Programming in Java – Part 03 – Introduction 2/2

Paolo Vercesi

ESTECO SpA

Agenda

- Collections
- Primitive type wrappers
- Enumerations
- Generics
- Exceptions

Collections

Collections Overview

The collection framework standardize the way in which group of objects are handled

The framework is high-performance, and it must be used by every Java programmer, no need to reinvent the wheel here

The API is consistent through the different classes, and it provides a high degree of interoperability

Maps are parts of the Collections framework, even if they are not collections in a strict sense

All classes and interfaces in the collection framework are generic

All the classes resides in the `java.util` package

Collections

Lists

- Used for access by index
- Elements are always ordered (never sorted)

Sets

- Used for queries about presence
- Don’t contain duplicated elements
- Elements can be unordered, ordered or sorted, depending on the implementation

Maps

- Used for access by key
- Elements can be unordered, ordered, or sorted , depending on the implementation

Queues

- Good for FIFO/LIFO access

Ordered/Sequenced vs sorted

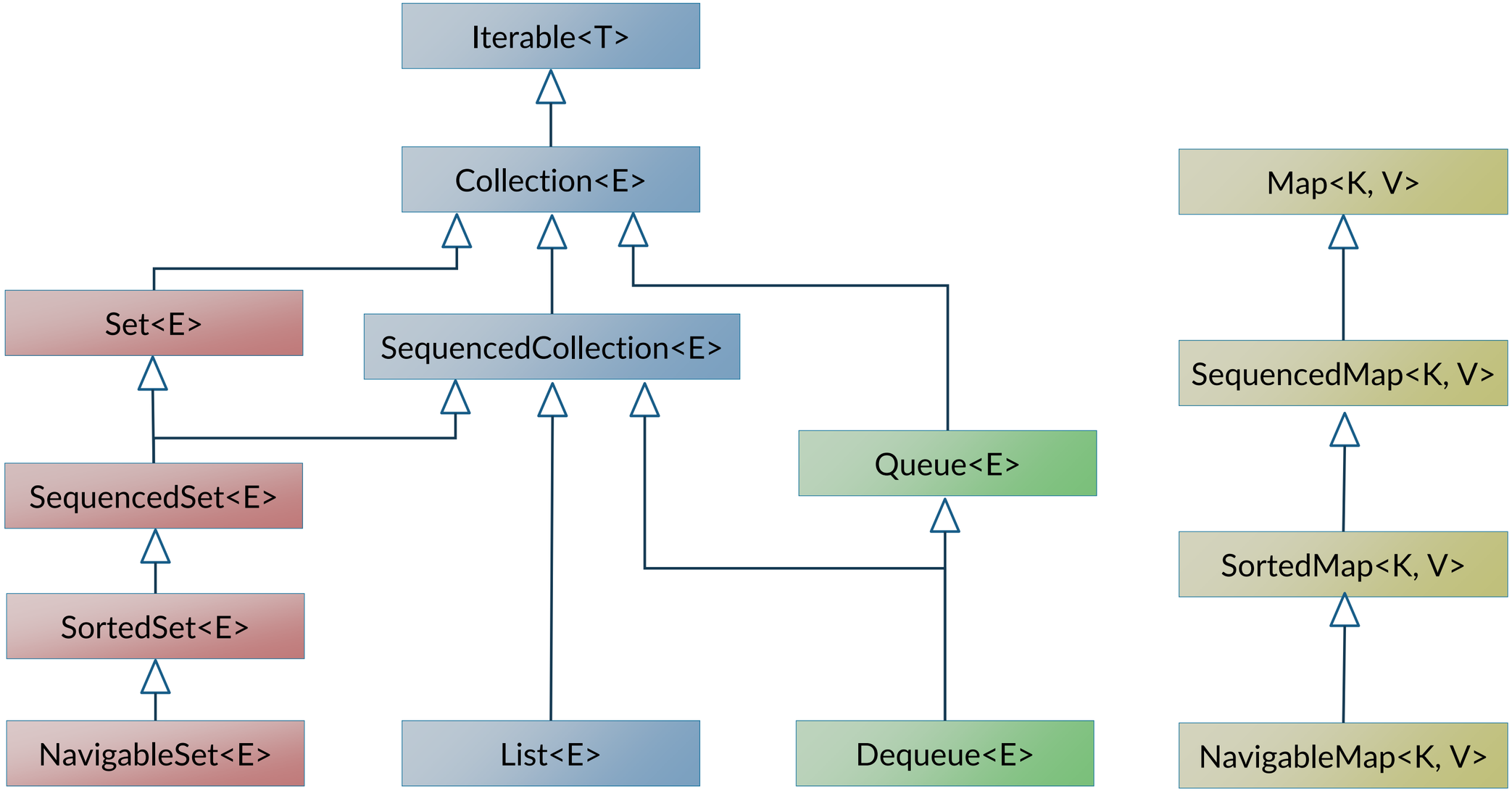
Ordered/Sequenced

- Elements are arranged following some order
- Insertion order
  - The first inserted element is in the first position
  - The last inserted element is in the last position
- Imposed order
  - Put this element in the 5^{th} position

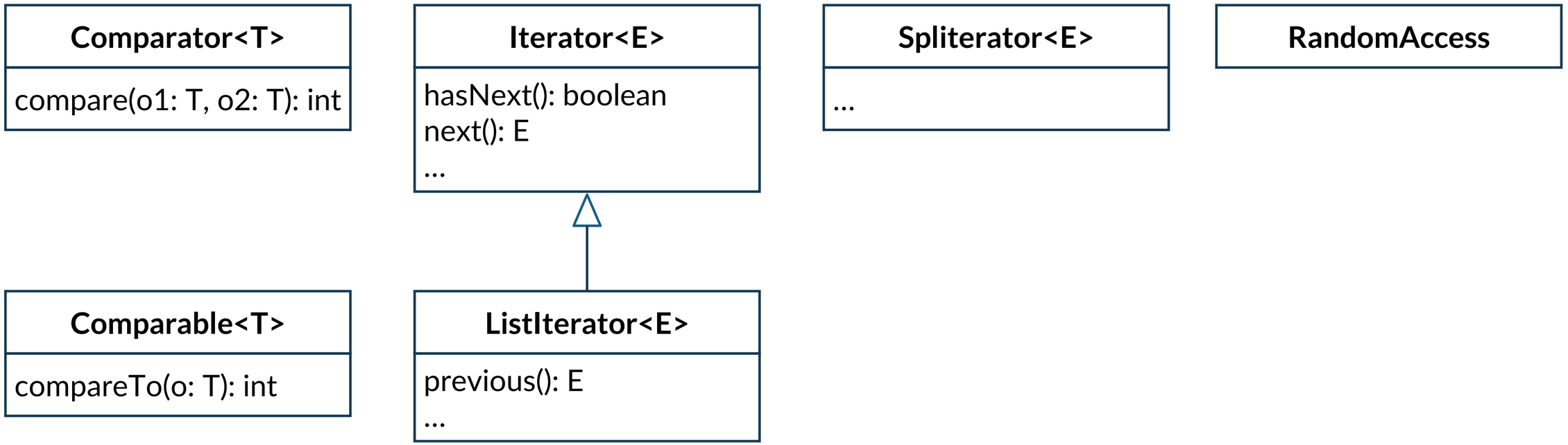
Sorted

- Elements are comparable
  - 1 < 10
  - “Dario” precedes “Paolo” lexicographically
- The collection keeps the elements sorted
- What if you alter an element?

The Collection interfaces



Other interfaces



Querying a collection

```
boolean isEmpty()
int size()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
Iterator<E> iterator()
default Spliterator<E> spliterator()
default Stream<E> stream()
default Stream<E> parallelStream()
Object[] toArray()
default <T> T[] toArray(IntFunction<T[]> generator)
<T> T[] toArray(T[] a)
```

Modifying a collection

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void clear()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
default boolean removeIf(Predicate<? super E> filter)
```

All these methods are optional, and they throw UnsupportedOperationException if the collection is unmodifiable.

The Iterable interface

Iterable objects can be used in for-each loops

```
Iterable<Integer> iterable = List.of(1, 2, 3, 4, 5);

for (Integer integer : iterable) {
    System.out.println(integer);
}
```

The Comparable interface

```
public interface Comparable<T> {

    public int compareTo(T o);
}
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object

The implementor must ensure signum(x.compareTo(y)) == -signum(y.compareTo(x)) for all x and y

The implementor must also ensure that the relation is transitive:
(x.compareTo(y) > 0 && y.compareTo(z) > 0) implies x.compareTo(z) > 0

Finally, the implementor must ensure that
x.compareTo(y)==0 implies that signum(x.compareTo(z)) == signum(y.compareTo(z)), for all z

The Comparator interface

```
public interface Comparator<T> {

    int compare(T o1, T o2);
}
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

How to remember? If they are integer, it is the result of `o1-o2`.

The implementor must ensure that signum(compare(x, y)) == -signum(compare(y, x)) for all x and y.

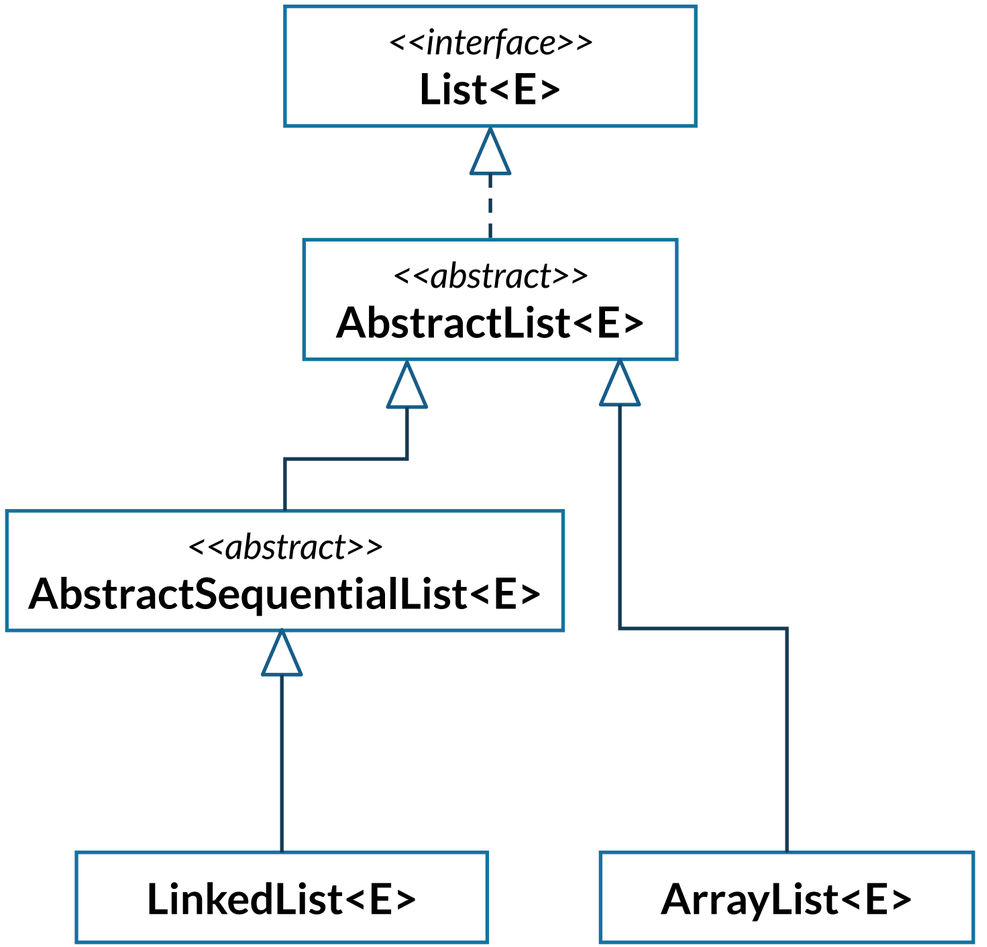
The implementor must also ensure that the relation is transitive:
((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

Finally, the implementor must ensure that
compare(x, y)==0 implies that signum(compare(x, z))==signum(compare(y, z)) for all z.

Lists

Lists

```
public interface List<E> extends Collection<E> {
    …
}
```



The List interface

The List interface extends the Collection interface with list specific methods

```
E get(int index)
E set(int index, E element)
E remove(int index)
boolean addAll(int index, Collection<? extends E> c)
default void replaceAll(UnaryOperator<E> operator)
default void sort(Comparator<? super E> c)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
List<E> subList(int fromIndex, int toIndex) (returns a view)
```

Creating unmodifiable lists

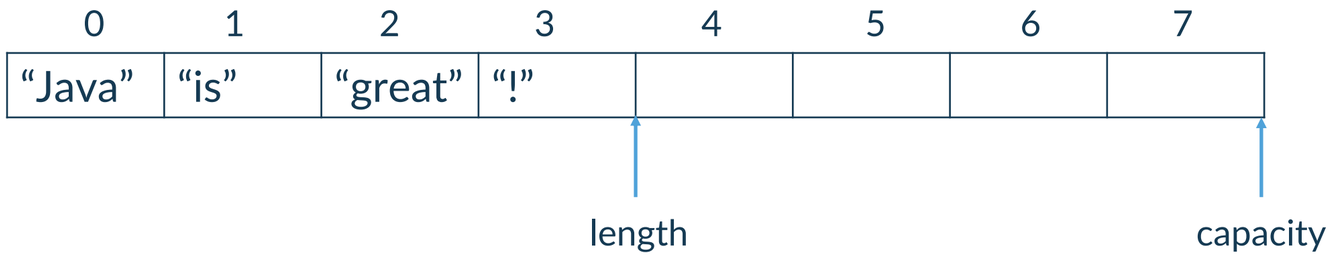
The List interface provides methods to create unmodifiable lists

```
static <E> List<E> copyOf(Collection<? extends E> coll)
static <E> List<E> of()
static <E> List<E> of(E e1)
…
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
```

The ArrayList class

Implements List and RandomAccess

Internally based on arrays

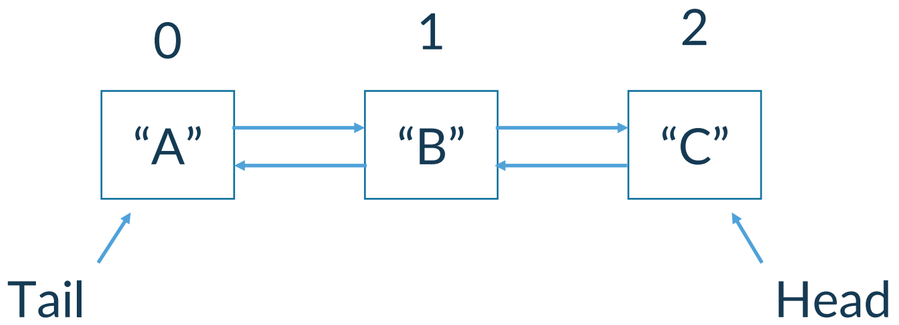



```
ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
void ensureCapacity(int cap)
void trimToSize( )
```

The LinkedList class

Implements List, Queue and Deque

Double linked list



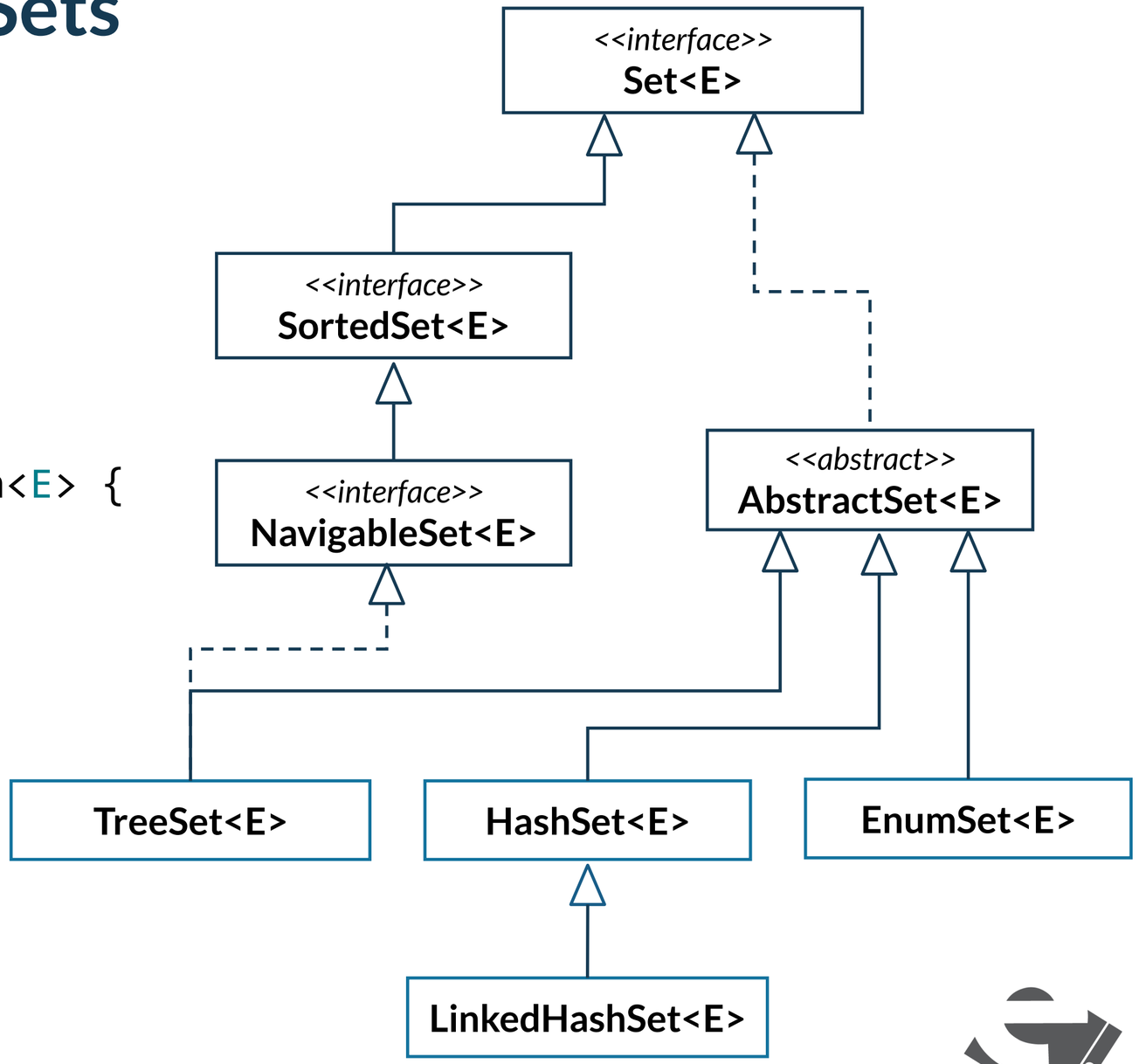

```
LinkedList( )
LinkedList(Collection<? extends E> c)
```

Sets

Sets

The set interface specifies the behavior of a collection that does not allow duplicate elements. The Set interface extends Collection, and it does not specify any additional instance method of its own

```
public interface Set<E> extends Collection<E> {
    …
}
```



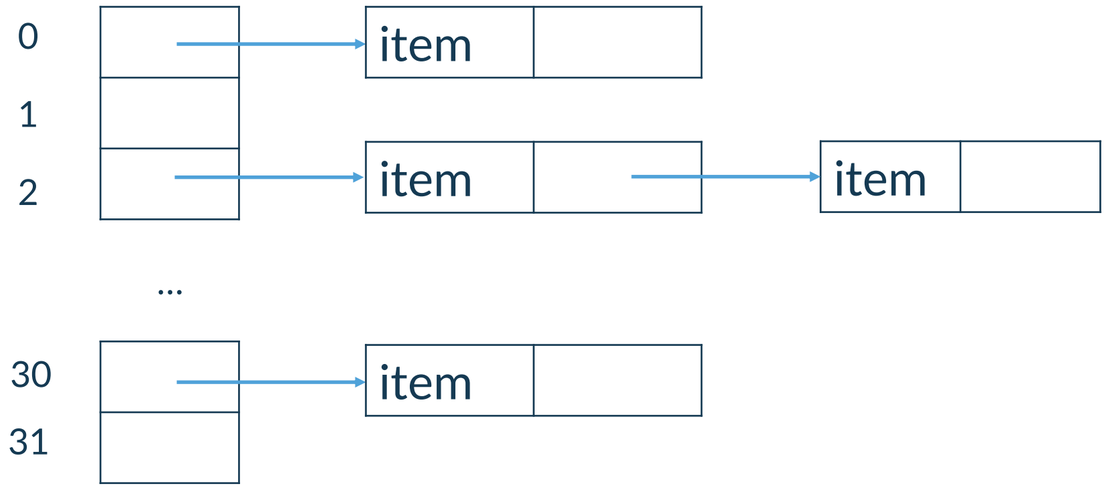
Creating unmodifiable sets

```
static <E> Set<E> of()
static <E> Set<E> of(E e1)
static <E> Set<E> of(E e1, E e2)
…
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
static <E> Set<E> of(E... elements)
static <E> Set<E> copyOf(Collection<? extends E> coll)
```

The HashSet class

Set interface, backed by a hash table, it does not define any additional methods beyond those provided by its super classes and interfaces.

```
HashSet()
HashSet(Collection<? extends E> c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)
```



HashSet has no predictable iteration order. LinkedHashSet allows insertion-order iteration over the map

The SortedSet interface

```
public interface SortedSet<E> extends Set<E> {
    E first()
    E last()
    SortedSet<E> headSet(E toElement)
    SortedSet<E> subSet(E fromElement, E toElement)
    SortedSet<E> tailSet(E fromElement)
    Comparator<? super E> comparator()
}
```

The iterator return the elements using the specified order.

The TreeSet class

The TreeSet class implements the NavigableSet and SortedSet interfaces, it based on a Red-Black tree implementation.

```
TreeSet()

TreeSet(Collection<? extends E> c)

TreeSet(Comparator<? super E> comp)

TreeSet(SortedSet<E> ss)
```

Examples with collections

Iterator

Every collection can return us an Iterator that enables us to cycle through the collection, obtaining or removing elements.

```
var set = Set.of(1, 2, 3, 4, 5);
for (Iterator<Integer> it = set.iterator(); it.hasNext();) {
    Integer item = it.next();
    if (Integer.valueOf(3).equals(item)) {
        it.remove();
    }
}
```

Lists can return a ListIterator that enables cycling through the collection in both directions

ConcurrentModificationException

```
var set = new HashSet<Integer>();
set.add(1);
set.add(2);
set.add(3);
set.add(4);
set.add(5);
for (Integer item : set) {
    if (Integer.valueOf(3).equals(item)) {
        set.remove(item);
    }
}
```

Altering a collection while cycling through the elements can cause a ConcurrentModificationException

Comparable

The Comparable interface is implemented by classes that want to define a “natural” order.

The String class implements Comparable, so String objects can be compared one with each other

```
public static void main(String[] args) {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    TreeSet<String> naturalOrder = new TreeSet<>(citiesOfFvg);

    System.out.println(naturalOrder);
}
```

```
[Gorizia, Pordenone, Trieste, Udine]
```

The result is the list of the cities
in the String lexicographical order

Comparator 1/3

Implementing the Comparator interface we can define an order for classes that doesn't implement Comparable, or we can override the "natural" order of classes that already implements Comparable

```
public static void main(String[] args) {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    TreeSet<String> lengthOrder = new TreeSet<>(new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() - o2.length();
        }
    });
    lengthOrder.addAll(citiesOfFvg);

    System.out.println(lengthOrder);
}
```

```
[Udine, Trieste, Pordenone]
```

The result is the list of the cities in order of name length

What about Gorizia?

Comparator 2/3

```
Comparator<String> comparator = new Comparator<>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
};
TreeSet<String> triesteOnly = new TreeSet<>(comparator);
triesteOnly.add("Trieste");

System.out.println(triesteOnly.contains("Gorizia"));
```

The answer is true because the order relation defined by the comparator is used as equivalence relation.

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be consistent with equals if it is to correctly implement the Set interface.
(See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal. The behavior of a set is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.

Comparator 3/3

```
public static void main(String[] args) {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    Comparator<String> comparator = new Comparator<>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() == o2.length() ? o1.compareTo(o2) : o1.length() - o2.length();
        }
    };

    TreeSet<String> lengthOrder = new TreeSet<>(comparator);
    lengthOrder.addAll(citiesOfFvg);

    System.out.println(lengthOrder);
}
```

```
[Udine, Gorizia, Trieste, Pordenone]
```

Gorizia is back!

Object.equals()

```
public boolean equals(Object obj)
```

Indicates whether some other object is “equal to” this one

The equals method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value x, x.equals(x) should return true
- It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
- It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified
- For any non-null reference value x, x.equals(null) should return false

An equivalence relation partitions the elements it operates on into *equivalence classes*; all the members of an equivalence class are equal to each other. Members of an equivalence class are substitutable for each other, at least for some purposes

The default implementation is the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true). In other words, under the reference equality equivalence relation, each equivalence class only has a single element

Equals and ordering

The implementations of the methods of the Comparable and Comparator interfaces must be consistent with equals

The “natural” ordering for a class C (implementing Comparable) is said to be *consistent with equals* if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C.

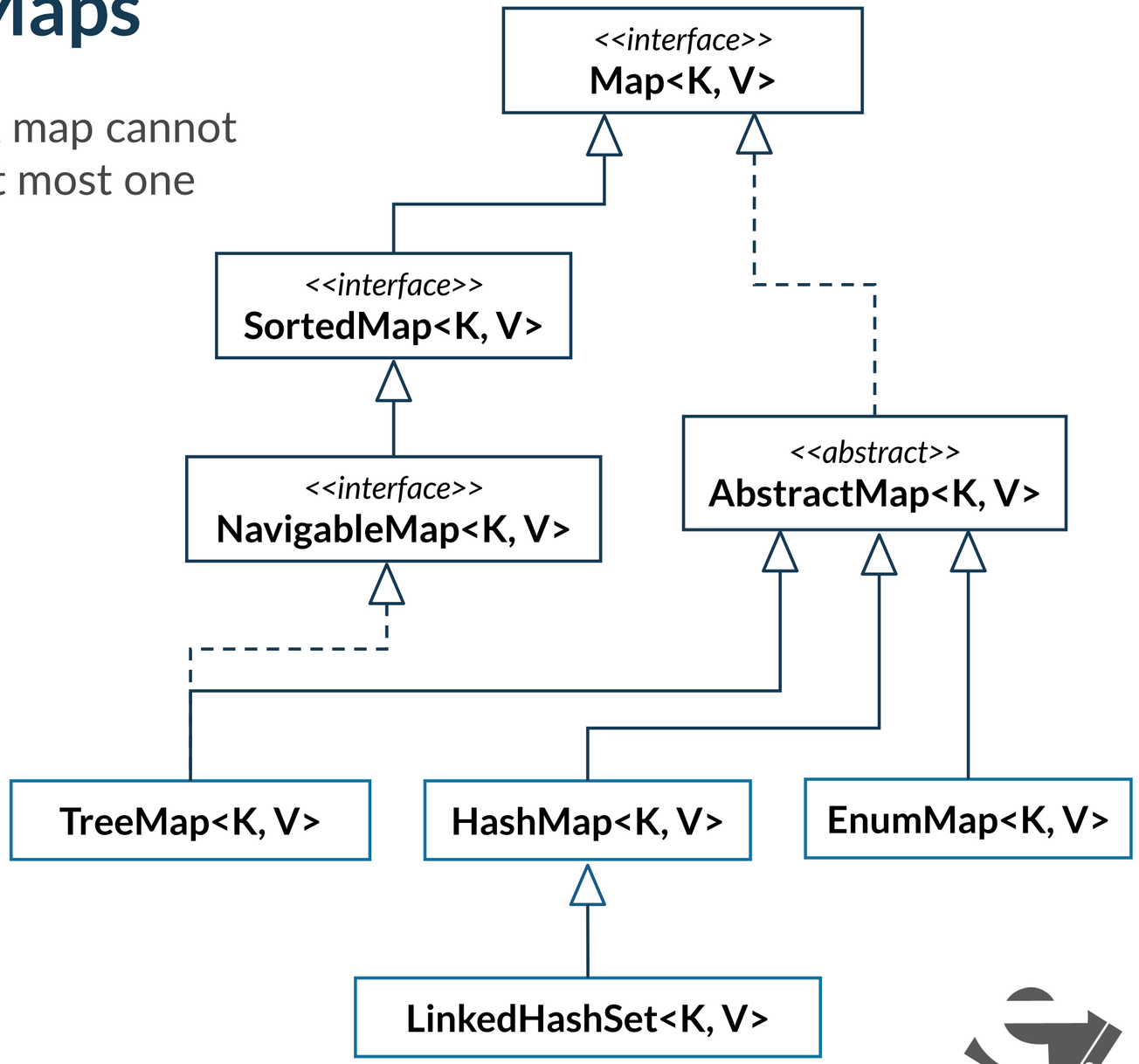
Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

Maps

Maps

A map is an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

```
public interface Map<K, V> {
    …
}
```



Querying a map

```
V get(Object key)
boolean containsKey(Object key)
boolean containsValue(Object value)
default V getOrDefault(Object key, V defaultValue)
boolean isEmpty()
int size()
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K,V>> entrySet()
```

Modifying a map

```
V put(K key, V value)
V remove(Object key)
default V replace(K key, V value)
void putAll(Map<? extends K,? extends V> m)
default V putIfAbsent(K key, V value)
default boolean replace(K key, V oldValue, V newValue)
default boolean remove(Object key, Object value)
boolean isEmpty()
void clear()
```

All these methods are optional, and they throw UnsupportedOperationException if the collection is unmodifiable.

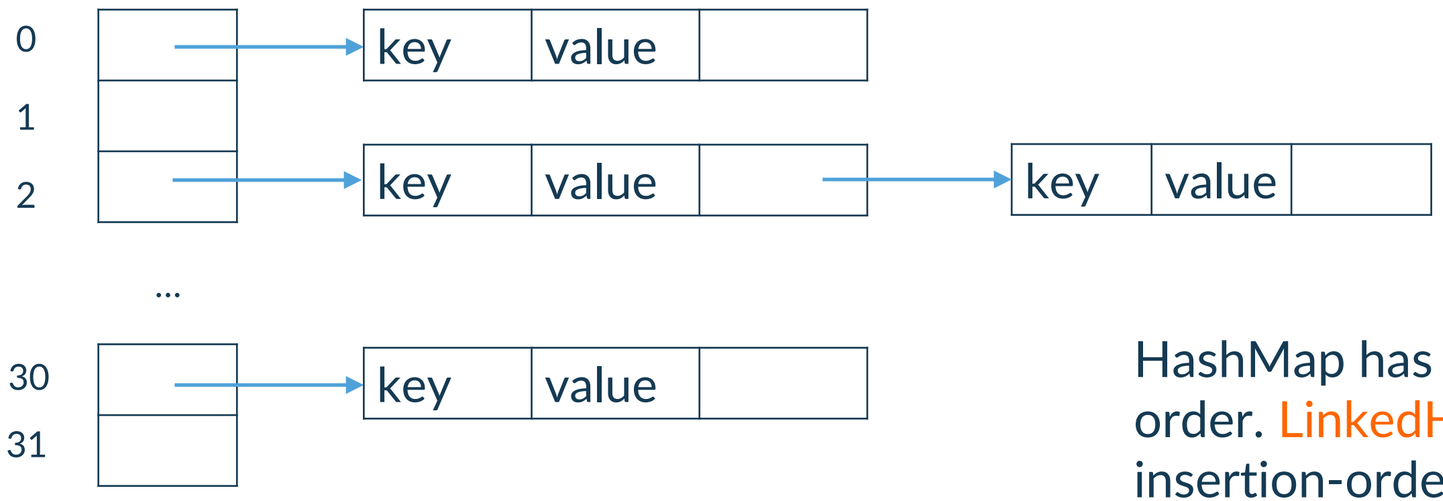
Creating unmodifiable maps

```
static <K,V> Map<K,V> copyOf(Map<? extends K,? extends V> map)
static <K,V> Map.Entry<K,V> entry(K k, V v)
static <K,V> Map<K,V> of()
static <K,V> Map<K,V> of(K k1, V v1)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2)
…
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, … K k10, V v10)
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)
```

HashMap

Implements the Map interface, backed by a hash table, it does not define any additional methods beyond those provided by its super classes and interfaces.

```
HashMap()

HashMap(Collection<? extends E> c)

HashMap(int capacity)

HashMap(int capacity, float fillRatio)
```



HashMap has no predictable iteration order. LinkedHashMap allows insertion-order iteration over the map

Object.hashCode()

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application
- If two objects are equal according to the equals method, then calling the hashCode method on each of the two objects must produce the same integer result
- It is *not* required that if two objects are unequal according to the equals method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables

HashCode and equals 1/4

Suppose we want to define a File class to represent file systems objects

And we want to store the file sizes into a Map, so that we can retrieve that information from the map

```
public class File {

    private String name;

    public File(String name) {
        this.name = name;
    }
}
```

```
public static void main(String[] args) {
    var fileSizes = new HashMap<File, Long>();
    fileSizes.put(new File("C:\\Users\\pvercesi\\lesson8.pdf"), 342340L);
    fileSizes.put(new File("C:\\Users\\pvercesi\\lesson9.pdf"), 512956L);

    File file = new File("C:\\Users\\pvercesi\\lesson8.pdf");
    System.out.println(fileSizes.get(file));
}
```

This example doesn't work as expected

Why?

HashCode and equals 2/4

```
public class File {
    private String name;

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

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        File file = (File) o;
        return Objects.equals(name, file.name);
    }

    @Override
    public int hashCode() {
        return name.hashCode();
    }
}
```

HashCode and equals 3/4

Now we want
to rename a File

```
public class File {

    private String name;

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

    public void rename(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    …
}
```

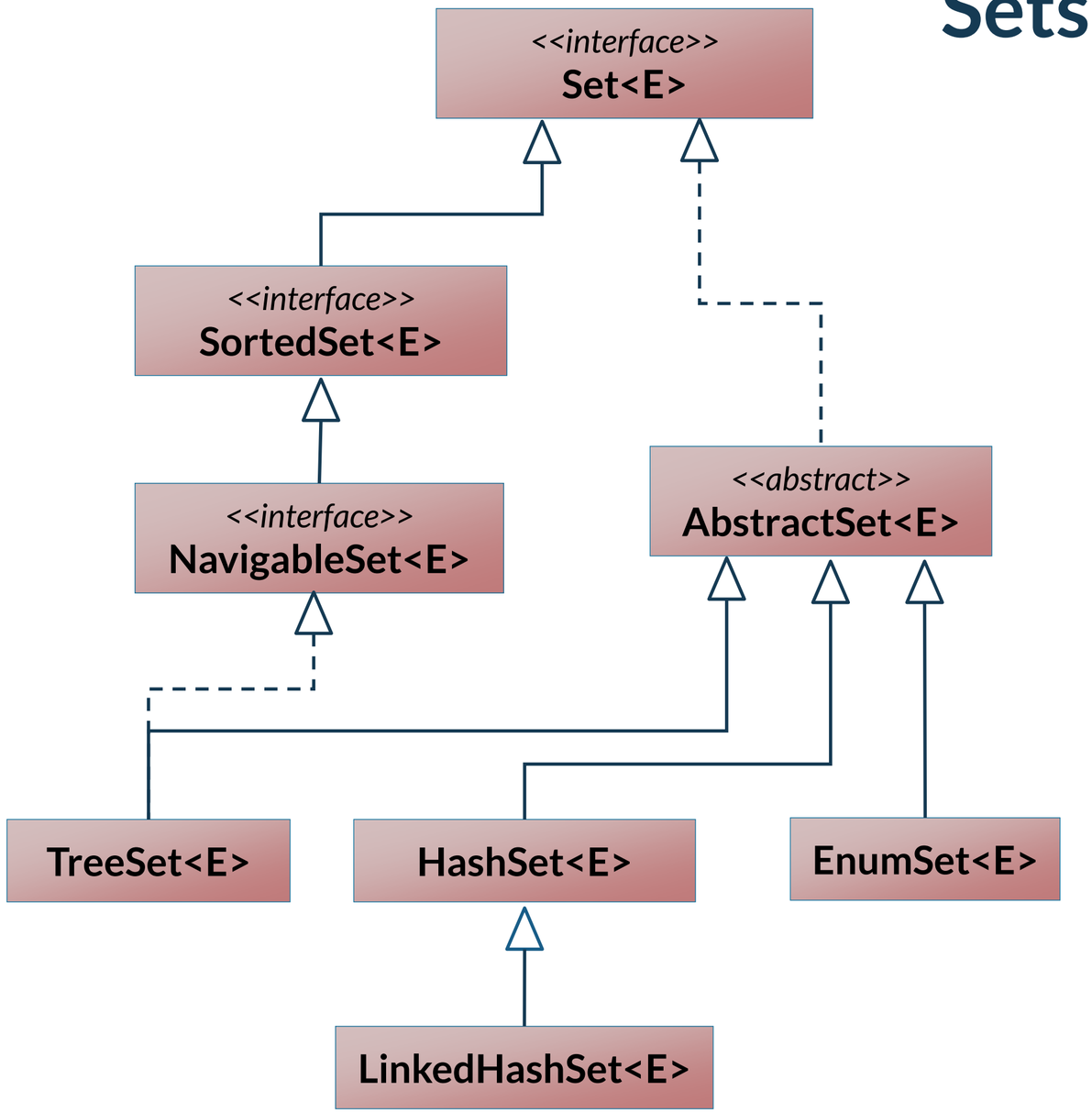
HashCode and equals 4/4

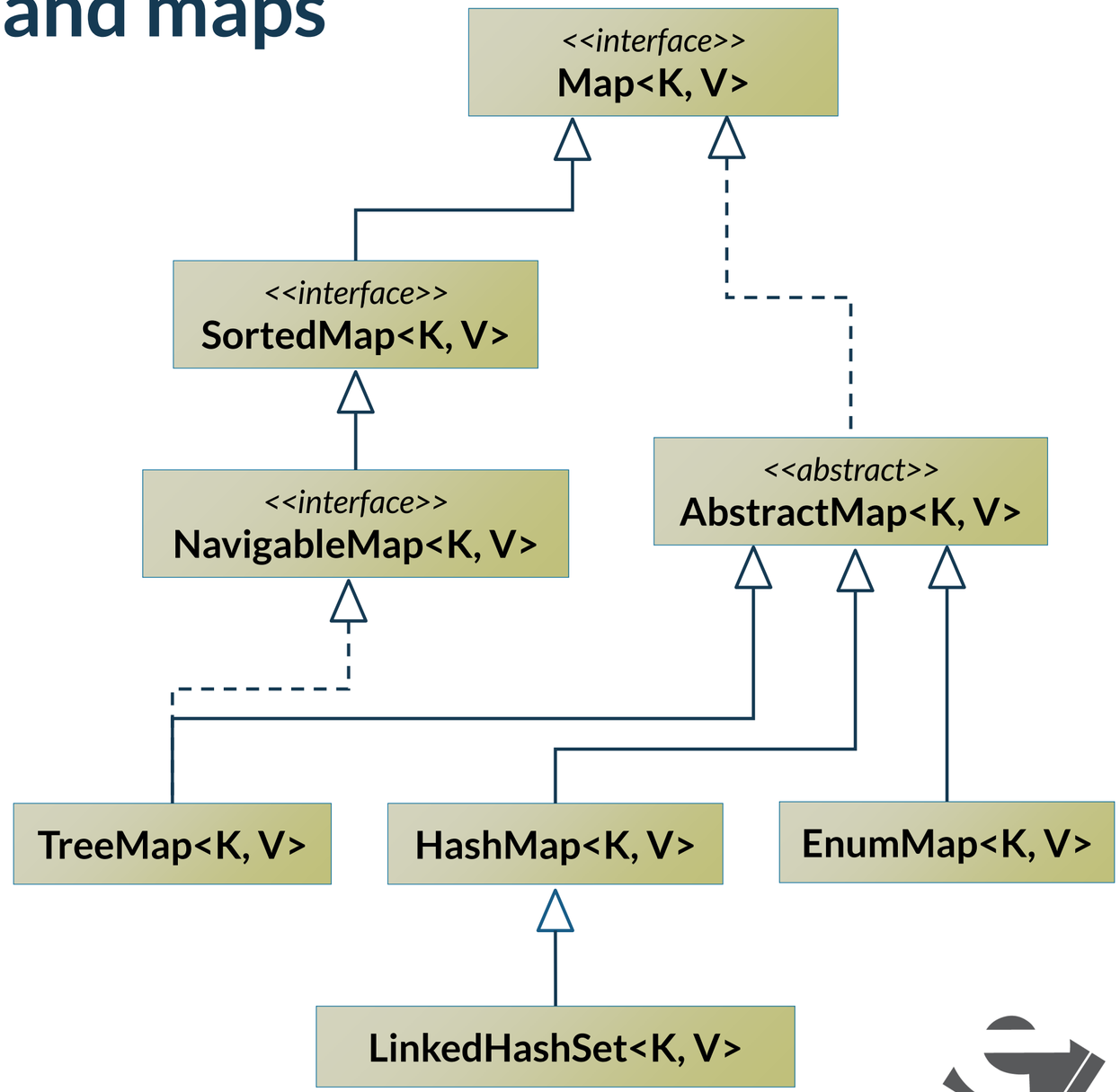
```
public static void main(String[] args) {
    var fileSizes = new HashMap<File, Long>();
    fileSizes.put(new File("C:\\Users\\pvercesi\\Desktop\\lesson8.pdf"), 342340L);
    fileSizes.put(new File("C:\\Users\\pvercesi\\Desktop\\lesson9.pdf"), 512956L);

    String fromName = "C:\\Users\\pvercesi\\Desktop\\lesson8.pdf";
    String toName = "C:\\Users\\pvercesi\\Desktop\\lesson8a.pdf";

    for (File file : fileSizes.keySet()) {
        if (file.getName().equals(fromName)) {
            file.rename(toName);
        }
        break;
    }

    File file = new File("C:\\Users\\pvercesi\\Desktop\\lesson8a.pdf");
    System.out.println(fileSizes.get(file));
}
```

After renaming, the hash code is changed but the object position in the hash table is not updated!
The API of the file object is flawed, we cannot use this class as key in hash-based collections

Fortunately, the java.io.File class of the Java API, doesn't work in this way ☺

Sets and maps





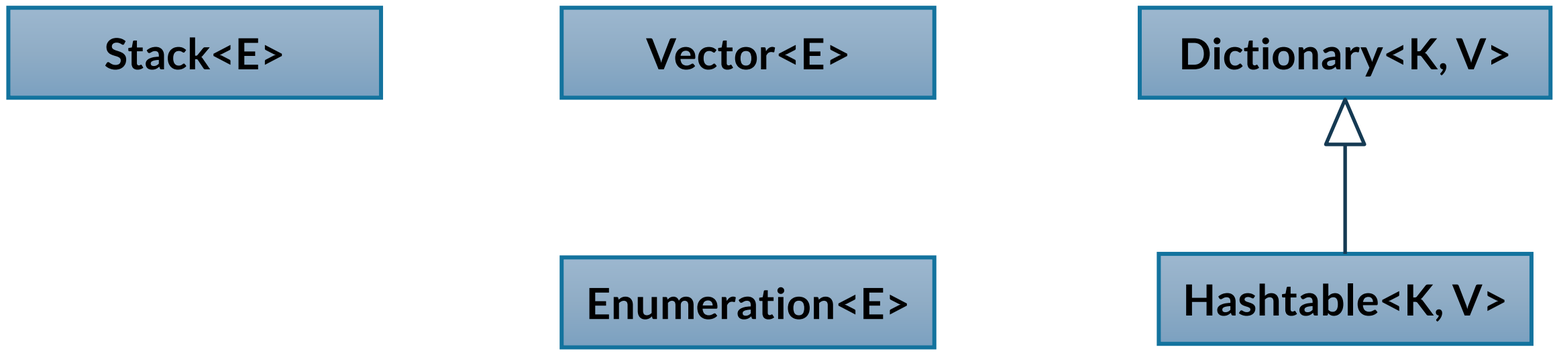
Summary

The Collections class

The java.util.Collections class provide several methods to work with collections

- factory methods to get empty collections
- factory methods to build unmodifiable collections
- factory methods for synchronized collections
- search in ordered lists
- copy, sort, rotate, shuffle, and fill lists
- count and replace occurrences
- find min and max
- ..

Legacy classes



They are part of the old Java API, you should not use them. If you are using an API that uses these classes, maybe there is more a modern implementation.

Summary of implementations

	List	Queue	Set	Map
Array based	**ArrayList**	**ArrayDeque**		
Linked	**LinkedList**		**LinkedHashSet**	**LinkedHashMap**
Hash based			**HashSet**	**HashMap**
Tree based			**TreeSet**	**TreeMap**
Bit-set based			**EnumSet**	**EnumMap**
Priority queue		**PriorityQueue**		

Primitive type wrappers

Primitive types

- `byte`
- `short`
- `int`
- `long`
- `character`
- `float`
- `double`
- `boolean`
- `void`

✓ Primitive types are not objects!

✓ Primitive types are used for performance reasons, however many situations require an object

✓ Most Java classes have methods that works on Objects and not on primitive type.

Primitive types are not objects

```
public interface List {

    boolean isEmpty();

    int getSize();

    boolean contains(Object value);

    Object[] getValues();

    Object get(int index);

    void add(Object value);

    void insertAt(int index, Object value);

    void remove(int index);

    int indexOf(Object value);
}
```

- ✓ Can I use primitive types with classes implementing this List interface?
- ✓ The same interface and consequently the same implementation cannot be used for primitive types

A List interface for the int type

```
public interface IntList {

    boolean isEmpty();

    int getSize();

    boolean contains(int value);

    int[] getValues();

    int get(int index);

    void add(int value);

    void insertAt(int index, int value);

    void remove(int index);

    int indexOf(int value);
}
```

- ✓ We would need one interface for the int type, one for the long type, one for the short type, and so forth
- ✓ In some cases, Java uses this approach. Why?
- ✓ Only for performance reasons. More on this when we will talk about Streams.

Primitive type wrappers

- `byte`
- `short`
- `int`
- `long`
- `character`
- `float`
- `double`
- `boolean`

- `Byte`
- `Short`
- `Integer`
- `Long`
- `Character`
- `Float`
- `Double`
- `Boolean`

✓ There exist one wrapper class for each primitive type

✓ Wrappers are classes that wrap primitive types within an object

From primitive value to wrapper object

The static factory method valueOf() is the recommended way to convert a primitive value to an object

```
int i = 60;
Integer i1 = Integer.valueOf(i);
Integer i2 = new Integer(i);
```

The use of primitive type wrapper's constructors is deprecated.

Why?

Hint: consider the boolean case

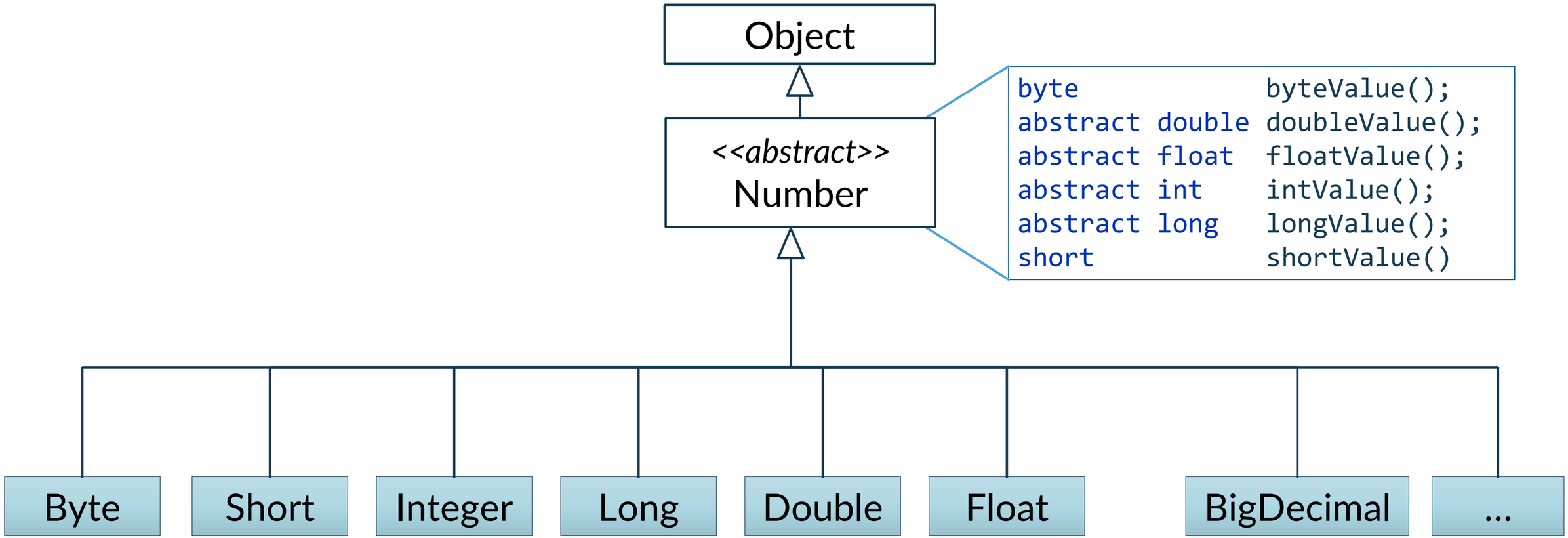
```
boolean b = true;
Boolean b1 = Boolean.valueOf(b);
Boolean b2 = new Boolean(b);
```

From wrapper object to primitive value

```
Boolean b = Boolean.FALSE;
boolean b1 = b.booleanValue();

Character c = Character.valueOf('a');
char c1 = c.charValue();
```

What about numerical values?

The Number hierarchy



Boxing and unboxing

Also known as auto-boxing and auto-unboxing

```
Boolean b = false;
boolean b1 = b;

Character c = 'a';
char c1 = c;

int i = 60;
Integer i1 = i;
Double d1 = i;
Double d1 = i1.doubleValue();
int i2 = i + i1;
```

Boxing is the process by which a primitive type is automatically wrapped into its equivalent type wrapper whenever an object of that type is needed.

There is no need to explicitly construct such an object.

Unboxing is the process by which the value of a boxed object is automatically extracted from a type wrapper when its value is needed.

There is no need to call a method such as `intValue()` or `doubleValue()`.

Unboxing can lead to NullPointerExceptions

Assignment

Explore the API of the primitive type wrappers.
If you haven’t yet done so.

Enumerations

Enumerations

```
enum Degree {
    HIGH_SCHOOL, BACHELOR, MASTER, PHD
}
```

Each enumeration constant is a public static final member of the Degree class

An enumeration declaration
1. is a list of named constants
2. that define a new data type
3. and its legal values.

Once it is declared, an enumeration cannot be changed at runtime.

```
Degree d1 = Degree.PHD;
Degree d2 = Degree.BACHELOR;

if (d1 == d2) {
    System.out.println("This seems a bit unusual!");
}
```

You don’t instantiate an enumeration, but you can reference its members

Enumerations in switch statements

In switch statements the enumeration constants don't need to be qualified by their enumeration type name

```
Degree d = getDegree();

switch (d) {
    case HIGH_SCHOOL -> System.out.println("High School");
    case BACHELOR -> System.out.println("Bachelor");
    case MASTER -> System.out.println("Master");
    case PHD -> System.out.println("PhD");
}
```

Arrow notation

No need for break statements

Enumerate enumerations

Enumerations automatically get two static methods, one to enumerate the constants and one to get a constant from its name

```
public static enum-type [ ] values( )
public static enum-type valueOf(String str )
```

Each enumeration constant has an ordinal value

```
final int ordinal( )
```

```
Degree d1 = Degree.PHD;
Degree d2 = Degree.valueOf("PHD");

if (d1 == d2) {
    System.out.println("This looks ok!");
}

for (Degree dd : Degree.values()) {
    System.out.println(dd.ordinal(dd) + " " + dd);
}
```

Enumerations are first class classes

```
enum Degree {
    HIGH_SCHOOL(5), BACHELOR(3), MASTER(2), PHD(3);

    private final int duration;

    Degree(int duration) {
        this.duration = duration;
    }

    public int getDuration() {
        return duration;
    }
}
```

They can have fields

They can have constructors

They can have methods

The EnumSet class

A specialized Set implementation for use with enum types. All the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created.

```
public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)
public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)
public static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)
public static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)
public static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)
public static <E extends Enum<E>> EnumSet<E> of(E e)
public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2)
public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)
public static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)
public static <E extends Enum<E>> EnumSet<E> range(E from, E to)
```

Generics

Generalized classes

```
public class GeneralizedStack {

      public int getSize();

      public Object top() {…}

      public Object pop() {…}

      public void push(Object value) {…}

}
```

Before the introduction of generics in Java 5, generalized classes, interfaces and methods operated with references to Object instances, with consequent problems of type safety

```
public static void main(String[] args) {

    GeneralizedStack stack = new GeneralizedStack();

    stack.push("Hello,");
    stack.push("World!");
    stack.push(new Object());

    while (stack.getSize() > 0) {
        String text = (String) stack.pop(); Runtime exception
        System.out.println(text);
    }
}
```

We had to rely on inherently unsafe casting

Specialized classes

```
public class StringStack {

    private final GeneralizedStack data =
            new GeneralizedStack();

    public int getSize() {
        return data.getSize();
    }

    public String top() {
        return (String) data.top();
    }

    public String pop() {
        return (String) data.pop();
    }

    public void push(String value) {
        data.push(value);
    }
}
```

```
public static void main(String[] args) {

    StringStack stack = new StringStack();

    stack.push("Hello,");
    stack.push("World!");
    stack.push(new Object());

    while (stack.getSize() > 0) {
        String text = stack.pop();
        System.out.println(text);
    }
}
```

Compile time error

No need for class casting, but specialized classes required boilerplate code and a considerable use of cast operations.

What are Generics?

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}
}
```

```
public static void main(String[] args) {

    Stack<String> stringStack = new Stack<>();

    stringStack.push("Hello,");
    stringStack.push("World!");
    stringStack.push(new Object());

    while (stringStack.getSize() > 0) {
        String text = stringStack.pop();
        System.out.println(text);
    }
}
```

Java 5 introduced the concept of parameterization of interfaces, classes and methods. A parameterized interface or class is called parameterized type or generic

In generic classes, generic interfaces, and generic methods the type of data upon which they operate is specified as a parameter

Parameterized types are used to improve type safety when compared with the use of Object and they are used to reduce boilerplate code

Compile time error

Parameters bounding

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}
}
```

The parameter T can be replaced by any class type.

```
public class NumberStack<N extends Number> extends Stack<N> {

    double average() {
        double sum = 0;
        for (Number number : data) {
            sum += number.doubleValue();
        }
        return sum / getSize();
    }
}
```

N is a bounded parameter; N can be replaced only by the superclass Number or by a subclass of the superclass Number.

Wildcard arguments 1/3

```
public class Stack<T> {

  public int getSize() {…}

  public void push(T value) {…}

  public T top() {…}

  public T pop() {…}

  public boolean sameSize(Stack<T> other) {
    return getSize() == other.getSize();
  }
}
```

```
public static void main(String[] args) {

  Stack<String> stringStack = new Stack<>();
  Stack<Double> doubleStack = new Stack<>();

  doubleStack.sameSize(stringStack);
}
```

This code doesn't compile

The `sameSize` method expects `Stack<Double>`

Wildcard arguments 2/3

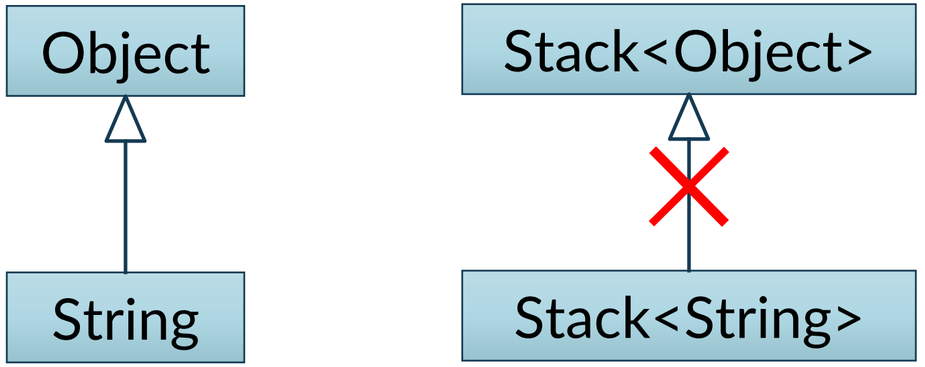
```
public class Stack<T> {

  public int getSize() {…}

  public void push(T value) {…}

  public T top() {…}

  public T pop() {…}

  public boolean sameSize(Stack<Object> other) {
    return getSize() == other.getSize();
  }
}
```

```
public static void main(String[] args) {

  Stack<String> stringStack = new Stack<>();
  Stack<Double> doubleStack = new Stack<>();

   doubleStack.sameSize(stringStack);
}
```

The main method still doesn't compile

String is a subclass of Object

Stack<String> is not a "subclass" of Stack<Object>



Wildcard arguments 3/3

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}

    public boolean sameSize(Stack<?> other) {
        return getSize() == other.getSize();
    }
}
```

The `sameSize` method now expects a `Stack<?>` that means a Stack with any parameterization

```
public static void main(String[] args) {

    Stack<String> stringStack = new Stack<>();
    Stack<Double> doubleStack = new Stack<>();

    doubleStack.sameSize(stringStack);
}
```

This code compiles

Multiple parameters

A generic can define multiple type parameters

```
public class Pair<F, S> {

    private final F first;
    private final S second;

    public Pair(F first, S second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public String toString() {
        return "Pair{first=" + first + ", second=" + second + '}';
    }
}
```

Generic methods 1/3

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}
}
```

Methods inside a generic class can make use of a class type parameter and are, therefore, automatically generic relative to the type parameter

Generic methods 2/3

```
public class Stack<T> {

    public int getSize() {…}

    public T top() {…}

    public T pop() {…}

    public void push(T value) {…}

    public <O extends T> void pushAll(Stack<O> other) {
        while (other.getSize() > 0) {
            push(other.pop());
        }
    }
}
```

Type parameters are declared before the return type

Type parameters are used in the argument list

Generic methods 3/3

```
public class Stack<T> {

    public int getSize() {…}

    public T top() {…}

    public T pop() {…}

    public void push(T value) {…}

    public void pushAll(Stack<? extends T> other) {
        while (other.getSize() > 0) {
            push(other.pop());
        }
    }
}
```

This class is equivalent to the previous one

Wildcards are preferred, they make the code more concise

Generic interfaces

```
public interface Stack<T> {

    int getSize();

    void push(T value);

    T top();

    T pop();
}
```

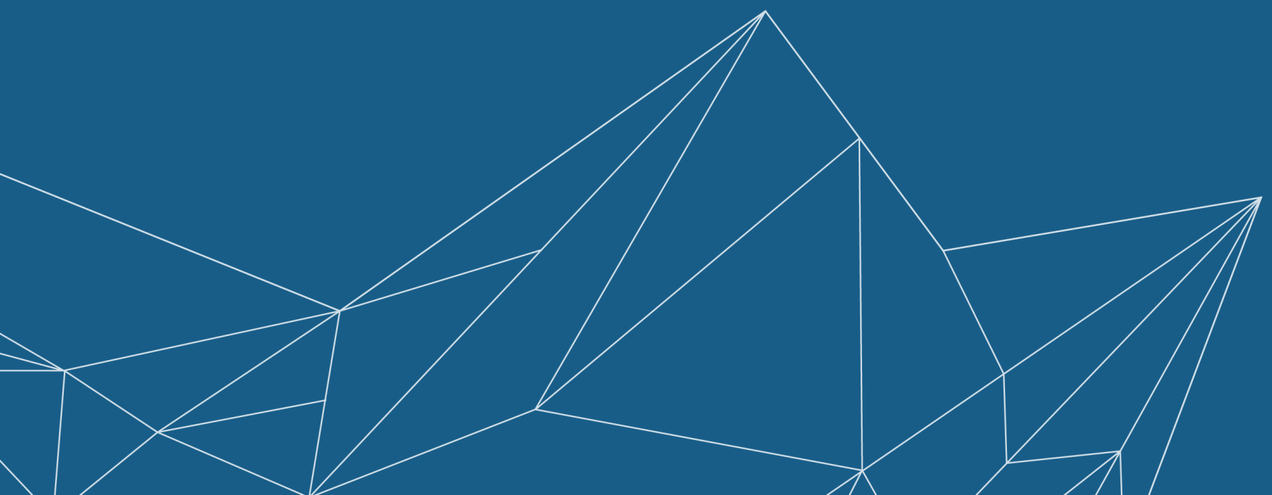
A generic interface is declared in the same way as a generic class

Local variable type inference

```
Stack<String> stringStack1 = new Stack<>();
var stringStack2  = new Stack<String>();
```

The second version is shorter and it should be preferred

Exceptions



How to report error conditions

When implementing a method there are three traditional approaches to report error conditions

Error codes

The method returns an error code.

E. g. 0 if everything is ok, -1 if an error happens, etc.

Error flags

The method set/reset an error flag in its class to report an error condition.

Exception

The method throws an Exception to interrupt its execution and to inform the caller that an exceptional condition arose.

Error codes

```
public class FixedSizeDisplay {

    public static final int OK = 0;
    public static final int TEXT_LENGTH_TOO_BIG = 1;

    private static final int SIZE = 10;

    public int display(String text) {
        if (text.length() > SIZE) {
            System.out.println(text.substring(0, 10));
            return TEXT_LENGTH_TOO_BIG;
        } else {
            System.out.println(text);
            return OK;
        }
    }
}
```

The method cannot return any value, it must return an error code

The caller must always check the returned error code

Error flags

```
public class FixedSizeDisplay {

    private static final int SIZE = 10;

    private boolean error;

    public void display(String text) {
        if (text.length() > SIZE) {
            System.out.println(text.substring(0, 10));
            error = true;
        } else {
            System.out.println(text);
            error = false;
        }
    }

    public boolean checkError() {
        return error;
    }
}
```

The method can return any value

The caller must always check the error status, usually by using a method of the same class

The PrintStream class (the class of System.out) uses this approach.

Throwing exceptions

```
public class FixedSizeDisplay {

    private static final int SIZE = 10;

    public void display(String text) throws Exception {
        if (text.length() > SIZE) {
            System.out.println(text.substring(0, 10));
            throw new Exception("Text length: " +
                text.length() + " exceeds display size");
        }
        System.out.println(text);
    }
}
```

The keyword throw is used to throw exceptions

The caller of display(String) shouldn't check all the invocations, but it needs to deal with exceptional cases only.

When we throw an exception, the method execution is interrupted at the point where the exception is thrown, and the exception is propagated to the caller hierarchy until it is caught by an appropriate try-catch block.

Example

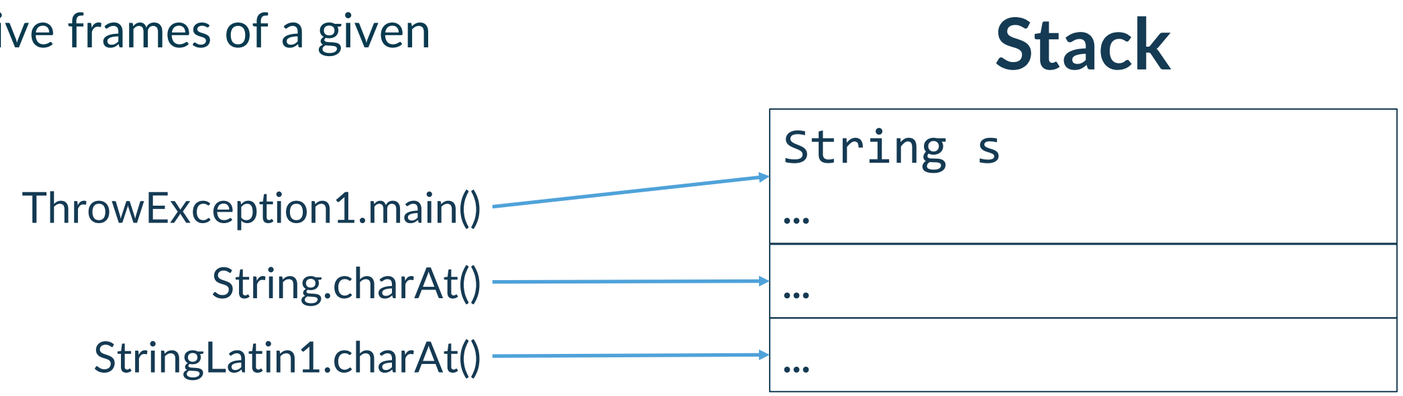
```
public class ThrowException {

    public static void main(String[] args) {
        String s = "Hello";

        System.out.println(s.charAt(10));
    }
}
```

An exception is an Object that describes an exceptional condition. It brings with itself a stack trace and usually an explanatory message.

```
$ java.exe it.units.sdm.exceptions.ThrowException
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 10
        at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)
        at java.base/java.lang.String.charAt(String.java:1512)
        at it.units.sdm.exceptions.ThrowException1.main(ThrowException1.java:8)
```

Stack trace

A stack trace is a report of the active frames of a given thread.



The exception stack trace reports all the method in the stack by indicating the fully-qualified class name and the line number of the last executed instruction for each method.

```
$ java.exe it.units.sdm.exceptions.ThrowException
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 10
        at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)
        at java.base/java.lang.String.charAt(String.java:1512)
        at it.units.sdm.exceptions.ThrowException1.main(ThrowException1.java:8)
```

The same information can be obtained for debugging by invoking the static method Thread.dumpStack()

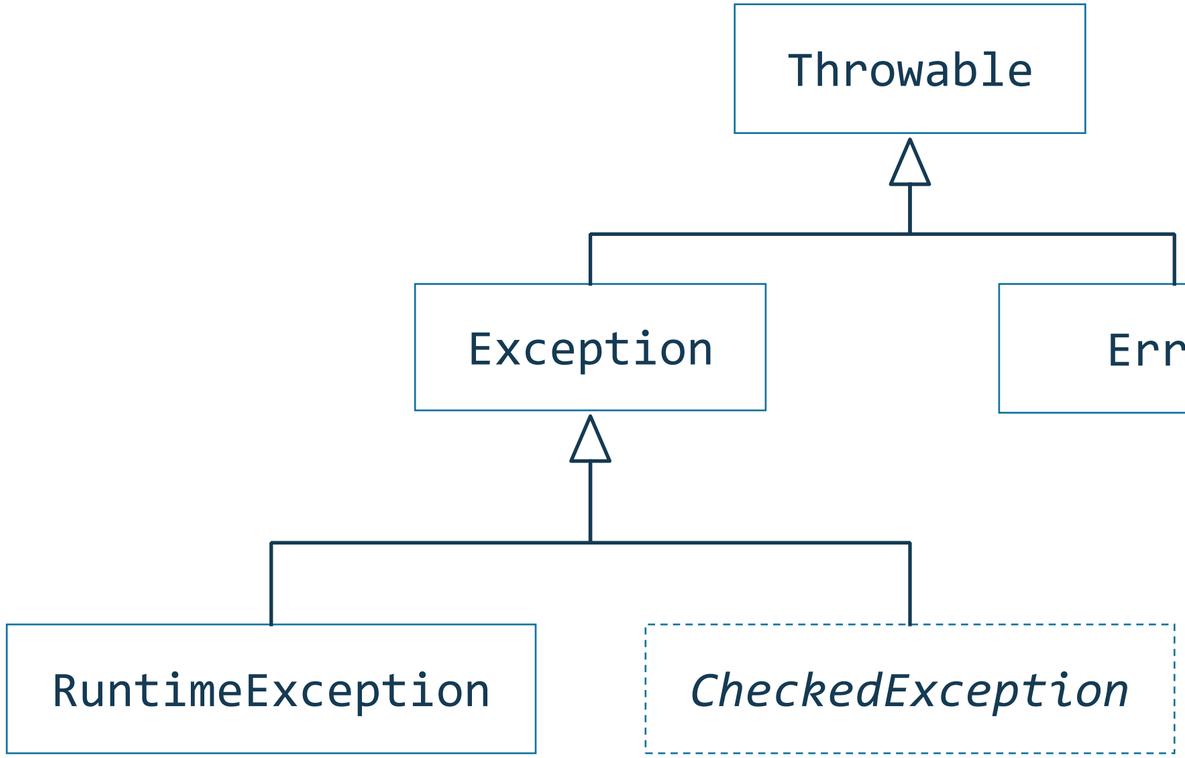
Catching exceptions

```
public class CatchException {

    public static void main(String[] args) {
        String s = "Hello";

        try {
            System.out.println(s.charAt(10));
        } catch (StringIndexOutOfBoundsException ex) {
            System.out.println("An error happened: " + ex.getMessage());
        }
    }
}
```

```
$ java it.units.sdm.exceptions.ThrowException1
An error happened: String index out of range: 10
```

Exceptions can be caught by try-catch blocks

The Exceptions hierarchy



All exceptions are subclasses of Throwable

An Error indicates a serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

RuntimeException and its subclasses are unchecked exceptions. Unchecked exceptions do not need to be declared in a method or constructor's throws clause.

The checked exception classes are all exception classes other than the unchecked exception classes. Checked exceptions need to be declared in method or constructor's throws clause.

Exception types

Throwable	Never catch Throwables.
Error	Errors happening at the JVM level.
Exception	The base class for checked and unchecked exceptions. By Catching Exception you catch all the exceptions that can be solved by the application logic.
RuntimeException (unchecked exceptions)	Might indicate a bug in the application. Usually are not caught because the problem is at the code level. E.g., NullPointerException, ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException, etc.
Checked exception	Indicate exceptions that can be caused by wrong data and they can be addressed by the application logic. You should catch and manage them. E.g., when you catch a FileNotFoundException you can ask the user to indicate another file.

Famous exceptions - Draft

1. NullPointerException
2. ClassCastException
3. ArrayIndexOutOfBoundException
4. IllegalArgumentException
5. IOException (checked)

Creating your own exceptions

```
public class FixedSizeDisplay {

    private static final int SIZE = 10;

    public void display(String text) throws TextTooLongException {
        if (text.length() > SIZE) {
            var newText = text.substring(0, 10);
            System.out.println(newText);
            throw new TextTooLongException(text.length());
        }
        System.out.println(text);
    }

    public static class TextTooLongException extends Exception {

        public TextTooLongException(int size) {
            super("Text length: " + size + " exceeds display size");
        }
    }
}
```

Handling multiple exceptions 1/2

```
public static void main(String[] args) {
    try {
        myMethod();
    } catch (UserException1 ex) {
        // do something
    } catch (UserException2 ex) {
        //do something
    }
}

static void myMethod() throws UserException1, UserException2 {
    if (System.currentTimeMillis() % 5 == 0) {
        throw new UserException1();
    } else {
        throw new UserException2();
    }
}
```

All exceptions must be caught or declared in the method declaration

A method can declare more exceptions

Handling multiple exceptions 2/2

```
public static void main(String[] args) {
    try {
        myMethod();
    } catch (UserException1 | UserException2 ex) {
        // do something
    }
}

static void myMethod() throws UserException1, UserException2 {
    if (System.currentTimeMillis() % 5 == 0) {
        throw new UserException1();
    } else {
        throw new UserException2();
    }
}
```

Exceptions can be caught together. If we perform the same recover operation.

Why don't we use
`catch (Exception ex)`
to catch both the exceptions?

try-catch-finally 1/3

```
public class Storage {

    public void store(String text) throws TimeoutException {
        //do something
    }

    public void close() {
        //close
    }

    public static void main(String[] args) throws TimeoutException {
        var storage = new Storage();
        for (String arg : args) {
            storage.store(arg);
        }
        storage.close();
    }
}
```

The specification of the Storage class says that a storage object must be closed, to be sure that all data has been store.

Are we satisfying the specification?

try-catch-finally 2/3

```
public class Storage {

    public void store(String text) throws TimeoutException {
        //do something
    }

    public void close() {
        //do something
    }

    public static void main(String[] args) throws TimeoutException {
        var storage = new Storage();
        try {
            for (String arg : args) {
                storage.store(arg);
            }
        } finally {
            storage.close();
        }
    }
}
```

The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

try-catch-finally 3/3

```
public static void main(String[] args) throws TimeoutException {
    var storage = new Storage();
    try {
        for (String arg : args) {
            int i = 0;
            while (true) {
                try {
                    storage.store(arg);
                    break;
                } catch (TimeoutException ex) {
                    if (++i == 5) {
                        throw ex;
                    }
                }
            }
        }
    } finally {
        storage.close();
    }
}
```

try-catch-finally blocks can be nested.

Exceptions can be rethrown

Is this code readable?

Chaining exceptions

```
public interface Display {
    void display(String text) throws DisplayException;
}

class PopupDisplay implements Display {

    @Override
    public void display(String text) throws DisplayException {
        try {
            double fontWidth = 100.0 / text.length();
            // display text based on the font width
            …
        } catch (ArithmeticException ex) {
            throw new DisplayException(ex);
        }
    }
}
```

```
Exception in thread "main" it.units.sdm.DisplayException: java.lang.ArithmeticException: / by zero
        at it.units.sdm.PopupDisplay.display(FixedSizeDisplay.java:63)
        at it.units.sdm.MainDisplay.main(FixedSizeDisplay.java:7)
Caused by: java.lang.ArithmeticException: / by zero
        at it.units.sdm.PopupDisplay.display(FixedSizeDisplay.java:60)
        ... 1 more
```

Assignment

```
public interface Collection {

    boolean isEmpty();

    int getSize();

    boolean contains(String string);

    String[] getValues();
}

public interface Stack extends Collection {

    void push(String string);

    String pop();

    String top();
}
```

```
public interface List extends Collection {

    void add(String string);

    String get(int index);

    void insertAt(int index, String string);

    void remove(int index);

    int indexOf(String string);

}
```

Implement the Stack and List interfaces.

Use exceptions to report wrong usages
e. g. illegal indexes, empty stack, etc.

More on exceptions, from the creator of Java

Failure and Exceptions A Conversation with James Gosling

https://www.artima.com/articles/failure-and-exceptions

Assignment

Assignment

Write a class (or a set of classes) that given a string it produces a Term Frequency table.
Consider the option to provide a list of stop words, normalization, etc.
Provide an option to print the table in alphabetical order and by frequency.

"Term frequency (TF) means how often a term occurs in a document. In the context of natural language, terms correspond to words or phrases ..."

Term	Frequency
english	8
language	7
words	12

...

input	7
cactus	1
fireworks	3

Solution of assignment

The TermFrequency class

```
public class TF {

    private final Map<String, Integer> map;

    public TF(Map<String, Integer> map) {
        this.map = new TreeMap<>(map);
    }

    private void print(Map<String, Integer> map) {
        for (var entry : map.entrySet()) {
            System.out.println(entry);
        }
    }

    …
```

```
    …

    public void printAlphabetically() {
        print(map);
    }

    public void printByFrequency() {
        Comparator<String> c = new Comparator<>() {
            @Override
            public int compare(String o1, String o2) {
                int f1 = map.get(o1);
                int f2 = map.get(o2);
                return f1 == f2 ? o1.compareTo(o2) : f1-f2;
            }
        };
        Map<String, Integer> treeMap = new TreeMap<>(c);
        treeMap.putAll(map);
        print(treeMap);
    }
}
```

A TermFrequency builder

Write a class (or a set of classes) that given a string it produces a Term Frequency table

```
public class TFBuilder {

  private Tokenizer tokenizer;
  private Normalizer normalizer;
  private Filter filter;

  public void setTokenizer(Tokenizer tokenizer) {
     this.tokenizer = tokenizer;
  }

  public void setNormalizer(Normalizer normalizer) {
     this.normalizer = normalizer;
  }

  public void setFilter(Filter filter) {
    this.filter = filter;
  }
  …
```

```
  …
  TF build(String text) {
    Collection<String> tokens = tokenizer.tokenize(text);
    Map<String, Integer> map = new HashMap<>();
    for (String term : tokens) {
      term = normalizer.normalize(term);
      if (filter.accept(term)) {
        map.merge(term, 1, new BiFunction<>() {
          @Override
          public Integer apply(Integer v, Integer d) {
            return v + 1;
          }
        });
      }
    }
    return new TF(map);
  }
}
```

Tokenizer & C.

```
public interface Tokenizer {

    Collection<String> tokenize(String text);
}

public interface Normalizer {

    String normalize(String token);
}

public interface Filter {

    boolean accept(String token);
}
```

A usage example

```
public static void main(String[] args) {
    Set<String> stopWords = Set.of("a", "the", "an", "of");

    TFBuilder tfBuilder = new TFBuilder();
    tfBuilder.setTokenizer(new Tokenizer() {
        @Override
        public Collection<String> tokenize(String text) {
            return Arrays.asList(text.split("\\s"));
        }
    });
    tfBuilder.setNormalizer(new Normalizer() {
        @Override
        public String normalize(String token) {
            return token.replaceAll("\\.|,|", "").toLowerCase();
        }
    });
    tfBuilder.setFilter(new Filter() {
        @Override
        public boolean accept(String token) {
            return !stopWords.contains(token);
        }
    });
    TF tf = tfBuilder.build("...");
    tf.printAlphabetically();
    tf.printByFrequency();
}
```

Thank you!

esteco.com