Course of Programming in Java

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

The Java Programming Language

Chapter 11: Generic Types

Chapter 21: Collections

by Ken Arnold, James Gosling, David Holmes

Covariance and Contravariance

```
class Fruit { }
class Apple extends Fruit {
    public void bite () {
        System.out.println ("Apple.bite");
    }
}
class Farmer {
    public Apple sell () {
        return new Apple();
    }
}
class Gardener extends Farmer {
    public Fruit sell () {
        return new Fruit();  // does not compile
    }
}
public class Lec6a {
    static void tasteFrom (Farmer f) {
        Apple a = f.sell ();
        a.bite ();
    }
    public static void main (String[] args) {
        tasteFrom (new Gardener ()); }
}
```

- `class S extends T` means `S` is a subtype of `T`
  - type theorists write this `S <: T`
  - "any term of type `S` can safely be used in a context where a term of type `T` is expected" or "every value described by `S` is also described by `T`"
- **Covariant** - Covariance is when a more specific type, `S`, can be used when a more generic type, `T`, is specified.
  - A function that returns `S` can be used in the same context as a function that returns `T`.
- **Contravariant** - When the more generic type, `T`, can be used where the more specific type, `S`, is specified.
  - A function that takes a `T` can be used in the same context as a function that takes a `S`.
- **Invariant** - The type specified is the only type that can be used.

Covariance example (covariant in return type) (fixed)

```
class Fruit { }
class Apple extends Fruit {
    public void bite () {
        System.out.println ("Apple.bite");
    }
}
class Farmer {
    public Fruit sell () {
        return new Fruit();
    }
}
class Gardener extends Farmer {
    public Apple sell () {
        return new Apple();
    }
}
public class Lec6a {
    static void tasteFrom (Farmer f) {
        Fruit a = f.sell ();
        // a.bite ();
    }
    public static void main (String[] args) {
        tasteFrom (new Gardener ());
    }
}
```

Contravariance example (broken)

```
class Vegetables {
    public void plate () {
        System.out.println ("Plate of vegetables.");
    }
}
class FishWithVegs extends Vegetables {
    public void plate () {
        System.out.println ("Fish on a plate.");
    }
}
class Vegetarian {
    public void eat (Vegetables vegs) {
        vegs.plate();
    }
}
class Pescatarian extends Vegetarian {
    public void eat (FishWithVegs fish) {
        fish.plate();
    }
}
public class Lec6b {
    public static void main (String[] args) {
        (new Vegetarian()).eat (new FishWithVegs ());
    }
}
```

Contravariance example (contravariant in argument type) (fixed)

```
class FishOrVegs {
    public void plate () {
        System.out.println ("Fish on a plate.");
    }
}
class Vegetables extends FishOrVegs {
    public void plate () {
        System.out.println ("Plate of vegetables.");
    }
}
class Vegetarian {
    public void eat (Vegetables vegs) {
        vegs.plate();
    }
}
class Pescatarian extends Vegetarian {
    public void eat (FishOrVegs fish) {
        fish.plate();
    }
}
public class Lec6c {
    public static void main (String[] args) {
        (new Pescatarian()).eat (new Vegetables ());
    }
}
```

Well, but it still does not work as expected!

```
class FishOrVegs {
    public void plate () {
        System.out.println ("Fish on a plate.");
    }
}
class Vegetables extends FishOrVegs {
    public void plate () {
        System.out.println ("Plate of vegetables.");
    }
}
class Vegetarian {
    public void eat (Vegetables vegs) {
        System.out.print ("Vegetarian: ");
        vegs.plate();
    }
}
class Pescatarian extends Vegetarian {
    public void eat (FishOrVegs fish) {
        System.out.print ("Pescatarian: ");
        fish.plate();
    }
}
public class Lec6c {
    public static void main (String[] args) {
        (new Pescatarian()).eat (new Vegetables ());
    }
}
```

Finally a workaround:

```
class FishOrVegs {
    public void plate () {
        System.out.println ("Fish on a plate.");
    }
}
class Vegetables extends FishOrVegs {
    public void plate () {
        System.out.println ("Plate of vegetables.");
    }
}
class Vegetarian {
    public void eat (Vegetables vegs) {
        System.out.print ("Vegetarian: ");
        vegs.plate();
    }
}
class Pescatarian extends Vegetarian {
    public void eat (FishOrVegs fish) {
        System.out.print ("Pescatarian: ");
        fish.plate();
    }
    public void eat (Vegetables vegs) {
        eat ((FishOrVegs)vegs);
    }
}
public class Lec6d {
    public static void main (String[] args) {
        Vegetarian client = new Pescatarian();
        client.eat (new Vegetables ()); }
}
```

Generic Types

```
class Cell<E> {
    private Cell<E> next;
    private E element;
    public Cell(E element) {
        this.element = element;
    }
    public Cell(E element, Cell<E> next) {
        this.element = element;
        this.next = next;
    }
    public E getElement() {
        return element;
    }
    public void setElement(E element) {
        this.element = element;
    }
    public Cell<E> getNext() {
        return next;
    }
    public void setNext(Cell<E> next) {
        this.next = next;
    }
}
```

- `E` is a type variable, it can be any identifier like `ElementType`, but single letters are good style (`E` for an element type, `K` for a key type, `V` for a value type, `T` for a general type).
- A generic type declaration can contain multiple type parameters, separated by commas, e.g. `interfaceMap<K, V>`
- `Cell<E>` is read as "`Cell` of `E`".
- To create an actual `Cell` tell what specific type to replace `E` with, e.g.

```
Cell<String> strCell = new Cell<String>("Hello");
```

- `Cell<String>` and `Cell<Number>` are not two separate classes.
  - At runtime, no generic type information is present in objects.

```
class SingleLinkQueue<E> {
    protected Cell<E> head;
    protected Cell<E> tail;
    public void add(E item) {
        Cell<E> cell = new
Cell<E>(item);
        if (tail == null)
            head = tail = cell;
        else {
            tail.setNext(cell);
            tail = cell;
        }
    }

    public E remove() {
        if (head == null)
            return null;
        Cell<E> cell = head;
        head = head.getNext();
        if (head == null)
            tail = null; //empty queue
        return cell.getElement();
    }
}
```

...

```
SingleLinkQueue<String> queue =
    new SingleLinkQueue<String>();
queue.add("Hello");
queue.add("World");
```

Now there is no need for a cast when invoking `remove`:

```
String hello = queue.remove();
```

Nor is it possible to add the wrong kind of element to the queue:

```
queue.add(25);    // INVALID
```

Limitations Related to Type Erasure

- A generic class with a type parameter `E` cannot use `E` in the type of a static field or anywhere within a static method or static initializer.
- If `SingleLinkQueue<E>` has a static `merge` method, it must be invoked as `SingleLinkQueue.merge`.
- You cannot instantiate `E` or create an array of `E`: There is a single class with a single definition of `toArray` below, and the compiler has to know at compile time what code to generate to create any objects or arrays.

Question 1. *Why creating an array here is a problem? It doesn't need to create an `E` object since "arrays are initialized to `null`".*

```
class SingleLinkQueue<E> {
    // ...
    public E[] toArray() {
        int size = 0;
        for (Cell<E> c = head; c != null; c = c.getNext())
             size++;
        E[] arr = new E[size];   // INVALID: won't compile
        // ... copy in elements ...
    }
}
```

In an inner class, the type variables of the outer class declaration are accessible to it and can be used directly, e.g.:

```
class SingleLinkQueue<E> {
    class Cell {
        private Cell next;
        private E element;
        public Cell(E element) {
            this.element = element;
        }
        public Cell(E element,
                    Cell next) {
            this.element = element;
            this.next = next;
        }
        public E getElement() {
            return element;
        }
        /* ..rest of Cell methods.. */
    }

    protected Cell head;
    protected Cell tail;

    public void add(E item) {
        Cell cell = new Cell(item);
        if (tail == null)
            head = tail = cell;
        else {
            tail.setNext(cell);
            tail = cell;
        }
    }

    public E remove() {
        if (head == null)
            return null;
        Cell cell = head;
        head = head.getNext();
        if (head == null)
            tail = null; //empty queue
        return cell.getElement();
    }
    /* ... rest of methods ... */
}
```

Bounded Type Parameters and Variance

- A sorted collection would restrict its type parameter to be a type that implements `Comparable` (`Comparable` is an *upper bound* of `E` below):

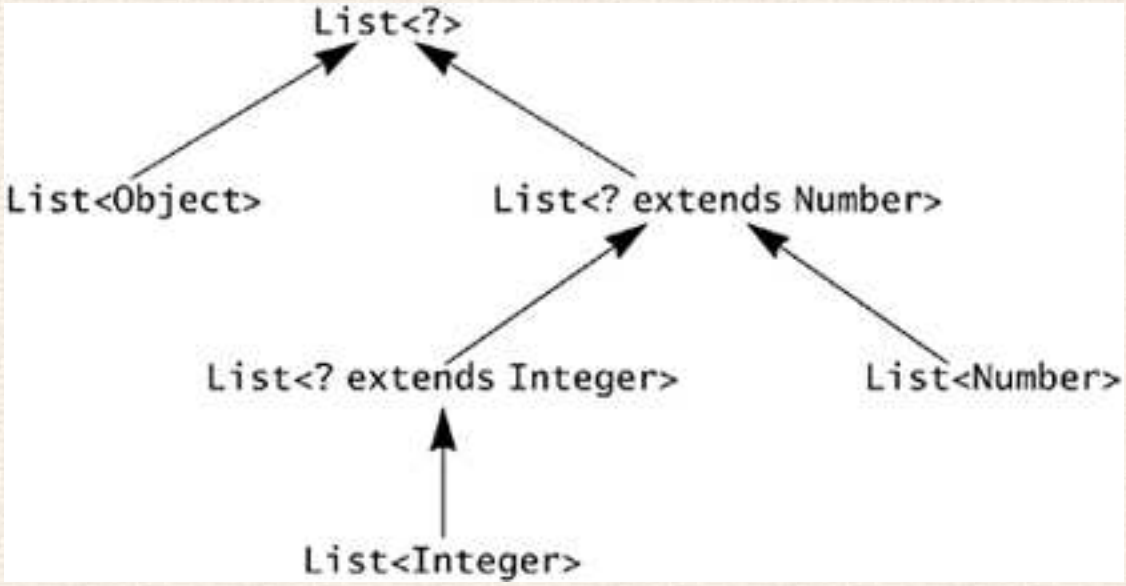
```
interface SortedCollection<E extends Comparable<E>> {
    // ... sorted collection methods ...
}
```

- Generic type parameters are *invariant*: even though `Integer` is a subtype of `Number`, `List<Integer>` is not a subtype of `List<Number>`. Contrast this with arrays, where `Integer[]` is a subtype of `Number[]`.

- We can declare a `List` of an arbitrary element type that is compatible with `Number` using the type argument wildcard '?':

```
static double sum(List<? extends Number> list) {
    double sum = 0.0;
    for (Number n : list)
        sum += n.doubleValue();
    return sum;
}
```

- Wildcard upper bounds `<? extends E>` are *covariant*, just like arrays



- Wildcard lower bounds `<? super E>` are *contravariant*: when the bound gets more concrete, the type gets more general.

```
interface SortedCollection<E extends Comparable<? super E>> {
    // ... sorted collection methods ...
}
```

If class `Value` implements `Comparable<Object>` then it can still correctly compare two `Value` objects, in fact it can do more than that.

- Because the wildcard represents an unknown type, you can't do anything that requires the type to be known (unbounded or upper-bounded case):

```
SingleLinkQueue<?> strings =
    new SingleLinkQueue<String>();
strings.add("Hello");                // INVALID: won't compile

SingleLinkQueue<? extends Number> numbers =
    new SingleLinkQueue<Number>();
numbers.add(Integer.valueOf(25));    // INVALID: won't compile
```

- Given a lower-bound, the wildcard is known to be the same as, or a super type of, the bound:

```
static void addString(SingleLinkQueue<? super String>
sq) {
    sq.add("Hello"); // OK: sq handles strings (or more)
}
```

Example: Producer Extends

```
public class Stack<E> {
    public Stack() { /* ... */ }
    public void push(E e) { /* ... */ }
    public E pop() { /* ... */ }
    public boolean isEmpty() { /* ... */ }

    // Wildcard type for parameter that is an E producer
    public void pushAll(Iterable<? extends E> src) {
        for (E e : src) push(e);
    }
}

// ...

Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

Example: Consumer Super

```
public class Stack<E> {
    public Stack() { /* ... */ }
    public void push(E e) { /* ... */ }
    public E pop() { /* ... */ }
    public boolean isEmpty() { /* ... */ }

    // Wildcard type for parameter that is an E consumer
    public void popAll(Collection<? super E> dst) {
        while (!isEmpty())
            dst.add(pop());
    }
}

// ...

Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

Generic Methods and Constructors

- associating parameter type with return type:

```
public <T> T[] toArray(T[] arr) {
    Object[] tmp = arr;
    int i = 0;
    for (Cell<E> c = head;
         c != null && i < arr.length;
         c = c.getNext())
        tmp[i++] = c.getElement();
    return arr;
}
```

- associating two parameter types::

```
public static <E> void merge(SingleLinkQueue<E> d,
                             SingleLinkQueue<? extends E> s)
{ /* ... merge s elements into d ... */ }
```

- Type parameters for methods are inferred, but can be passed explicitly: it requires using qualified names (e.g. with a variable, `this`, or `super`)

```
SingleLinkQueue<String> q = this.<String>merge(d, s);
```

- *Raw types*: for compatibility with old code, type parameters can be omitted: `List` is roughly equivalent to `List<?>` (use the generic form in new code).
- The *erasure* of a type variable is the erasure of its bound (for unbounded, `Object`); signature erasures:

```
class Base<T> {
   void m(int x)    { }
   void m(T t)      { }
   void m(String s) { }
   <N extends Number> void m(N n)
       { }
   void m(SingleLinkQueue<?> q)
       { }
}
```

Resulting run-time method signatures:

```
void m(int x)    { }
void m(Object t) { }
void m(String s) { }
void m(Number n) { }
void m(SingleLinkQueue q) { }
```

- A class cannot have several methods with the same signature erasure.
- Cannot override a nongeneric method with generic one.
- Informally, one method is more specific than another if all calls to the first method could be handled by the second. Most specific is selected.

Example (argument type covariant in method type parameter)

```
public class Union {
    public static <E> Set<E> union(Set<? extends E> s1,
                                   Set<? extends E> s2)
    { /* ... */ }
}
```

we can:

```
Set<Integer> integers = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers =
    Union.<Number>union(integers, doubles);
```

(We need to provide above the type parameter using `Union.<Number>union` because type inference is insufficient.)

Class Extension and Generic Types

- General extension:

```
class GeneralList<E> implements List<E> { /* ... */ }
```

- Nongeneric extension of a particular instance:

```
class StringList implements List<String> { /* ... */ }
```

- Generic extension of nongeneric type:

```
class LocalEventService<T extends Event> extends
                        AbstractEventService { /* ... */ }
```

- A class can implement only one raw variant of a generic interface.

```
class Value implements Comparable<Value> { /* ... */ }
class ExtendedValue extends Value
    implements Comparable<ExtendedValue> { /* INVALID! */ }
```

  - Here it should be `class Value implements Comparable<? super Value>`

Collections

Collections (aka. *containers*) are mostly in the `java.util` package. It has interfaces:

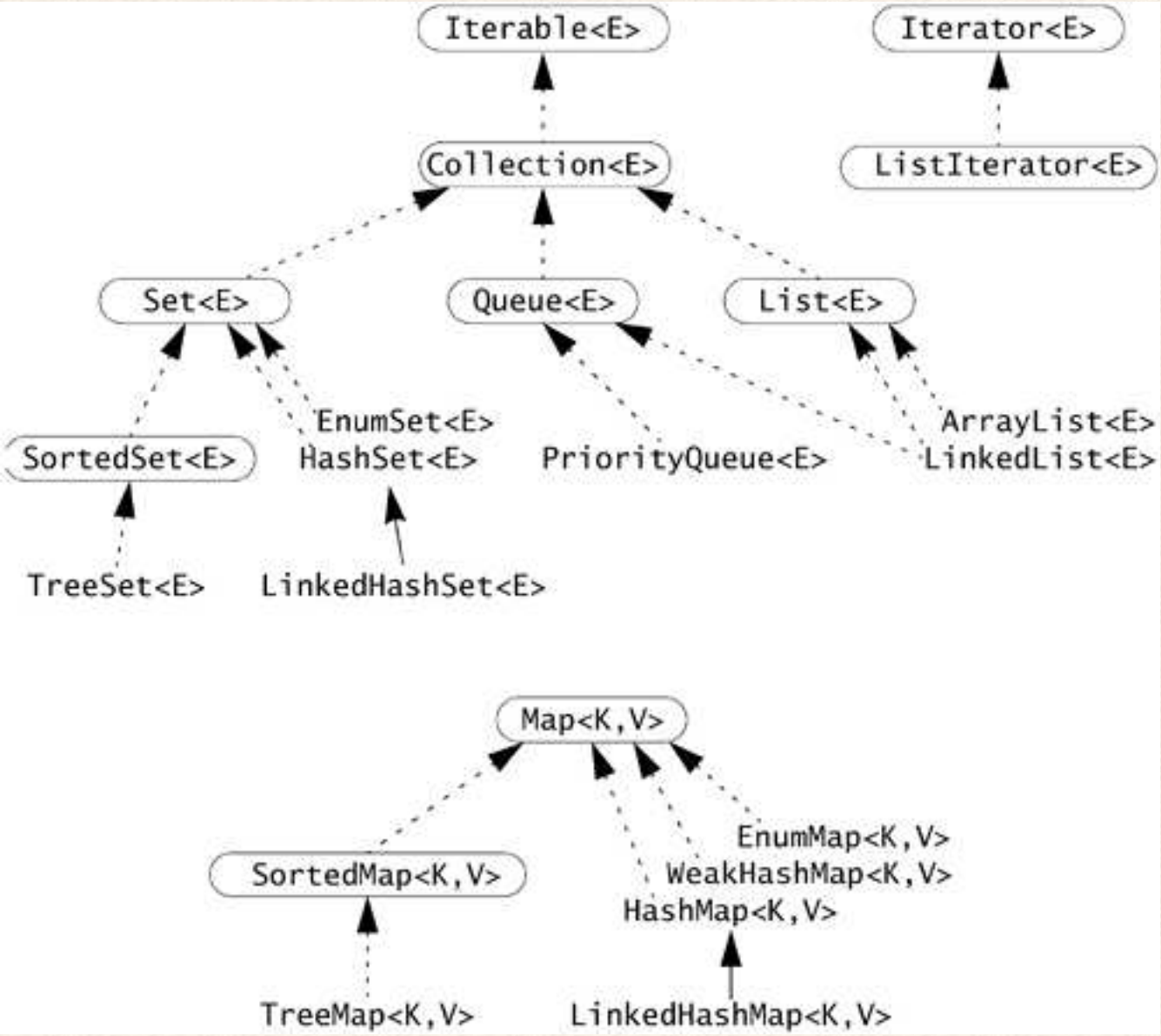
- `Collection<E>` provides `add`, `remove`, `size`, `toArray`...
- `Set<E>`: no duplicate elements can be present, elements not necessarily stored in any particular order.
- `SortedSet<E>` A set whose elements are sorted (extends `Set<E>`).
- `List<E>`: stay in a particular order unless the list is modified.
- `Queue<E>` A collection with an implied ordering in its elements, has a head element and operations like `peek` and `poll`.
- `Map<K,V>` A mapping from keys to at most one value each (does not extend `Collection`, although maps can be viewed as collections.)
- `SortedMap<K,V>` A map whose keys are sorted (extends `Map<K,V>`).
- `Iterator<E>` An interface for objects that return elements from a collection one at a time.

- `ListIterator<E>` An iterator for `List` objects that adds useful `List`-related methods. Returned by `List.listIterator`.
- `Iterable<E>` An object that provides an `Iterator` and so can be used in an enhanced `for` statement. (Defined in the `java.lang` package.)

Specific implementations can refuse to execute some operations by throwing the unchecked `java.lang.UnsupportedOperationException`.

- `HashSet<E>` General-purpose implementation for `Set`: searching, adding, and removing are mostly insensitive to the size of the contents.
- `TReeSet<E>` A `SortedSet` implemented as a balanced binary tree.
- `ArrayList<E>` A `List` implemented using a resizable array.
- `LinkedList<E>` A doubly linked `List` and `Queue` implementation.
- `HashMap<K,V>` Cheap lookup and insertion times.
- `TreeMap<K,V>` `SortedMap` as a balanced binary tree ordered by key.

Methods that return individual elements of a collection will give you a `NoSuchElementException` if the collection is empty.


Iterable<E>
Iterator<E>
Collection<E>
ListIterator<E>
Set<E>
Queue<E>
List<E>
EnumSet<E>
HashSet<E>
ArrayList<E>
LinkedList<E>
SortedSet<E>
PriorityQueue<E>
TreeSet<E>
LinkedHashSet<E>
Map<K,V>
EnumMap<K,V>
SortedMap<K,V>
WeakHashMap<K,V>
HashMap<K,V>
TreeMap<K,V>
LinkedHashMap<K,V>

Iteration

`Collection<E>` extends `Iterable<E>`, which defines an `iterator` method that returns an object that implements the `Iterator<E>` interface:

`public boolean hasNext()`

Returns `true` if the iteration has more elements.

`public E next()`

Returns the next element in the iteration, or throws `NoSuchElementException`.

`public void remove()`

Removes the element returned most recently. Can be called only once per call of `next`, otherwise throws `IllegalStateException`. Optional.

Enhanced `for` loop cannot be used when removing elements.

```
public void removeLongStrings
    (Collection<? extends String> coll, int maxLen) {
    Iterator<? extends String> it = coll.iterator();
    while (it.hasNext()) {
        String str = it.next();
        if (str.length() > maxLen)
            it.remove();
    }
}
```

Do not modify a collection outside of an iterator when it is in use.

Many iterators are *fail-fast*: throw `ConcurrentModificationException` rather than risk performing an action after collection has been modified outisde of the iterator.

ListIterator

You can iterate forward using `hasNext` and `next`, backward using `hasPrevious` and `previous`, or index into positions from `0` to `list.size()-1`.

```
ListIterator<String> it = list.listIterator(list.size());
while (it.hasPrevious()) {
    String obj = it.previous();
    System.out.println(obj);
    // ... use obj ...
}
```

`nextIndex` or `previousIndex` get the index of the element returned by subsequent `next` or `previous` call.

More in `ListIterator`:

`public void set(E elem)`

Replaces the last element returned by `next` or `previous` with `elem`. Optional.

`public void add(E elem)`

Inserts `elem` into the list in front of the next element that would be returned, or at the end if `hasNext` returns `false`. The iterator is moved forward; if you invoke `previous` after an `add` you will get the added element. Optional.

Ordering with `Comparable` and `Comparator`

The interface `java.lang.Comparable<T>`:

> `public int compareTo(T other)`
>
> > Returns a value that is less than, equal to, or greater than zero as this object is less than, equal to, or greater than the `other` object. Return zero only if `equals` with the same object would return `true`.

The ordering defined by `compareTo` is a class's natural ordering.

If a given class does not implement `Comparable` or if its natural ordering is wrong for some purpose, you can often provide a `java.util.Comparator` object instead. The `Comparator<T>` interface has the method

> `public int compare(T o1, T o2)`
>
> > Ordering as in `Comparable.compareTo`.

Using Hashed Collections (`HashSet` & `HashMap`)

- Hashed collections are based on `equals` and `hashCode` methods, whose default implementations in the class `Object` use physical identity.
- Library classes like `String` implement `equals` and `hashCode` using "natural equivalence", e.g. comparing the content of the objects.
- `equals` should be *reflexive*, *symmetric*, *transitive*, and *consistent* (over time), and should return `false` on `null`.
  - Do not handle objects of a superclass specially! (breaks symmetry)

```
@Override public boolean equals(Object o) {
  return o instanceof CaseInsensitiveString && // no special treatment for String
      ((CaseInsensitiveString) o).s.equalsIgnoreCase(s); }
```

  - Unfortunately, **there is no way to extend an instantiable class and add a value component while preserving the requirements**, so be careful (or use composition). (*Effective Java* by Joshua Bloch item 8)
- Always override `hashCode` when you override `equals` (*Effective Java* by Joshua Bloch item 9):

1. Store a constant nonzero value, say, 17, in an int variable called `result`.
2. For each significant field `f` in your object (each field taken into account by the `equals` method, that is), do the following:
    a. Compute an `int` hash code `c` for the field:
        i. If the field is a boolean, compute `(f ? 1 : 0)`.
        ii. If the field is a `byte`, `char`, `short`, or `int`, compute `(int) f`.
        iii. If the field is a `long`, compute `(int) (f ^ (f >>> 32))`.
        iv. If the field is a `float`, compute `Float.floatToIntBits(f)`.
        v. If the field is a `double`, compute `Double.doubleToLongBits(f)`, and then hash the resulting long as in step 2.a.iii.
        vi. If the field is an object reference and this class's `equals` method compares the field by recursively invoking `equals`, recursively invoke `hashCode` on the field.

vii. If a more complex comparison is required, compute a "canonical representation" for this field and invoke `hashCode` on the canonical representation. If the value of the field is null, return 0 (or some other constant, but 0 is traditional).

viii. If the field is an `array`, treat it as if each element were a separate field, or (if every element is significant) use one of the `Arrays.hashCode` methods added in release 1.5.

b. Combine the hash code c computed in step 2.a into result as follows:
`result = 31 * result + c;`

3. Return result.

4. When you are finished writing the `hashCode` method, ask yourself whether equal instances have equal hash codes. Write unit tests to verify your intuition! If equal instances have unequal hash codes, figure out why and fix the problem.

The `Collection` Interface

`public int size()`

Returns the number of elements the collection currently holds.

`public boolean isEmpty()`

`public boolean contains(Object elem)`

Returns `true` if this collection has an element on which invoking `equals` with `elem` returns `true`.

`public Iterator<E> iterator()`

Returns an iterator that steps through the elements of this collection.

`public Object[] toArray()`

`public <T> T[] toArray(T[] dest)`

Returns an array that contains all the elements of this collection, either `dest` or a new one if `dest` is too small.

`public boolean add(E elem)`

Makes sure that this collection contains the object `elem`, returning `true` if this required changing the collection – always if the collection allows duplicates. Optional.

`public boolean remove(Object elem)`

Removes a single instance of `elem` from the collection, returning `true` if the element existed in this collection. Optional.

Preserving types when converting to arrays: either

```
String[] strings = new String[collection.size()];
strings = collection.toArray(strings);
```

or

```
String[] strings = collection.toArray(new String[0]);
```

Operating in bulk

`public boolean containsAll(Collection<?> coll)`

> `true` if the collection contains each of the elements in `coll`.

`public boolean addAll(Collection<? extends E> coll)`

> Adds each element of `coll` to this collection, returning `true` if any addition required changing the collection. Optional.

`public boolean removeAll(Collection<?> coll)`

> `true` if any removal required changing the collection. Optional

`public boolean retainAll(Collection<?> coll)`

> Removes from this collection all elements that are not elements of `coll`, returning `true` if any removal required changing the collection. Optional.

`public void clear()`

> Removes all elements from this collection. Optional.

Set and SortedSet

Set<E> provides no additional methods, but a Set has no duplicate elements

The iterators on a SortedSet<E> collection will always return the elements in a specified order (by default, the element's natural order).

SortedSet<E> adds some methods that make sense in an ordered set:

```
public Comparator<?  super E> comparator()
```

Returns the Comparator being used by this sorted set, or null if the elements' natural order is being used.

```
public E first()
```

Returns the first (lowest) object in this set.

```
public E last()
```

Returns the last (highest) object in this set.

`public SortedSet<E> subSet(E min, E max)`

Returns a view of the set that contains all the elements of this set whose values are greater than or equal to `min` and less than `max`. Changes to the collection that fall within the range will be visible through the returned subset and vice versa. Cannot modify the returned set to contain an element that is outside the specified range.

`public SortedSet<E> headSet(E max)`

As above, but all values which are less than the value of `max`.

`public SortedSet<E> tailSet(E min)`

As above, but all values which are greater than or equal to the value of `min`.

You can create snapshots by making copies of the view, as in

```
public <T> SortedSet<T> copyHead(SortedSet<T> set, T max) {
    SortedSet<T> head = set.headSet(max);
    return new TreeSet<T>(head); // contents from head
}
```

HashSet<E> and LinkedHashSet<E>

`public HashSet(int initialCapacity, float loadFactor)`

Creates a new `HashSet` with `initialCapacity` hash buckets and the given `loadFactor`. When the ratio of the number of elements in the set to the number of hash buckets is greater than or equal to the load factor, the number of buckets is increased.

`public HashSet(int initialCapacity)`

Uses a default load factor.

`public HashSet()`

Default initial capacity and load factor.

`public HashSet(Collection<? extends E> coll)`

Creates a new `HashSet` whose initial contents are the elements in `coll`. The initial capacity is based on the size of `coll`, and the default load factor is used.

`LinkedHashSet` iterates over elements in the order they were added.

`TreeSet<E>`

Stores its contents in a balanced tree.

`public TreeSet()`

Creates a new `treeSet` that is sorted according to the natural order of the element types.

`public TreeSet(Collection<? extends E> coll)`

Use `TreeSet()` and then add the elements of `coll`.

`public TreeSet(Comparator<? super E> comp)`

A new `treeSet` that is sorted in the `comp` order.

`public TreeSet(SortedSet<E> set)`

Creates a new `treeSet` with the same initial contents and sorted in the same way as `set`.

List<E>

`public E get(int index)`

Returns the `index`th entry in the list.

`public E set(int index, E elem)`

Sets the indexth entry in the list to `elem`, replacing the previous element and returning it. Optional.

`public void add(int index, E elem)`

Adds `elem` to the list at the `index`th position, shifting every element farther in the list down one position. Optional.

`public E remove(int index)`

Removes and returns the `index`th entry in the list, shifting every element farther in the list up one position. Optional.

`public int indexOf(Object elem)`

Returns the index of the first object in the list that is `equal` to `elem`. Returns -1 if no match is found.

`public int lastIndexOf(Object elem)`

Returns the index of the last object in the list that is `equal` to `elem`. Returns -1 if no match is found.

`public List<E> subList(int min, int max)`

Returns a `List` that is a view on this list over the range, starting with `min` up to, but not including, `max`. Changes made to the returned list are reflected in this list. Do not change the underlying list while using a sublist.

`public ListIterator<E> listIterator(int index)`

Iterate through the list starting at the `index`th entry.

`public ListIterator<E> listIterator()`

Iterate through the list starting at the beginning.

`ArrayList<E>`

- Fast: Adding and removing elements at the end, getting the element at a specific position.
- Slow: Adding and removing elements from the middle: $O(n - i)$ where n is the size of the list and i is the position of the element being removed.

`public ArrayList()`

Creates a new `ArrayList` with a default capacity.

`public ArrayList(int initialCapacity)`

Creates a new `ArrayList` that initially can store `initialCapacity` elements without resizing.

`public ArrayList(Collection<? extends E> coll)`

Creates a new `ArrayList` whose initial contents are the contents of `coll`. The capacity of the array is initially 110% of the size of `coll` to allow for some growth without resizing. The order is that returned by the collections iterator.

`public void trimToSize()`

Sets the capacity to be exactly the current size of the list (allocates a smaller array if needed).

`public void ensureCapacity(int minCapacity)`

Sets the capacity to `minCapacity` if the capacity is currently smaller. You can use this to ensure the array will be reallocated at most once when adding a large number of elements to the list.

`LinkedList<E>`

- Fast: Adding or removing an element in the middle is $O(1)$ because it requires no copying.
- Slow: getting the element at a specific position i is $O(i)$ since it requires starting at one end and walking through the list to the i^{th} element.

```
public LinkedList()
public LinkedList(Collection<? extends E> coll)
```

Elements of `coll` in the order of the collection's iterator.

```
public E getFirst()
public E getLast()
public E removeFirst()
public E removeLast()
public void addFirst(E elem)
```

Adds `elem` into this list as the first element.

```
public void addLast(E elem)
```

Queues

The `Queue<E>` interface extends `Collection<E>`: defines a head position, which is the next element that would be removed. Queues often operate as first-in-first-out, but can also be last-in-first-out (commonly known as stacks) or have a specific ordering defined by a comparator.

`public E element()`

Returns, but does not remove, the head of the queue. If the queue is empty a `NoSuchElementException` is thrown.

`public E peek()`

Returns, but does not remove, the head of the queue. If the queue is empty, `null` is returned.

`public E remove()`

Returns and removes the head of the queue. If the queue is empty a `NoSuchElementException` is thrown.

`public E poll()`

Returns and removes the head of the queue. If the queue is empty, `null` is returned.

The `LinkedList` class provides the simplest implementation of `Queue`.

`PriorityQueue`

`PriorityQueue<E>` is an unbounded queue, based on a priority heap.

`public PriorityQueue(int initialCapacity)`

Can store `initialCapacity` elements without resizing, uses the natural order of the elements.

`public PriorityQueue()`

Default initial capacity, uses the natural order of the elements.

`public PriorityQueue(int initialCapacity, Comparator<? super E> comp)`

Orders the elements according to the supplied comparator.

`public PriorityQueue(Collection<? extends E> coll)`

Creates a new `PriorityQueue` whose initial contents are the contents of `coll`. The capacity of the queue is initially 110% of the size of `coll`. If `coll` is a `SortedSet` or another `PriorityQueue`, then this queue will be ordered the same way; otherwise, the elements will be sorted according to their natural order.

Map<K,V> and SortedMap<K,V>

`public int size()`

Returns the size of this map, that is, the number of key/value mappings it currently holds.

`public boolean isEmpty()`

`public boolean containsKey(Object key)`

`true` if the collection contains a mapping for the given `key`.

`public boolean containsValue(Object value)`

the collection contains at least one mapping to the `value`.

`public V get(Object key)`

The object to which `key` is mapped, `null` if it is not mapped.

`public V put(K key, V value)`

Associates `key` with the given value in the map. If a map already exists for `key`, its value is changed and the original value is returned. If no mapping exists, `put` returns `null`. Optional.

`public V remove(Object key)`

Removes any mapping for the `key`. The return value has the same semantics as that of `put`. Optional.

`public void putAll(Map< ? extends K, ? extends V> otherMap)`

Puts all the mappings in `otherMap` into this map. Optional.

`public void clear()`

Removes all mappings. Optional.

`containsValue` will often be slow, i.e. $O(n)$.

Viewing the map using collections:

`public Set<K> keySet()`

Returns a `Set` whose elements are the keys of this map.

`public Collection<V> values()`

A `Collection` whose elements are the values of this map.

`public Set<Map.Entry<K,V>> entrySet()`

Returns a `Set` whose elements are `Map.Entry` objects that represent single mappings in the map.

The interface `Map.Entry<K,V>` defines:

`public K getKey()`

`public V getValue()`

`public V setValue(V value)`

Sets the value for this entry and returns the old value.

`SortedMap` adds methods that make sense in an ordered map:

```
public Comparator<?  super K> comparator()
```

Returns the comparator being used for sorting this map, or `null` if the map is sorted using the keys' natural order.

```
public K firstKey()
```

```
public K lastKey()
```

```
public SortedMap<K,V> subMap(K minKey, K maxKey)
```

Returns a view of the portion of the map whose keys are greater than or equal to `minKey` and less than `maxKey`.

```
public SortedMap<K,V> headMap(K maxKey)
```

As above, but keys are less than `maxKey`.

```
public SortedMap<K,V> tailMap(K minKey)
```

As above, but keys are greater than or equal to `minKey`.

Changes made to the submap or to the original map are visible to the other.

HashMap, LinkedHashMap, IdentityHashMap and WeakHashMap

```
public HashMap(int initialCapacity, float loadFactor)

public HashMap(int initialCapacity)

public HashMap()

public HashMap(Map<? extends K, ? extends V> map)
```

> Creates a new HashMap whose initial mappings are copied from map. The initial capacity is based on the size of map; the default load factor is used.

- When the number of entries in the hash map exceeds the product of the load factor and the current capacity, the capacity will be doubled: all the elements be rehashed and stored in the correct new buckets.
- You need to balance the cost of normal operations against the costs of iteration and rehashing. The default load factor of 0.75 provides a good general trade-off.

- `LinkedHashMap<K,V>` extends `HashMap<K,V>` by defining an order to iterating the entries in the map. By default, return the entries in the order in which they were added.
- Additionally, `LinkedHashMap` provides a constructor that takes the initial capacity, the load factor and a boolean flag `accessOrder`: if it is `true`, then the map is sorted from the most recently accessed entry to the least recently accessed entry, making it suitable for implementing a Least Recently Used (`LRU`) cache. The only methods to count as an access of an entry are direct use of `put`, `get`, and `putAll`.
- Generally, `Map` requires that equality for keys be based on equivalence (i.e. the `equals` method). `IdentityHashMap` class uses object identity (comparison using ==).
- `WeakHashMap` refers to keys by using `WeakReference` objects instead of strong references. Weak references let the objects be collected as garbage, so you can put an object in a `WeakHashMap` without the map's reference forcing the object to stay in memory. When a key is only weakly reachable, its mapping may be removed from the map, which also drops the map's strong reference to the key's value object.

TreeMap

The TReeMap class implements SortedMap, keeping its keys sorted in the same way as TReeSet. This makes adding, removing, or finding a key/value pair $O(\log n)$. So you generally use a TreeMap only if you need the sorting or if the hashCode method of your keys is poorly written, thereby destroying the $O(1)$ behavior of HashMap.

`public treeMap()`

TreeMap sorted according to the natural order of the keys.

`public treeMap(Map<? extends K, ? extends V> map)`

Add all the key/value pairs of map to a new TreeMap().

`public TReeMap(Comparator<? super K> comp)`

Creates a new TreeMap that is sorted according to the order imposed by comp.

`public treeMap(SortedMap<K, ? extends V> map)`

Creates a new TReeMap whose initial contents will be the same as those in map and that is sorted the same way as map.

enum Collections

```
enum FieldModifiers { STATIC, FINAL, VOLATILE, TRANSIENT }
public class Field {
    public EnumSet<FieldModifiers> getModifiers() {
        // ...
    }
    // ... rest of Field methods ...
}
```

`public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)`

Creates an `EnumSet` containing all the elements of the given enum type.

`public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)`

Creates an empty `EnumSet` for the `enumType`.

`public static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> set)`

`public static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> set)`

`public static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> coll)`

Creates an `EnumSet` from the given *nonempty* collection.

`public static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)`

Creates an `EnumSet` containing all the specified enum values.

- `EnumSet` uses a bit-vector internally so it is both compact and efficient.
- The iterator returns the enum values in their natural order: the order in which the enum constants were declared.
- `EnumMap<K extends Enum<K>,V>` is internally implemented using arrays

`public EnumMap(Class<K> keyType)`

An empty `EnumMap` that can hold keys of the given enum type.

`public EnumMap(EnumMap<K, ? extends V> map)`

Copy.

`public EnumMap(Map<K, ? extends V> map)`

Creates an `EnumMap` from the given nonempty map.

The `Collections` Utilities

```
public static <T extends Object & Comparable<?  super T>> T
  min(Collection<?  extends T> coll)
```

Returns the smallest valued element of the collection based on the elements' natural order.

```
public static <T> T
 min(Collection<?  extends T> coll, Comparator<?  super T> comp)
```

Returns the smallest valued element of the collection according to `comp`.

```
public static <T extends Object & Comparable<?  super T>> T
 max(Collection<?  extends T> coll)
```

```
public static <T> T
 max(Collection<?  extends T> coll, Comparator<?  super T> comp)
```

```
public static <T> Comparator<T> reverseOrder()
```

Returns a `Comparator` that reverses the natural ordering of the objects it compares.

```
public static <T> Comparator<T> reverseOrder(Comparator<T> comp)
```

Returns a `Comparator` the reverses the order of the given comparator.

```
public static <T> boolean
  addAll(Collection<?  super T> coll, T...  elems)
```

Adds all of the specified elements to the given collection, returning `true` if the collection was changed.

```
public static boolean
  disjoint(Collection<?> coll1, Collection<?> coll2)
```

Returns `true` if the two given collections have no elements in common.

There are numerous methods for working with lists:

```
public static <T> boolean
  replaceAll(List<T> list, T oldVal, T newVal)
```

Replaces all occurrences of `oldVal` in the list with `newVal`.
Returns `true` if any replacements were made.

```
public static void reverse(List<?> list)
```

Reverses the order of the elements of the list.

```
public static void shuffle(List<?> list)
```

Randomly shuffles (permutes) the list.

```
public static int indexOfSubList(List<?> source, List<?> target)
```

Returns the index of the start of the first sublist of `source`
that is equal to `target`.

```
public static <T extends Comparable<?  super T>> void sort(List<T> list)
```

Sorts `list` in ascending order, according to its elements' natural ordering.

```
public static <T> void sort(List<T> list, Comparator<?  super T> comp)
```

Sorts `list` according to `comp`.

```
public static <T> int
 binarySearch(List<?  extends Comparable<?  super T>> list,T key)
```

```
public static <T> int
 binarySearch(List<?  extends T> list, T key, Comparator<?  super T> comp)
```

Uses a binary search algorithm to find a `key` object in the `list`, returning its index. The list must be in its elements' natural order / in the order defined by `comp`. If the object is not found, if i is the index at which the key could be inserted and maintain the order, the value returned will be $-(i+1)$.

Even more utilities:

`public static int  frequency(Collection<?> coll, Object elem)`

> Returns the number of times that the given element appears in the given collection.

`public static <T> Set<T> singleton(T elem)`

> Returns an immutable set containing only `elem`.

`public static <T> List<T> singletonList(T elem)`

> Returns an immutable list containing only `elem`.

`public static <K,V> Map<K,V>  singletonMap(K key, V value)`

> Returns an immutable map containing only one entry: a mapping from `key` to `value`.

`Collections` has static methods that return **unmodifiable wrappers**: `unmodifiableCollection`, `unmodifiableSet`, `unmodifiableSortedSet`, `unmodifiableList`, `unmodifiableMap`, and `unmodifiableSortedMap`.

Modifying methods of the unmodifiable wrapper throw `UnsupportedOperationException`. Any changes to the collection will be visible through the wrapped collection:, the contents of an unmodifiable wrapper can change, but not through the wrapper itself.

The `Arrays` Utility Class

- `sort` Sorts an array into ascending order.
- `binarySearch` Searches a sorted array for a given key. Returns the key's index, or a negative value encoding a safe insertion point.
- `fill` Fills in the array with a specified value.
- `equals` **and** `deepEquals` Return `true` if the two arrays they are passed are the same object, are both `null`, or have the same size and equivalent contents, using `Object.equals` on each non-`null` element of the array. The `deepEquals` method recursively takes into account the equivalence of nested arrays.
- `hashCode` **and** `deepHashCode` Return a hash code based on the contents of the given array (`deepHashCode` recursively looks into nested arrays).
- `toString` **and** `deepToString` Return a string representation of the contents of the array: a comma seperated list of the array's contents, enclosed by '`[`' and '`]`'. The array contents are converted to strings with `String.valueOf`.

- You can view an array of objects as a `List` by using the object returned by the `asList` method. The result is backed by the passed array, it is modifiable but not resizable.

```
public static <T> List<T> asList(T... elems)
```

Writing Iterator Implementations

```
public class ShortStrings implements Iterator<String> {
    private Iterator<String> strings;  // source for strings
    private String nextShort;      // null if next not known
    private final int maxLen;      // only return strings <=
    public ShortStrings(Iterator<String> strings, int maxLen) {
        this.strings = strings; this.maxLen = maxLen; nextShort = null; }
    public boolean hasNext() {
        if (nextShort != null)  // found it already
            return true;
        while (strings.hasNext()) {
            nextShort = strings.next();
            if (nextShort.length() <= maxLen) return true;
        }
        nextShort = null; return false;   // didn't find one
    }
    public String next() {
        if (nextShort == null && !hasNext())
            throw new NoSuchElementException();
        String n = nextShort;   // remember nextShort
        nextShort = null;       // consume nextShort
        return n;               // return nextShort
    }
    public void remove() { throw new UnsupportedOperationException(); }
}
```

- `hasNext` will work if invoked multiple times before a `next`.
- `next` works even if programmer using it has never invoked `hasNext`.
- `remove` is not allowed because it cannot work correctly. If `remove` invoked `remove` on the underlying iterator, the following legal (although odd) code can cause incorrect behavior:

```
it.next();
it.hasNext();
it.remove();
```

  You cannot build a filtering iterator on top of another `Iterator` object. You can build one on top of a `ListIterator`: it allows to back up to the previously returned short string.

Writing Collection Implementations

- `AbstractCollection`, `AbstractSet`, `AbstractList`, `AbstractSequentialList`, `AbstractQueue`, and `AbstractMap` contain skeletal implementations on which the `java.util` collections are based.
- `AbstractCollection`
  - for an unmodifiable collection provide `iterator()` and `size()`
  - for modifiable, also override `add` (and the iterator supporting `remove`)
- `AbstractList` – for random access "backing store" like an array
  - for an unmodifiable list provide `get(int)` and `size()`
  - for modifiable, also override `set(int, E)` and (for variable size) `add(int, E)`, `remove(int)`
- `AbstractSequentialList` – for linear access like a linked list
  - provide the `listIterator(int)` method

```
public class ArrayBunchList<E> extends AbstractList<E> {
    private final E[][] arrays;
    private final int size;
    public ArrayBunchList(E[][] arrays) {
        this.arrays = arrays.clone();
        int s = 0;
        for (E[] array : arrays)
            s += array.length;
        size = s;
    }
    public int size() {
        return size;
    }
    public E get(int index) {
        int off = 0;    // offset from start of collection
        for (int i = 0; i < arrays.length; i++) {
            if (index < off + arrays[i].length)
                return arrays[i][index - off];
            off += arrays[i].length;
        }
        throw new ArrayIndexOutOfBoundsException(index);
    }
```

```
    public E set(int index, E value) {
        int off = 0;    // offset from start of collection
        for (int i = 0; i < arrays.length; i++) {
            if (index < off + arrays[i].length) {
                E ret = arrays[i][index - off];
                arrays[i][index - off] = value;
                return ret;
            }
            off += arrays[i].length;
        }
        throw new ArrayIndexOutOfBoundsException(index);
    }
}
```

- `ArrayBunchList` implements modifiable but not resizable collection
- `AbstractList` provides `Iterator` and `ListIterator`, but based on `get`, which is inefficient in the above implementation. Below is an optimized `Iterator` for `ArrayBunchList`.

```
private class ABLIterator implements Iterator<E> {
    private int off;        // offset from start of list
    private int array;      // array we are currently in
    private int pos;        // position in current array

    ABLIterator() {
        off = 0;
        array = 0;
        pos = 0;
        // skip any initial empty arrays (or to end)
        for (array = 0; array < arrays.length; array++)
            if (arrays[array].length > 0)
                break;
    }

    public boolean hasNext() {
        return off + pos < size();
    }
```

```
    public E next() {
        if (!hasNext())
            throw new NoSuchElementException();
        E ret = arrays[array][pos++];

        // advance to the next element (or to end)
        while (pos >= arrays[array].length) {
            off += arrays[array++].length;
            pos = 0;
            if (array >= arrays.length)
                break;
        }
        return ret;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```