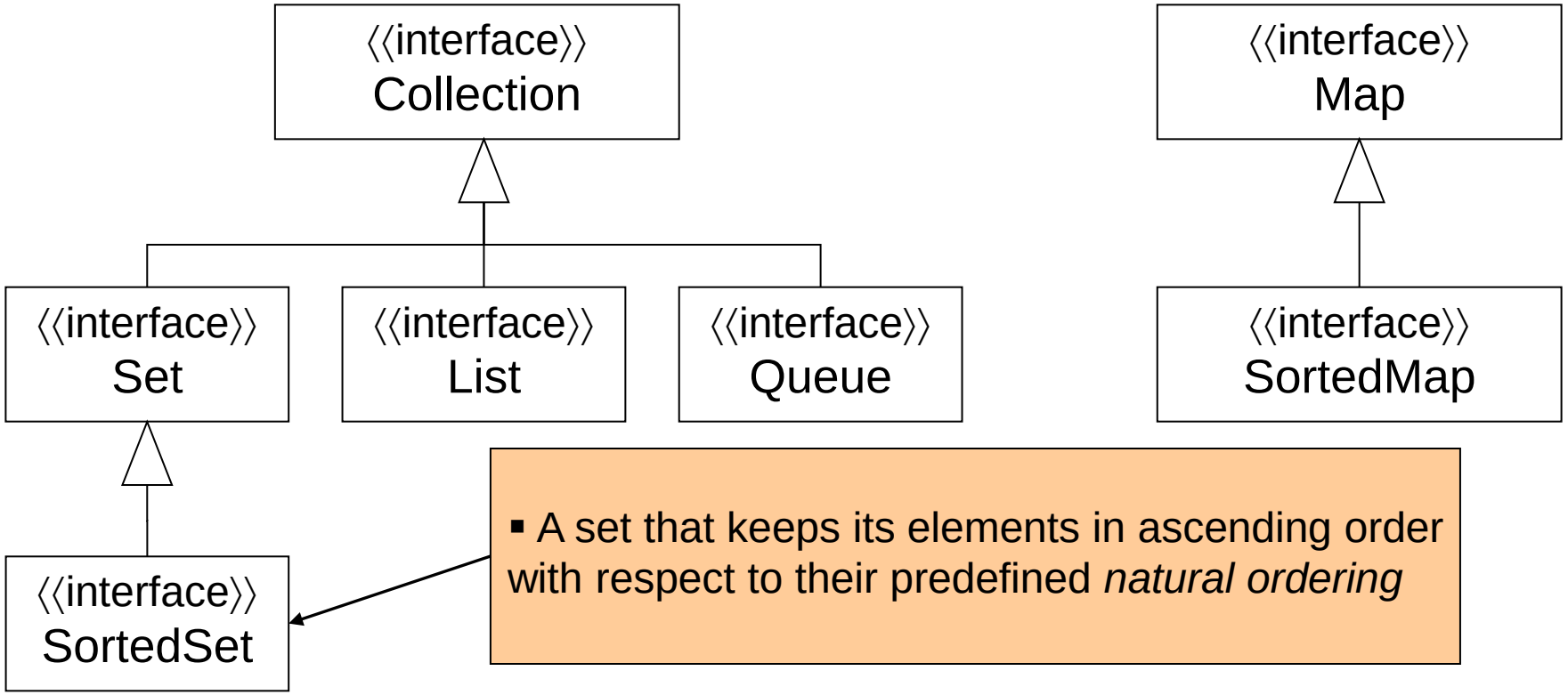
The most important interfaces

- A dictionary of pairs (key, value).
- They cannot contain duplicated keys.
- Realizes the math concept of function (*mapping)*



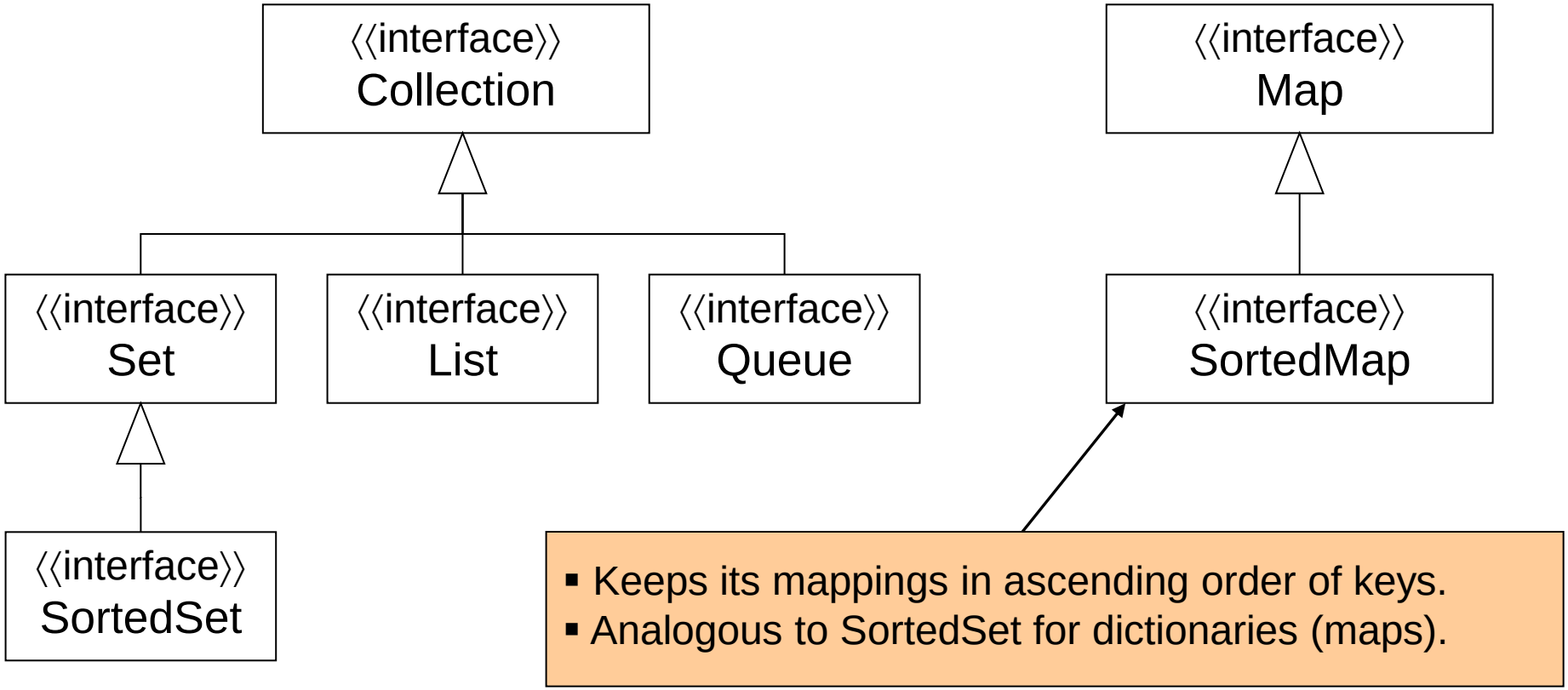
Interfaces of the collections

The most important interfaces



Interfaces of the collections

The most important interfaces



The `Collection` interface

- Useful for sharing collections, with maximum generality
- All provided implementations have a constructor that accepts a `Collection`, in order to initialize their content
- Thus, it allows easy conversion between different types of collections:

```
Collection<String> c = new HashSet<String>();
c.add("One");
c.add("Two");
c.add("Three");
// initializing a list out of the set
List<String> list = new ArrayList<String>(c);
```

The Collection interface

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);                          //optional
    boolean remove(Object element);                  //optional
    Iterator<E> iterator();
    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);              //optional
    boolean retainAll(Collection<?> c);              //optional
    void clear();                                    //optional
    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

How to traverse collections

- With an **enhanced for**:

```
for (Object o : collection)
    System.out.println(o);
```

- Caution: it does not allow removing/adding elements from inside the for block (will throw a `ConcurrentModificationException`)
- With an **iterator**. An iterator is an object that allows traversing collections (even several in parallel) and element removal

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next())) it.remove();
}
```

The `Set` interface

- A collection that does not admit duplicated elements.
- Contains only inherited methods from `Collection` but adding the restriction for no duplicates.
- Three implementations:
  - `HashSet:` keeps elements in a hash.
  - `TreeSet:` keeps elements ordered in a red-black tree. Elements must implement `Comparable`.
  - `LinkedHashSet:` keeps elements in a linked hash.

The Set interface

Examples

- Eliminate duplicated elements in an existing collection *c*:

```
Collection<Type> noDups = new HashSet<Type>(c);
```

- Print repeated words:

```
import java.util.*;
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicated: " + a);
        System.out.println(s.size() + " distinct words:”
                                    + s);
    }
}
```

The `List` interface

- An ordered collection (a sequence) and may contain duplicate elements
- In addition to methods inherited from *Collection*, it has methods for
  - **Access by position index —** to get elements given their index position in the list
  - **Searching —** to find a specific element in the list and return its index position
  - **Iteration —** to extend the semantics of `Iterator` taking advantage of the sequential nature of a list
  - **Range —** to allow operating on sublists views of a list
- Three implementations: `ArrayList`, `LinkedList`, and `Vector`

The `List` interface

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);                          //optional
    boolean add(E element);                               //optional
    void add(int index, E element);                       //optional
    E remove(int index);                                  //optional
    boolean addAll(int index, Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

List iterators

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

- Iterating backwards, from the end to the first element:

```
for (ListIterator<Type> it = list.listIterator(list.size());
     it.hasPrevious(); )
{
    Type t = it.previous();
    ...
}
```

LinkedList vs. ArrayList

LinkedList<E>

- get(int index) is O(n)
- add(E element) is O(1)
- add(int index, E element) is O(n)
- remove(int index) is O(n)
- Iterator.remove() is O(1) **← main benefit of LinkedList**
- ListIterator.add(E element) is O(1) **← main benefit of LinkedList**

ArrayList<E>

- get(int index) is O(1) **←main benefit ArrayList**
- add(E element) is O(1) amortized, but O(n) in the worst case, because the array must be resized and copied.
- add(int index, E element) is O(n - index) amortized, but O(n) in the worst case
- remove(int index) is O(n - index) (i.e. remove the last one is O(1))
- Iterator.remove() is O(n - index)
- ListIterator.add(E element) is O(n - index)

Exercise

- Given two lists of integers, return a collection with the common elements, without repetitions

[try it, and then check the step-by-step explanation video: **https://www.youtube.com/watch?v=BWtK8_E3uQk**]

The Queue interface

```
public interface Queue<E> extends Collection<E> {
  E element();
  boolean offer(E e);
  E peek();
  E poll();
  E remove();
}
```

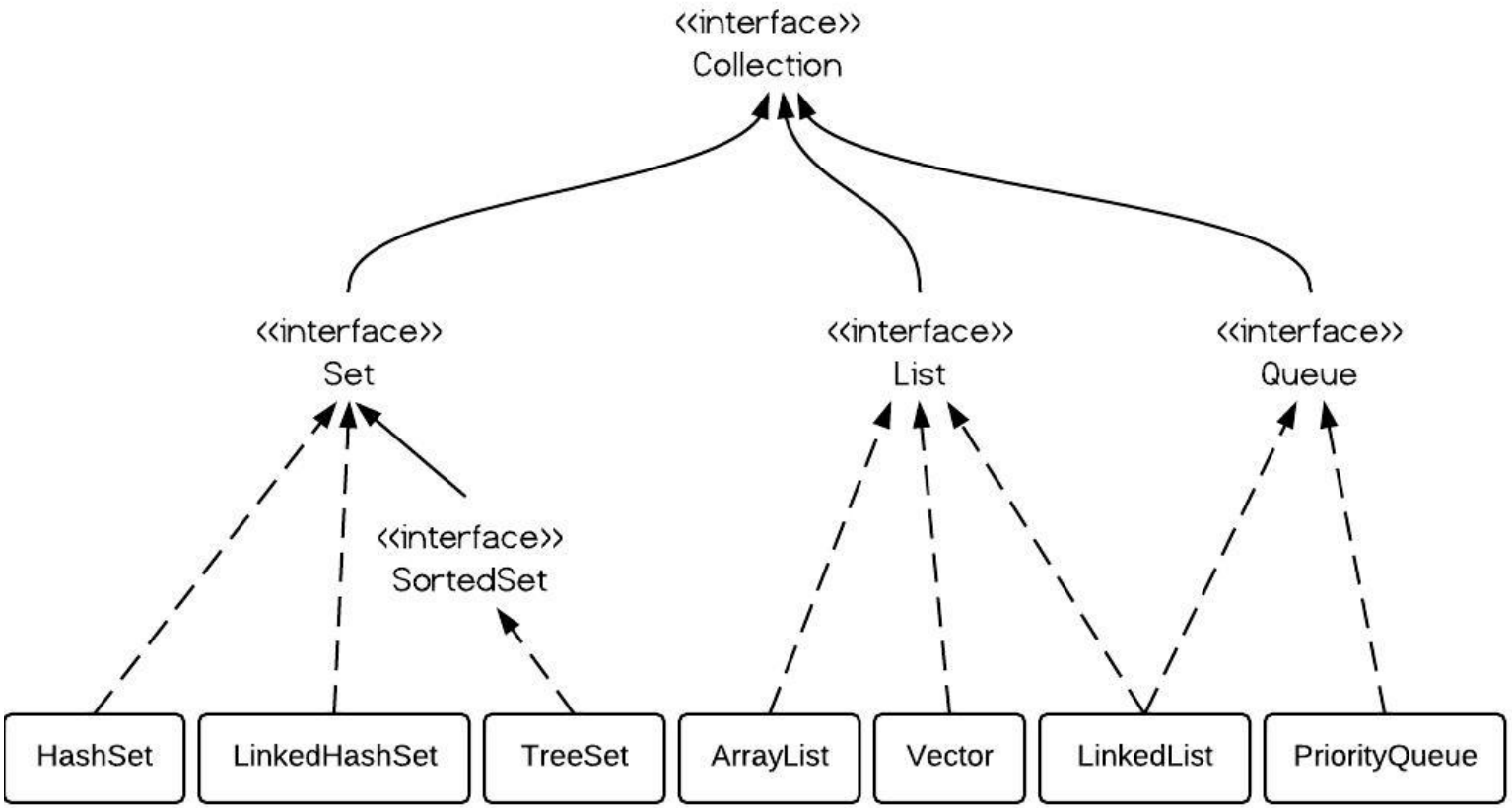
	Throws an exception	Returns a special value (null/false)
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

The Queue interface

```
import java.util.*;

public class Countdown {
    public static void main(String[] args) throws InterruptedException{
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();
        for (int i = time; i >= 0; i--)
            queue.add(i);
        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

Implementations of some collections



http://www.programcreek.com/

The `Map` interface

- An object that associates *values* to *keys*
- It cannot contain duplicated keys
- Each key is mapped to at most one value
- Models the mathematical concept of function (*mapping*)
- Three implementations: `HashMap`, `TreeMap`, and `LinkedHashMap`

The Map interface

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry<K, V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

The Map interface

Example

```
import java.util.*;

public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();

        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

The `Map` interface

Example

- Iterate over the whole set of keys:

```
for (KeyType key : m.keySet())
  System.out.println(key);
```

- Iterate over all pairs of tuples (key,value):

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());
```

Sorting

- A collection of type `List` can be sorted using:

```
Collections.sort(aList);
```

- Sorting applies the *natural order* defined by any class implementing the interface `Comparable` (method `compareTo`).

- Alternatively, we can use the interface `Comparator<T>` with method `compare(T, T)`.

The SortedSet interface

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

- Keeps its elements in ascending order, according to the *natural order* of it keys or according to the `Comparator` provided (optionally) in the constructor used to create the `SortedSet`.

The `SortedSet` interface

Examples

- Obtain the number of words between "*doorbell*" and "*pickle*", without including the latter.

```
int count = dictionary.subSet("doorbell", "pickle").size();
```

- Remove all words starting with a letter f.

```
dictionary.subSet("f", "g").clear();
```

The SortedMap interface

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

- Keeps its entries in ascending order, according to the *natural order* of it keys or according to the `Comparator` provided (optionally) in the constructor used to create the `SortedMap`.

Variations of collections

- There are no specific types for collection variants (e.g., fixed-size, read-only, or immutable)
- Java does not declare constant objects, as C++ with the **const** modifier
- Solution: interfaces may have **optional methods**
  - An optional method is just a normal method, with a default implementation that throws `UnsupportedOperationException`
  - If they are executed when not permitted they throw `UnsupportedOperationException`
  - We must document which methods are not supported

Exercise

- Create a class that maintains stock share prices of companies
- Observer objects can register to obtain the changes in the stock shares
- Aim for a general design, which allows having different kinds of observer objects:
  - An observer that prints in the console the changes in the stock shares
  - Another that prints in a file
  - etc.
- Modify the program so that observers can register to changes in the shares of specific companies, and not to all changes

Note: we will be using the Observer design pattern:
https://en.wikipedia.org/wiki/Observer_pattern

Contents

- Introduction.
- Interfaces.
- **Comparing objects**
- Implementations.
- Algorithms.

- Generic types.

equals()

- It is a method of the `Object` class
  - The standard way of comparing objects, instead of using “==“
- Any class may override it as required to define a custom notion of equality for instances of that class
- Overriding `equals()` facilitates searching in arrays, collections and maps
  - `<list>.contains(obj)`
  - Searching for an object or key will compare it against all objects in the collection by applying equals with each object as parameter.
- If each instance of a class in considered unique, there is no need to override `equals` (by default, it implements object reference equality, similarly to “==“).

equals()

- Any implementation of `equals` should satisfy the following properties
  - **Reflexive**: `this.equals(this)` must be true.
  - **Symmetric**: if x and y are two references, `x.equals(y)` is true if and only if `y.equals(x)` is true.
  - **Transitive**: let x, y and z be three references, if `x.equals(y)` is true, and `y.equals(z)` is true, then `x.equals(z)` must be true
  - **Consistency**: `x.equals(y)` must return the same when invoked repeatedly if none of the information of x and y involved in the equality check has changed
  - **Comparison with null**: `obj.equals(null)` must be false if `obj` is any object different from `null`

Example

```
public class VersionNumber {
   private int release;
   private int revision;
   private int patch;

   public VersionNumber(int release, int revision, int patch) {
      this.release = release;
      this.revision = revision;
      this.patch = patch;
   }
   @Override
   public String toString() {
      return "("+release+", "+revision+", "+patch+")";
   }
   @Override
   public boolean equals(Object obj) {
      if (obj==this) return true;
      if (!(obj instanceof VersionNumber)) return false;
      VersionNumber vn = (VersionNumber)obj;
      return vn.patch == this.patch &&
             vn.revision == this.revision &&
             vn.release == this.release;
   }
}
```

hashCode()

- *Hashing*: an efficient technique for storing and retrieving data (e.g., interface Map, Set)
- It requires to compute an *index* (hash code) for each object
- This computation uses a hash function
- If there are collisions (two or more different elements with the same hash code) the objects in collision are kept in a linked list

The general contract of hashCode

- **Consistency**: all invocations of `obj.hashCode()` must return the same value if `obj` has not been modified in any of its information used to define equality by `equals()`
- **Equality** by `equals()` implies same value returned by `hashCode()`
- **Inequality** by `equals()` does not necessarily imply a different value returned by `hashCode()`

Example

```
public class VersionNumber {
   private int release;
   private int revision;
   private int patch;

   //…
   @Override
   public boolean equals(Object obj) {
      if (obj==this) return true;
      if (!(obj instanceof VersionNumber)) return false;
      VersionNumber vn = (VersionNumber)obj;
      return vn.patch == this.patch &&
             vn.revision == this.revision &&
             vn.release == this.release;
   }
   @Override
   public int hashCode() {
      int hashValue = 11;
      hashValue = 31*hashValue+release;
      hashValue = 31*hashValue+revision;
      hashValue = 31*hashValue+patch;
      return hashValue;
   }
}
```

Consistency: both methods use patch, revision and release for the calculation

The interface Comparable<E>

- The *natural ordering* for instances of a class is specified by implementing the interface `Comparable<E>`
  - Method `compareTo(E o)`
  - Many predefined classes implement it: `String`, `Date`, `File`, etc.
  - Comparable objects can be used as elements in sorted sets, keys in a sorted map, and elements in lists ordered by `Collections.sort(alist)`
- A total ordering can also be specified using a comparator object that implements the interface `Comparator`

Example

```
public class VersionNumber implements Comparable<VersionNumber>{
   private int release;
   private int revision;
   private int patch;

   //...
   @Override
   public int compareTo(VersionNumber vno) {
      int releaseDiff = release-vno.release;
      if (releaseDiff!=0) return releaseDiff;
      int revisionDiff = revision-vno.revision;
      if (revisionDiff!=0) return revisionDiff;
      int patchDiff = patch-vno.patch;
      if (patchDiff!=0) return patchDiff;
      return 0;
   }
   @Override
   public boolean equals(Object obj) {
      if (obj==this) return true;
      if (!(obj instanceof VersionNumber)) return false;
      VersionNumber vn = (VersionNumber)obj;
      return vn.patch == this.patch &&
             vn.revision == this.revision &&
             vn.release == this.release;
   }
}
```

Example

```
public class VersionNumber implements Comparable<VersionNumber>{
   private int release;
   private int revision;
   private int patch;

   //...
   @Override
   public int compareTo(VersionNumber vno) {
      int releaseDiff = release-vno.release;
      if (releaseDiff!=0) return releaseDiff;
      int revisionDiff = revision-vno.revision;
      if (revisionDiff!=0) return revisionDiff;
      int patchDiff = patch-vno.patch;
      if (patchDiff!=0) return patchDiff;
      return 0;
   }
   @Override
   public boolean equals(Object obj) {
      if (obj==this) return true;
      if (!(obj instanceof VersionNumber)) return false;
      VersionNumber vn = (VersionNumber)obj;
      return this.compareTo(vn)==0;
   } // We can simplify equals by reusing compareTo
}
```

Exercise

- Create a class shoe, made of number, model and colour
- Add methods to compare (by model alphabeticaly, then colour alphabeticaly, and then by number in increasing order)

Contents

- Introduction
- Interfaces
- Comparing objects
- **Implementations**
- Algorithms

- Generic types

Implementations

Interfaces	Implemented as				
	Hash Table	Resizable Array	Tree	Linked List	Hash Table+ Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue				LinkedList PriorityQueue	
Map	HashMap		TreeMap		LinkedHashMap

Implementations of `Set`

- `HashSet`, `TreeSet` and `LinkedHashSet`.
  - HashSet: keeps elements “ordered” by hash
  - TreeSet: keeps elements order by natural order (requires comparable elements)
  - LinkedHashSet: keeps elements in insertion order
- We can initialize it to a given initial size:

```
Set<String> s = new HashSet<String>(64);
```

- The set grows as needed
- Two special purpose implementations:
  - `EnumSet`: containing enumeration objects (internal representation by an array of bits)
  - `CopyOnWriteArraySet`.

Implementations of `List`

- `ArrayList`, `LinkedList`, `Vector`.
  - ArrayList: uses internally an array to store elements
  - LinkedList: uses internally a doubly-linked list
  - Vector: An older class similar to ArrayList

- Special purpose implementations:
  - `CopyOnWriteArrayList`.

Implementations of `Map`

- `HashMap`, `TreeMap`, `LinkedHashMap`.
  - HashMap: Stores the keys using hash
  - TreeMap: Keeps ordered the keys using natural order (requires comparable keys)
  - LinkedHashMap: stores the keys using insertion order
- Special purpose implementations:
  - EnumMap: with keys of an enumeration type.
  - WeakHashMap: keys may eliminated (by garbage collections) when no longer referenced from elsewhere.
  - IdentityHashMap.
- Other concurrent implementations.

Contents

- Introduction
- Interfaces
- Comparing objects
- Implementations
- **Algorithms**

- Generic types

Array-Collection transformations

- Transform a collection into an array of Object

```
Collection c;
Object[] r= c.toArray();
```

- Transform a collection int a typed array

```
String[] s= c.toArray(new String[0]);
```

  - The empty array is used to inform `toArray` of the type of element of the array that it must return
  - At execution time, an exception is thrown if the type of the collection elements is not compatible (same or subtype) with the type used for the empty array

Array-Collection transformations

- Creating a list with specific initial contents

```
T a, b, c;
List<T> r=Arrays.asList(a, b, c);
```

- Transforming an array into a list

```
T[] a;
List<T> r= Arrays.asList(a);
```

- Both use the same method `asList`
  - Method `asList` admits a variable number of parameters, `T… a`, which are automatically converted into an array
  - If the list is modified, the array will be modified, **but resizing the list is not possible** since the list is only a view of the array and uses the array to store the elements

General purpose algorithms

- The classes `Arrays` and `Collections` contain static methods for several algorithms
- Sorting
  - `Collections.sort(List l)`. Uses merge sort according to the *natural order* of its elements (interface `Comparable`)
  - `Collections.sort(List l, Comparator c)`.

    Sorts the list according to de provided comparator.
- Shuffling (random order)
  - `Collections.shuffle(List l)`.

    Randomly permutes the list

Miscellaneous manipulations

- Collections.
  - `reverse(List l)`: Reverses the list
  - `fill(List<T>, T e)`: substitutes all elements in the list by element e.
  - `copy(List<T> dest, List<T> src)`: Copies all elements from list «src» to the list «dest»
  - `swap(List<T> l, int i, int j)`: Swaps the elements at the specified positions in the specified list.
  - `addAll(Collection<T> c, T… elementos)`: Adds all of the specified elements to the specified collection c

Search algorithms

- Collections.
  - `binarySearch(List<T> l, T e)`. The list must store all its elements in ascending order and T must implement `comparable<T>`
    - It accepts a comparator as a third parameter to be applied instead of the natural ordering
  - `frequency(Collection c, Object o)`: Returns the number of times an element o appears in the collection c
  - `disjoint(Collection a, Collection b)`: Returns true if and only if two collections are disjoint
  - `min, max`: Finding minumum and maximum values

Other methods in Collections

- `emptyList, emptyMap, emptySet`: Each returns an empty immutable collection.
- `singletonList, singletonMap`: Returns a list or a map with a single element or key-value pair, in both cases the return is immutable.
- `List<T> nCopies(int n, T e)`: Returns an inmutable list with n elements all of them are references to the same object «e».

Other methods in Collections

- `public static <T> List<T> unmodifiableList(List<? extends T> list)`
  - Creates and returns an unmodifiable view of the specified list. It is not a copy of the list (it uses delegation).
  - Useful for methods which need to return `List<A>` from an attribute of type `List<B>`, since `List<B>` is not a subtype of `List<A>` even if B is a subclass of A
  - Being inmutable it is not possible to add an element of type C, even if C is a subclass of A
  - There an analogous methods for `Map` and `Set`

Exercise

- Airport (final exam from 2014)

Contents

- Introduction
- Interfaces
- Comparing objects
- Implementations
- Algorithms

- **Generic types**

Generic Types

- A mechanism to create types that have other types as paremeters
- It allows us to create an implementation parameterized by diferente types. For example:
  - `Map<String, List<Integer>> m`: m is a map whose keys are of type `String` and whose values are of type `Integer`
- Without generic type:
  - `Map m`: m is a map whose keys are of type `Object` and whose values are of type `Object`.

How generic types work

- When defining a class (or a method) the generic types are declared between the symbols < and >

```
public interface List<E>{
    void add (E e);
    Iterator<E> iterator();
}
```

- The compiler is responsible for doing all type checking
- Only one class is created, and at execution time the type information is no longer available.
  - This limitation prevents basic types (int, double, …) from being used as generic types
- In C++, a different class is generated for each combination of generic class and types, allowing that basic types be used as generic parameters.

Example

```
public class Pair<A, B> {
    private A first;
    private B second;
    public Pair(A first, B second) {
        this.first=first;   this.second=second;
    }
    public int hashCode(){
        return first.hashCode()+second.hashCode();
    }
    public boolean equals(Object obj){
        if (obj instanceof Pair){
            Pair o=(Pair) obj;
            return first.equals(o.first) &&
            second.equals(o.second);
        }
        else return super.equals(obj);
    }
    public String toString(){
        return "{"+first+", "+second+"}";
    }
    public A getFirst() {
        return first;
    }

    public void setFirst(A first) {
        this.first = first;
    }

    public B getSecond() {
        return second;
    }

    public void setSecond(B second) {
        this.second = second;
    }
}
```

Limitation: Types A and B cannot be instantiated

Requirements of the parameter types

- In a generic class, we can require certain features from the parameters:

```
public class ReqComparable<A extends Comparable<A> & Serializable>
{
  private List<A> elems = new ArrayList<A>();

  public ReqComparable(A ...params) {
     elems.addAll(Arrays.asList(params));
     Collections.sort(elems);
  }

  @Override public String toString() {
    return this.elems.toString();
  }
}
```

Inheritance and generic types

- When we create a subclass of a generic class, we can:
  - Instanciate the type parameters:

```
public class ListStrings extends ArrayList<String>{
  …
} // ListStrings is not a generic type
```

  - Leave open the type parameters:

```
public class MyList<T> extends ArrayList<T>{
  …
} // MyList<T> is generic, and we need to instantiate
  // it when used to declare a variable
```

Inheritance and generic types

- What happens in this example?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

- Is there an error?
  - The type `List<String>` does not «inherit» from `List<Object>`

- Why not?

```
lo.add(new Object());
String s = ls.get(0);
```

  - If it did inherit from List<Object> objects of the wrong type could be added to it. Java does not have the notion of constant objects (all constants are of basic types), and so the compiler could not ckeck this type of errors

Inheritance and generic types

- Using Object as parameter

```
public void print(List<Object> l){
   // code for printing
}
```

- Can we then pass a list of any type as a parameter to print?
  - No, we cannot. The above parameter is more restrictive than using print(List l), since there is no inheritance from List<Object> to lists of any other types

- Solution: Add flexibility to parameters using the wildcard type ?

```
public void print(List<?> l){
   // code for printing
}
```

The wildcard ?

- Within the method print we can access any element of the list with get

```
List<?> l;
Object o= l.get(0);
```

  - It works: although `get` returns an object of the unknown type «?», all types inherit from `Object`

- However, can we used `add`?

```
Object o;
l.add(o);
```

- List elements are of an unknown type, and so we cannot use `add` with an instance of `Object` since `Object` does not inherit from «?». We will not be able to invoke any method which takes a parameter of an unknown type

Wildcard: allowing inheritance

```
public void addVehicles(List<? extends Vehicle> l){
  //add all vehicles assuming attribute List<Vehicle > v;
  v.addAll(l);
}
```

- Now we are allowed to do this:

```
List<Car> c;
o.addVehicles(c);
//Works if Car extends from Vehicle
```

Wildcard super

- `? super T` denotes an unknown type which must be supertype of T (or T).
- For instance, the following method works fine:

```
public void insert(List<? super Number> aList) {
  aList.add(8);
  aList.add(new Float(8));
}
```

- But this method does not, why?

```
public void insert(List<? extends Number> aList) {
  aList.add(8);
  aList.add(new Float(8));
}
```

Wildcard super

```
public class ComparablePair<A extends Comparable<? super A>, B extends Comparable<? super B>>
extends Pair<A, B> implements
    Comparable<ComparablePair<A, B>>{

    /** Creates a new instance of ComparablePair */
    public ComparablePair(A a, B b) {
        super (a, b);
    }

    public int compareTo(ComparablePair<A, B> o){
        int r1=getFirst().compareTo(o.getFirst());
        if (r1!=0) return r1;
        else return getSecond().compareTo(o.getSecond());
    }
}
```

Example

```
public interface Transform <A, B> {
// convert from A to B using a function
    B transform (A element);
}
…
public class StringTransform implements Transform<Object, String> {
    public String transform(Object element){
      return element.toString();
    }
}
```

Example

```
public class TransformList<A, B> extends AbstractList<B> implements List<B>{
    private Transform<? super A,? extends B> transformer;
    private List<? extends A> l;
    /** Constructor creates a transformer view of the given list, by means of a transformer*/
    public TransformList(List<? extends A> l, Transform<? super A,? extends B> transformer) {
        this.l=l;
        this.transformer=transformer;
    }
    // We use delegation of methods get and size, only methods required by List using AbstractList
    public B get(int pos){
        return transformer.transform(l.get(pos));
    }
    public int size(){
        return l.size();
    }
}
```

Generic methods

- Methods can also be generic in Java

```
public static <T> void copyToArray(T[] array,
                                   List<? extends T> aList)
{
  int i = 0;
  for (T t : aList)
    array[i++] = t;
}

…
List<String> ls = new ArrayList<String>();
ls.add("element");
ls.add("element2");

Object[] array = new Object[2];
copyToArray(array, ls);
```

Generic methods

- Methods with more than one generic type

```
public static <T, S extends T>
   boolean copyArray(T[] dest, S[] src)
   {
      if (dest.length < src.length) return false;
      for (int i= 0; i<src.length; i++)
         dest[i] = src[i];
      return true;
   }
…
String src[] = {"two", "strings"};
Object dest[] = new Object[2];
copyArray(dest, src);
```

Generic methods

- Explicit binding of generic type at invocation:

```
public class Util {
  public static <R> R as(List<? super R> aList, int pos) {
      return (R)aList.get(pos); // unchecked!!
  }
}
...
List<Object> list2 = new ArrayList<Object>();
list2.add(3);

int n = 2 + Util.<Integer>as(list2, 0);

System.out.println(n);
```