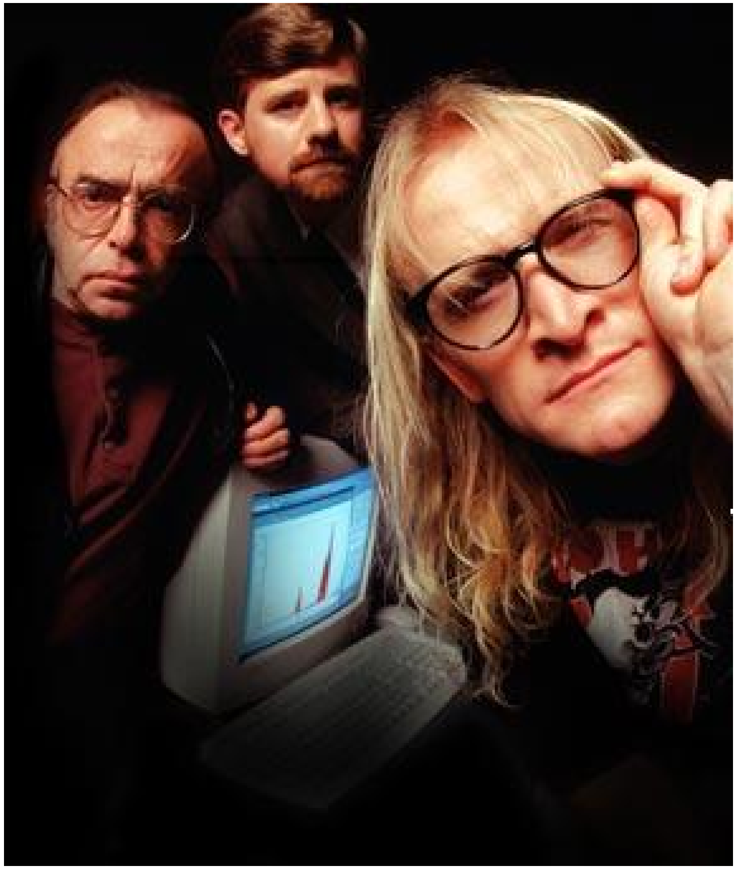
Algoritmid ja andmestruktuurid

- Andmestruktuuride implementatsioonid
- Collections

TTÜ Programmeerimisolümpiaad



IEEExtreme 24h võistlus

- **14-15. okt**

ACM olümpiaad

- **10. okt kell 16.45-22.00**
- **31. okt – 3. nov Minsk**

Registreerige kuni

- 3-liikmeline meeskond aadressil olympiaad@cs.ttu.ee

https://courses.cs.ttu.ee/pages/ProgComp

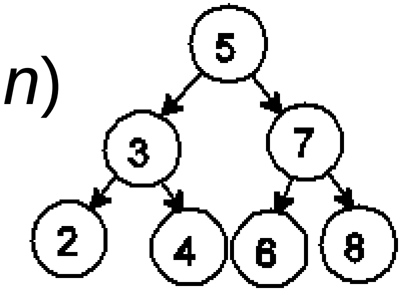
Teated

- Kontrolltöö 23. oktoobril loengu ajal
  - Materjal kuni 6. (tänase) loenguni
  - Kasutada võib paberkandjal materjale

Konteiner

- Konteiner – koondab hulga sama tüüpi objekte. Põhioperatsioonid:
  - lisa objekt
  - kustuta objekt
  - leia objekt
  - anna järgmine objekt

- List, massiiv – mõni operatsioon O(*n*)
- Binaarne otsingupuu – operatsioonid O(lg *n*) eeldusel, et puu on tasakaalus



- *Associative array*
- …

Konteinerid

- Mõned toetavad kiiret otsingut
- Mõned kiiret lisamist/kustutamist
- Mõned efektiivset itereerimist

	LinkList	**Tree**	**HashTable**
Hoidmine	Light	Less light	Medium
Itereerimine	simple	moderate	difficult
Lisamine	O(1)	O(lg n)	O(1)
Kustutamine	O(1)	O(lg n +)	O(1)
Otsing	O(n)	O(lg n)	O(1)

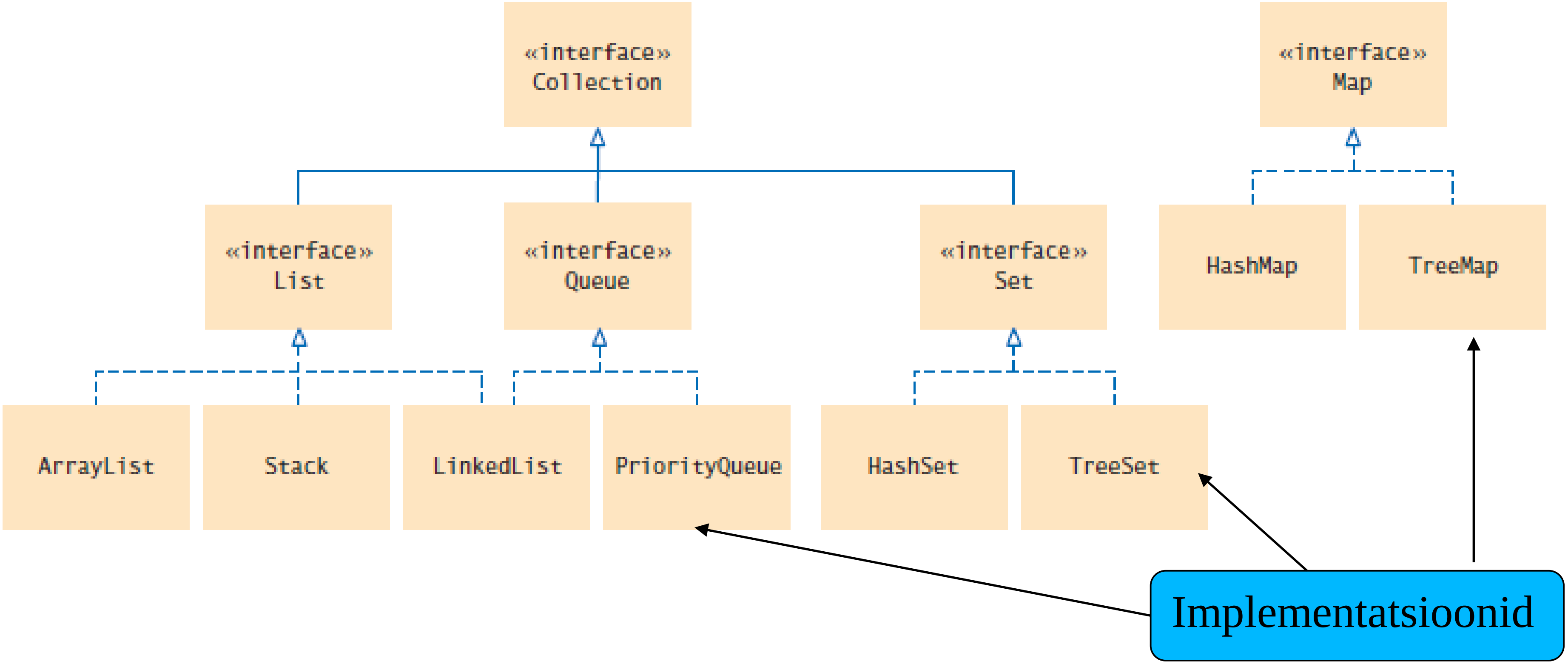
Konteinerite implementatsioonid

- Mitmetes programmeerimiskeeltes on andmestruktuuride valmisteegid
  - Java Collections Framework
  - .Net Framework Class Library - Collections
  - C++ Standard Template Library
  - ...
- On oluline osata neid kasutada ja mõista mis on iga andmestruktuuri taga ning mis keerukused on seetõttu operatsioonidel

Collections

- Liidesed (*interfaces*)
  Abstraktsed andmetüübid, mis määravad kasutuse sõltumata alternatiivsetest implementatsioonidest
  OOP keeltes moodustavad tavaliselt hierarhia
- Implementatsioon (klassid)
  Liideste efektiivsed ja korduvkasutatavad implementatsioonid
- Algoritmid
  Võimaldavad andmestruktuuride operatsioone (sortimine, otsimine) efektiivselt teostada.
  Implementeeritud polümorfselt.

Liidesed

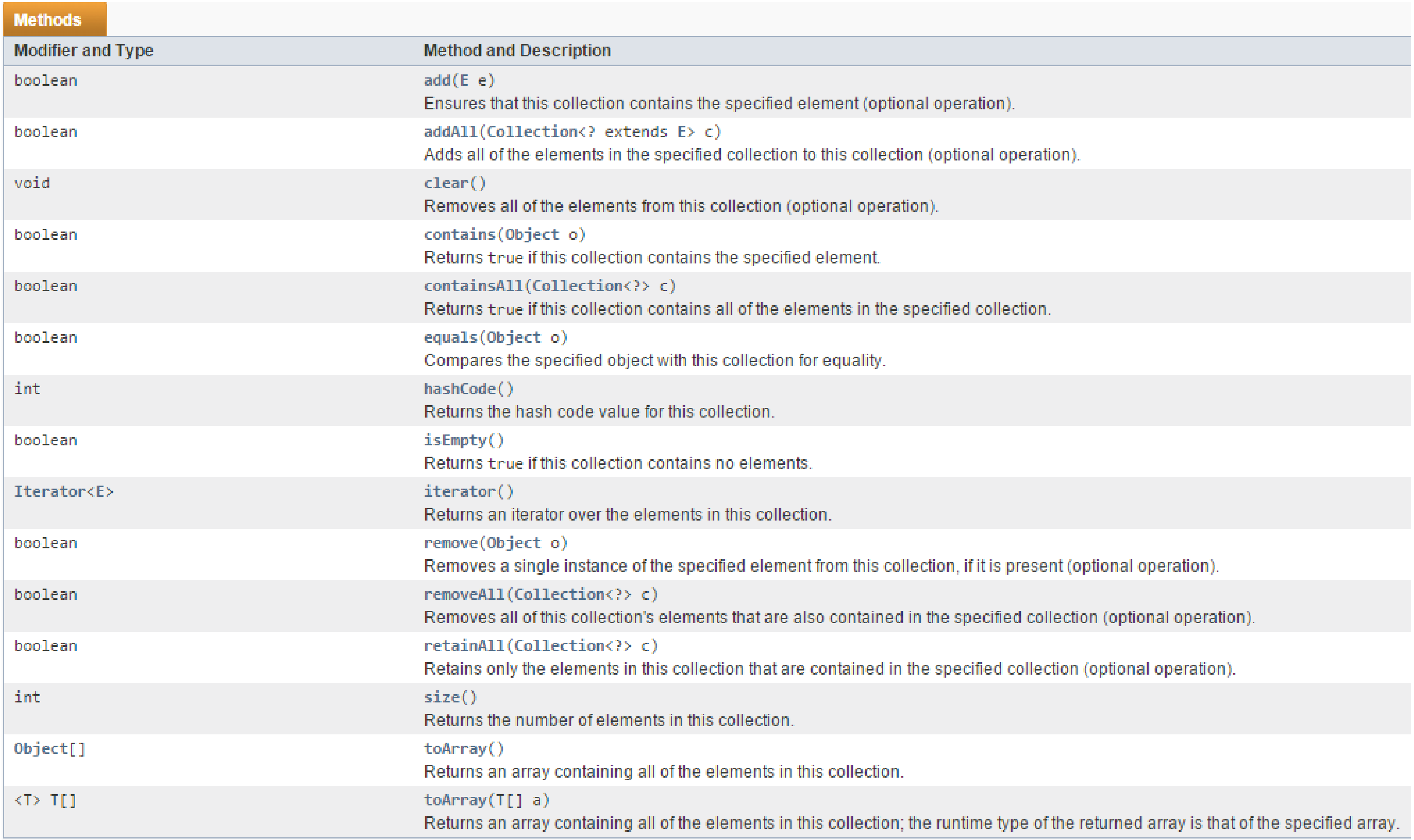

«interface»
Collection
«interface»
Map
«interface»
List
«interface»
Queue
«interface»
Set
HashMap
TreeMap
ArrayList
Stack
LinkedList
PriorityQueue
HashSet
TreeSet
Implementatsioonid

Collections liides

Kõik *Collections* klassid implementeerivad:

- Lisamine – andmestruktuur kasvab, kui andmeid lisatakse
- Kustutamine – anmestruktuur kahaneb, kui andmeid kustutatakse
- Itereerimine – saab itereerida üle kõigi andmete mingis kindlaksmääratud järjekorras
- Suurus – saab küsida objektide arvu andmestruktuuris
- Saab küsida, kas konkreetne objekt on andmestruktuuris

Collections liides



Methods

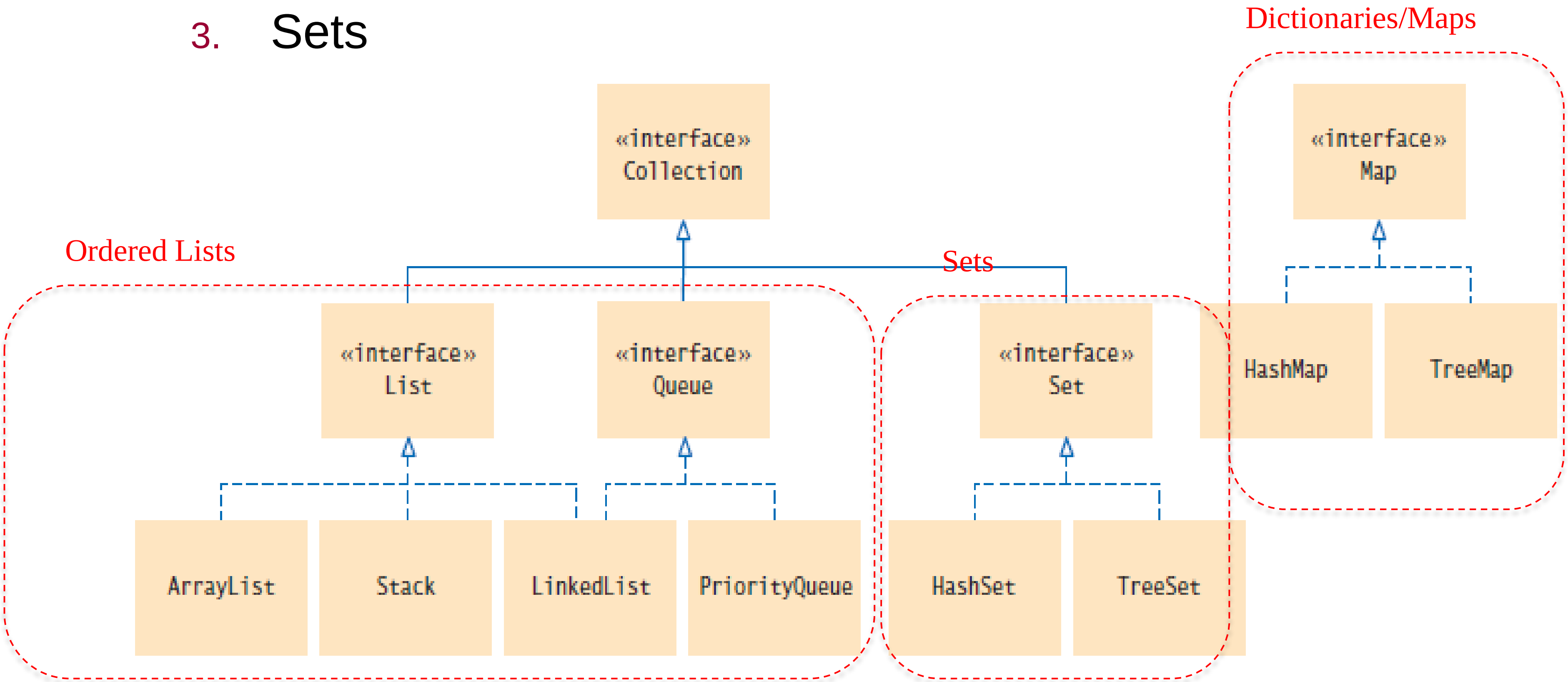
Modifier and Type	Method and Description
`boolean`	`add(E e)` Ensures that this collection contains the specified element (optional operation).
`boolean`	`addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection (optional operation).
`void`	`clear()` Removes all of the elements from this collection (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this collection contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns `true` if this collection contains all of the elements in the specified collection.
`boolean`	`equals(Object o)` Compares the specified object with this collection for equality.
`int`	`hashCode()` Returns the hash code value for this collection.
`boolean`	`isEmpty()` Returns `true` if this collection contains no elements.
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this collection.
`boolean`	`remove(Object o)` Removes a single instance of the specified element from this collection, if it is present (optional operation).
`boolean`	`removeAll(Collection<?> c)` Removes all of this collection's elements that are also contained in the specified collection (optional operation).
`boolean`	`retainAll(Collection<?> c)` Retains only the elements in this collection that are contained in the specified collection (optional operation).
`int`	`size()` Returns the number of elements in this collection.
`Object[]`	`toArray()` Returns an array containing all of the elements in this collection.
`<T> T[]`	`toArray(T[] a)` Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Liidesed ja implementatsioonid

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Collections

- ***Collections* jaotus:**
  1. Ordered lists
  2. Dictionaries
  3. Sets



1. **Ordered Lists:**
   - Allows us to insert items in a particular order
   - Allow later retrieving them in some pre-defined order
   - Specific objects can also be retrieved based on their position in the list
   - By **default**, items are added at the end of an ordered list
   - E.g. a **student waiting list**:
     - Order maintenance is important to be fair in selecting students from waiting list
   - Ordered lists are realized in java using :
     - `List` interface
     - `Queue` interface

List Interface Implementations

- **ArrayList** - FIFO
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- **ArrayDequeue**
  - Array implementation of queue and stack
- **LinkedList**
  - sequential access
  - low cost insert and delete
  - high cost random access
  - Can be used as a queue and stack

Using the Enhanced `for` Loop with Array Lists

- E.g. print elements in ArrayList *names*:

```
for (String name : names){
   System.out.println(name);
}
```

- This is equivalent to:

```
for (int i = 0; i < names.size(); i++){
   String name = names.get(i);
   System.out.println(name);
}
```

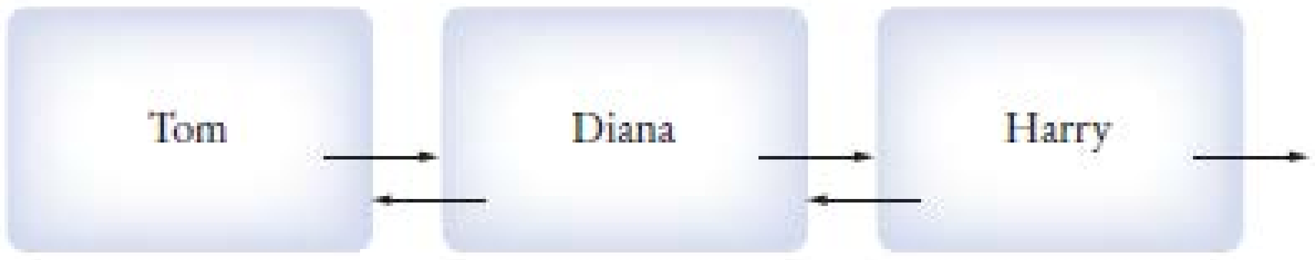
Choosing Between Array Lists and Arrays

- For most programming tasks, array lists are easier to use than arrays
  - Array lists can grow and shrink.
  - Arrays have a nicer syntax.

- Recommendations
  - If the size of a collection never changes, use an array.
  - If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.
  - Otherwise, use an array list.

Linked Lists

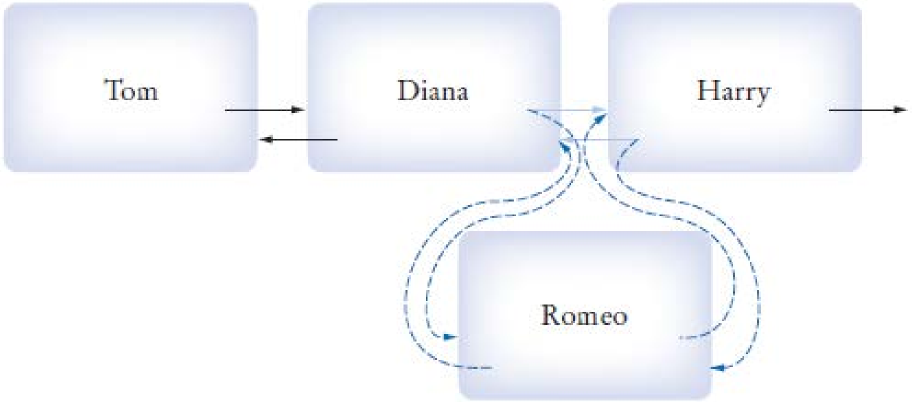
- Doubly-linked list implementation of the List interface.
- A linked list **consists of a number of nodes**
- Each node stores element + has references to the next node and previous node.



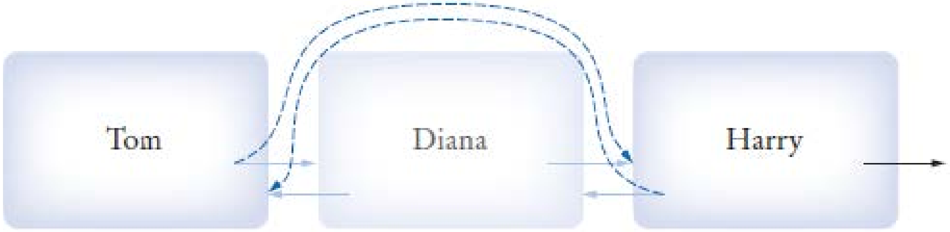
  - **Visiting the elements** of a linked list **in sequential order** is efficient.
  - **Random access** is **NOT** efficient.

Linked Lists

- **Adding and removing elements** in the middle of a linked list is efficient.
- When **inserting/adding or removing a node**:
  - Only the neighboring node references need to be updated (Unlike arrays!)



Adding a new node with element=“Romeo”



Removing node with element=“Diana”

The LinkedList Class of the Java Collections Framework

- Some additional `LinkedList` methods:

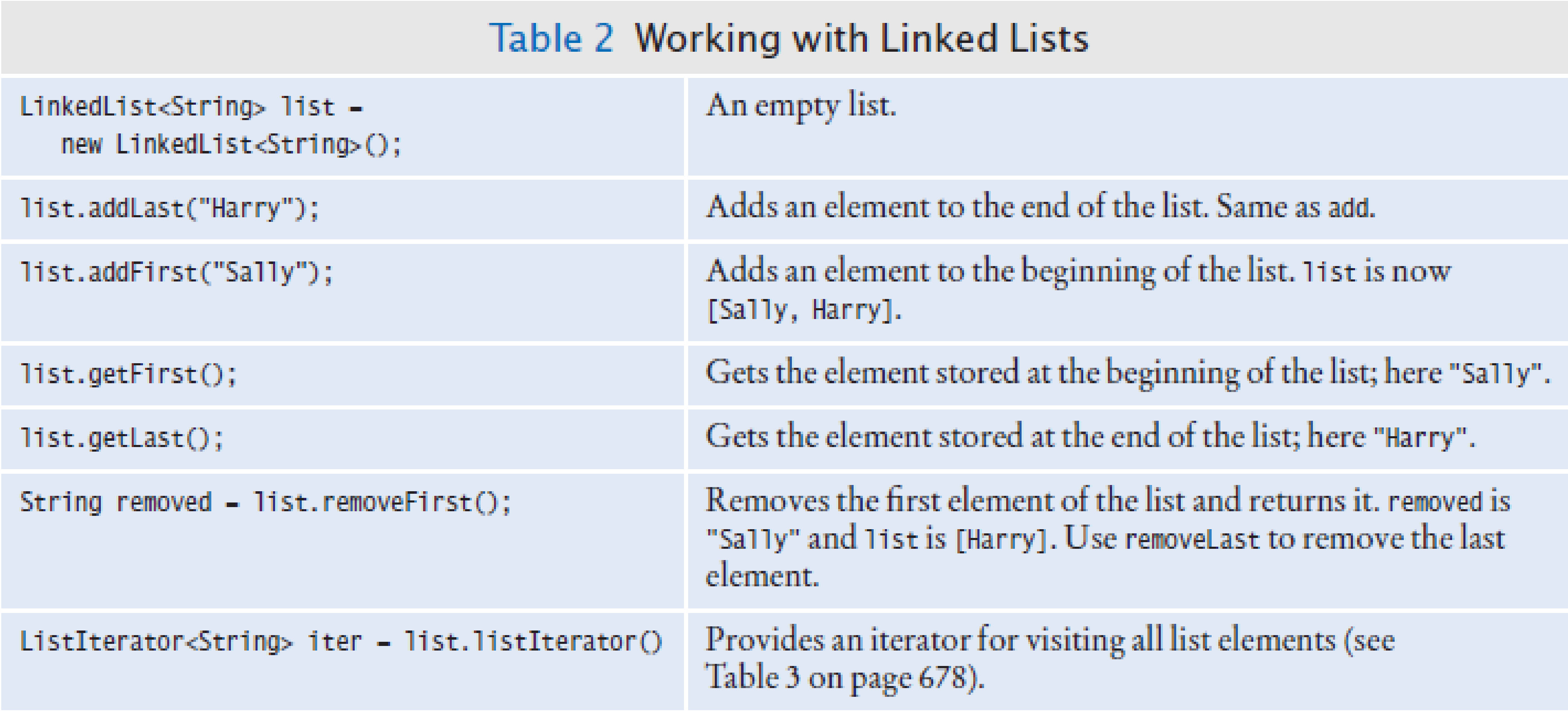


Table 2 Working with Linked Lists

`LinkedList<String> list =`
`new LinkedList<String>();`
	An empty list.

`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 678).

Refer javadoc api

List Iterator

- To **traverse all elements in a linked list of strings, use next() method:**

Using while loop:

```
while (iterator.hasNext())
{
   String name = iterator.next();
   //Do something with name
}
```

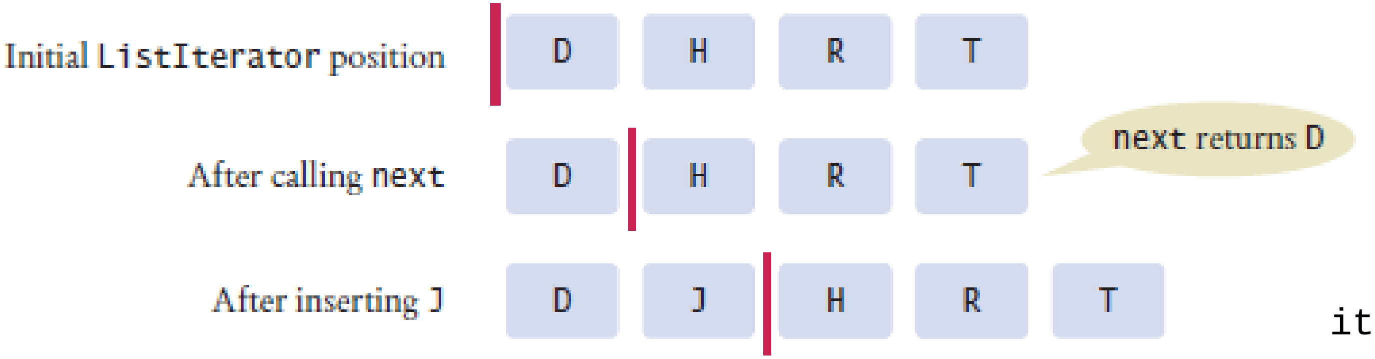
- Smilar to “for each” loop:

```
for (String name : employeeNames)
{
   //Do something with name
}
```

List Iterator

- iterator points between two elements:
- The `add` method:
  - adds an object <u>after</u> the iterator.
  - Then <u>moves</u> the iterator position <u>past</u> the new element.

```
iterator.add("Juliet");
```



```
iterator.next()
```

```
iterator.add(“J”)
```

A Conceptual View of the List Iterator

List Iterator

- `ListIterator` interface extends `Iterator` interface.
- Methods of the `Iterator` and `ListIterator` interfaces

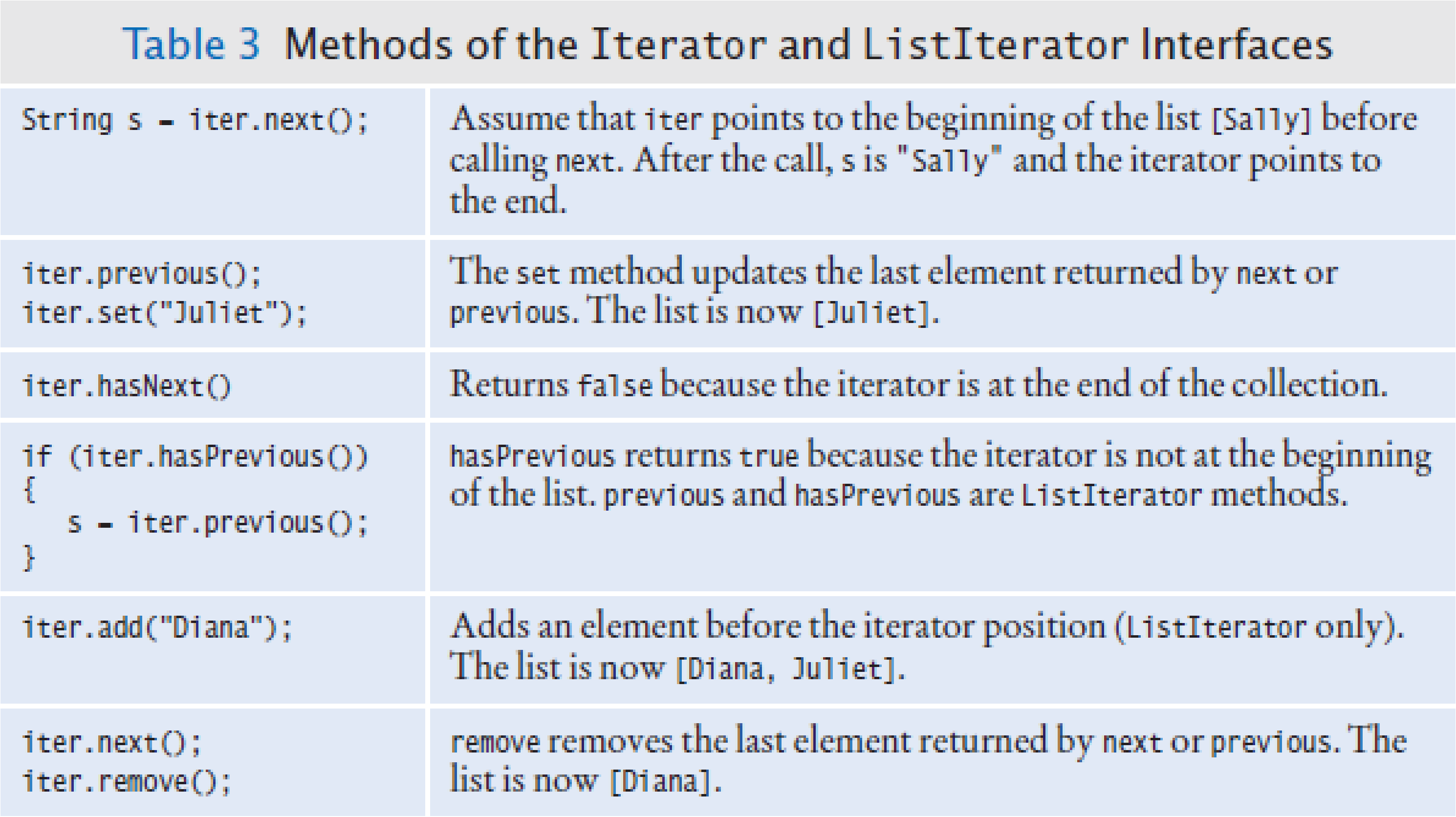


Table 3 Methods of the Iterator and ListIterator Interfaces

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.

`iter.previous();`
`iter.set("Juliet");`
	The `set` method updates the last element returned by `next` or `previous`. The list is now `[Juliet]`.

`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`   s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list. `previous` and `hasPrevious` are `ListIterator` methods.

`iter.add("Diana");`	Adds an element before the iterator position (`ListIterator` only). The list is now `[Diana, Juliet]`.

`iter.next();`
`iter.remove();`
	`remove` removes the last element returned by `next` or `previous`. The list is now `[Diana]`.

Sets

- An unordered collection
- i.e. you CANNOT ask for a particular item by number/position once it has been inserted into the set.
- We can iterate though elements one by one
  - But, **order is not predetermined**
- Duplicate entries aren't allowed in a set
  - Unlike lists
- E.g. group employees by department
- **Inserting and removing** elements is more efficient with a set than with a list.

Sets

- Two implementing classes :
  - `HashSet`
    - based on hash table
  - `TreeSet`
    - based on binary search tree
- A `Set` implementation arranges the elements so that it can locate them quickly.

Sets

- **HashSet**
  - Elements are internally grouped according to a hashcode
- **TreeSet**
  - Elements are kept in sorted order
  - The nodes are arranged in a tree shape**, not in a linear sequence**
  - You can form tree sets for any class that implements the `Comparable` interface (must implement compareTo method):
    - Example: `String` or `Integer`.
  - **Use a `TreeSet` if you want to visit the set's elements in sorted order.**
    - Otherwise choose a `HashSet`
      - It is a **more efficient** — if the hash function is well chosen

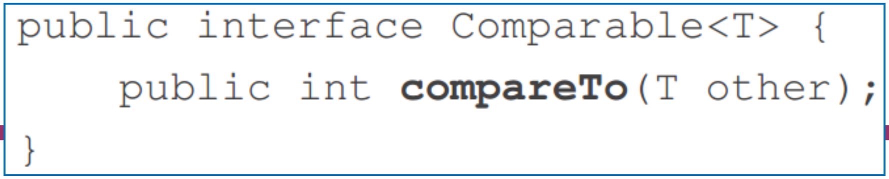
Comparable Interface

- For a class to be used as element type in a TreeSet, class must implement Comparable interface

- A class implementing Comparable interface should implement `compareTo` method

For two object `obj1` , `obj2` of same type, a call of `obj1.compareTo( obj2)` should return:

- a value < 0 if `obj1` comes "before" `obj2` in the ordering (`obj1 < obj2`)
  - usually `return -1`
- a value > 0 if `obj1` comes "after" `obj2` in the ordering, (`obj1 > obj2`)
  - usually `return 1`
- exactly 0 if `obj1` and `obj2` are considered "equal" in the ordering (`obj1 = obj2`)
  - `return 0`



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

LinkedHashSet

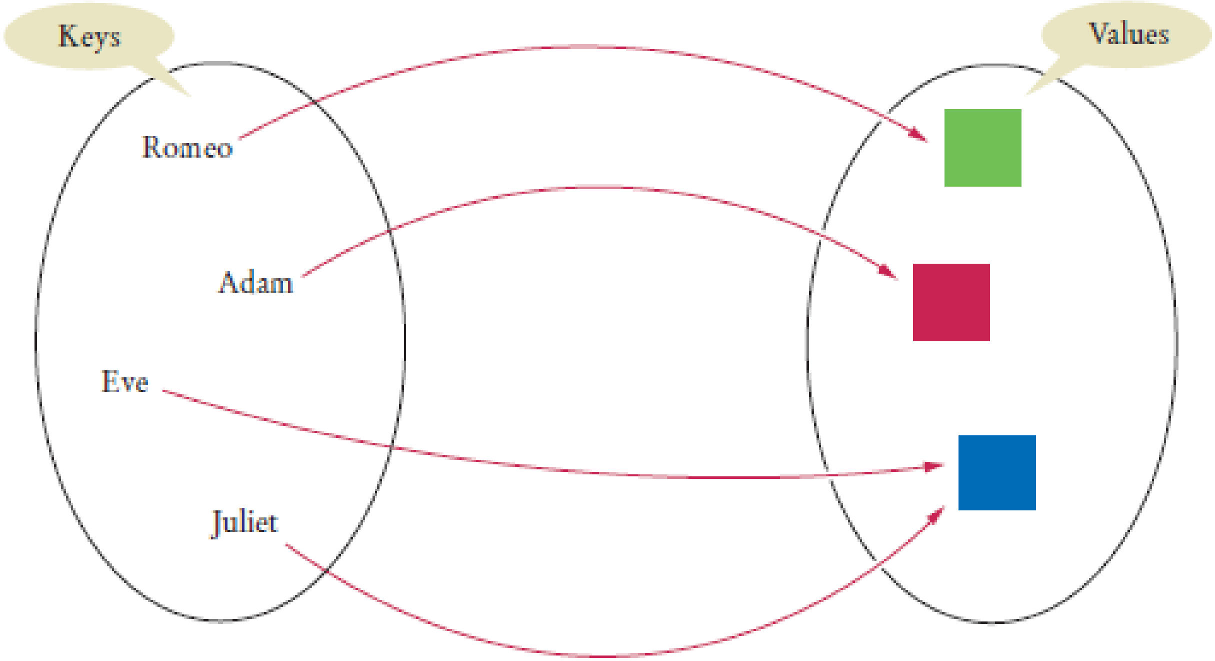
- Extends HashSet
- Maintains a linked list of entries of the HashSet in the order they were inserted
- Can be used to iterate over HashSet in efficient manner

Dictionaries/ Maps

- Provides a means for storing each object reference along with a unique lookup key that can later be used to quickly retrieve the object
- The **key** is often selected based on one or more of the object's attribute values.
  - E.g. a Student object's **student ID number would make an excellent key**, because its value is inherently unique for each Student.

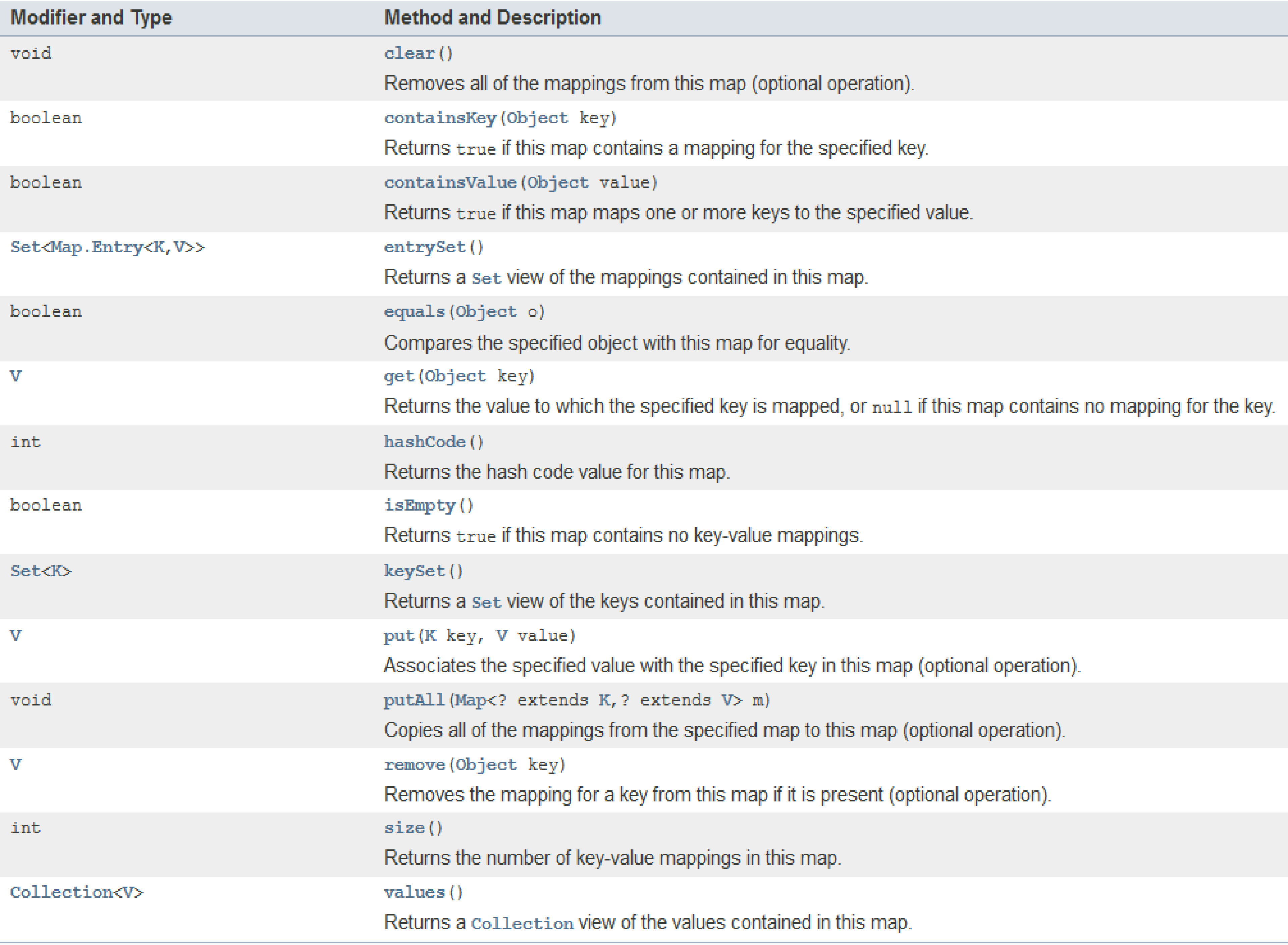
Maps

- A map allows you to associate elements from a **key set** with elements from a **value collection**.
- Use a map when you want to look up objects by using a key.
- No duplicate keys allowed



Dictionaries/ Maps

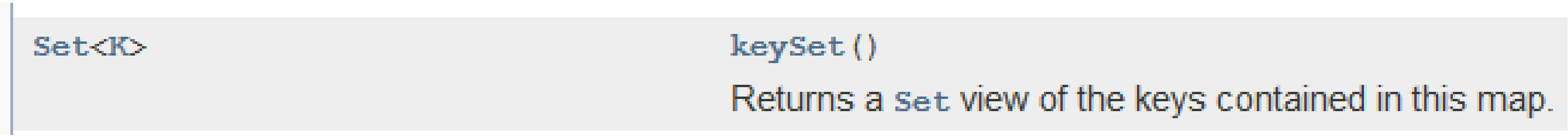
- Some **predefined Java classes** that implement the notion of a dictionary are:
  - HashMap
  - TreeMap
    - The TreeMap is sorted according to the natural ordering of its **keys**, or by a Comparator provided at map creation time
    - guaranteed log(n) time cost for the containsKey, get, put and remove operations



Modifier and Type	Method and Description
`void`	`clear()` Removes all of the mappings from this map (optional operation).
`boolean`	`containsKey(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.
`boolean`	`equals(Object o)` Compares the specified object with this map for equality.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`int`	`hashCode()` Returns the hash code value for this map.
`boolean`	`isEmpty()` Returns `true` if this map contains no key-value mappings.
`Set<K>`	`keySet()` Returns a `Set` view of the keys contained in this map.
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map (optional operation).
`void`	`putAll(Map<? extends K,? extends V> m)` Copies all of the mappings from the specified map to this map (optional operation).
`V`	`remove(Object key)` Removes the mapping for a key from this map if it is present (optional operation).
`int`	`size()` Returns the number of key-value mappings in this map.
`Collection<V>`	`values()` Returns a `Collection` view of the values contained in this map.

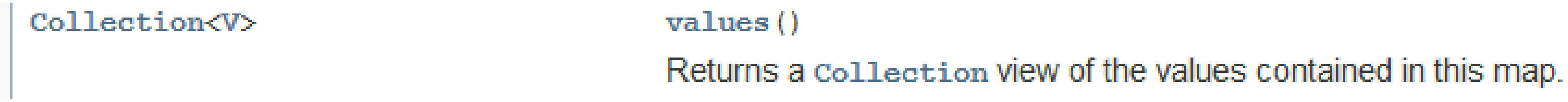
Questions

Why is the collection of the keys of a map a set and not a list?



`Set<K>`	`keySet()` Returns a `Set` view of the keys contained in this map.

Why is the collection of the values of a map not a set?



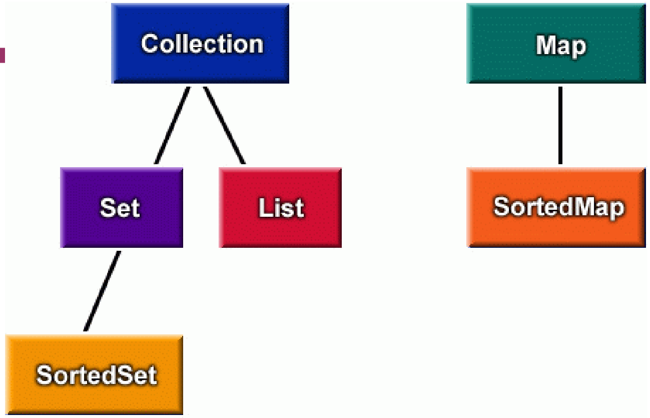
`Collection<V>`	`values()` Returns a `Collection` view of the values contained in this map.

Problem

Suppose you want to track how many times each word occurs in a document. What datastructure would you use?

Suppose you want to check if the string in question is a keyword or not. What datastructure would you use?

Java Collections and Map interface



Collection

+add(element: Object): boolean
+addAll(collection: Collection): boolean
+clear(): void
+contains(elment: Object): boolean
+containsAll(collection: Collection):boolean
+equals(object: Object): boolean
+hashcode(): int
+iterator(): Iterator
+remove(element: Object): boolean
+removeAll(collection: Collection): boolean
+retainAll(collection: Collection): boolean
+size(): int
+toArray(): Object[]
+toArray(array: Object[]): Object[]

Map

+clear() : void
+containsKey(key: Object) : boolean
+containsValue(value: Object) : boolean
+entrySet() : Set
+get(key: Object) : Object
+isEmpty() : boolean
+keySet() : Set
+put(key: Object, value: Object) : Object
+putAll(m: Map) : void
+remove(key: Object) : Object
+size() : int
+values() : Collection

Java Collections

-

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

- **TreeSet** ja **TreeMap** on **SortedSet** ja **SortedMap** implementatsioonid
- **Interfaces**
  - **List** - objektide järjestatud kogu – *insert* ja *delete* säilitavad järjestuse
  - **Set** - objektide kogu, korduvad objektid pole lubatud, järjestus pole garanteeritud
  - **Map** - võtme ja objekti paaride kogu, otsimine võtme alusel, võtmed unikaalsed

Millal kasutada

- **ArrayList** – dünaamiline massiiv
  - Kiire otsepöördus
- **LinkedList** – lingitud list
  - Kiire lisamine ja kustutamine algusest ja keskelt
  - addFirst, getFirst, removeFirst, addLast, getLast, removeLast, clone

- **TreeSet** ja **TreeMap**
  - järjestatud ligipääas, *min*, *max* - (O(log *n*))

- **HashSet** ja **HashMap**
  - Kiire pöördumine, *insert*, *remove.* Järjestus pole oluline.

Kokkuvõtteks konteineritest

- Hash tabelil põhinevad andmestruktuurid – O(1)
  - puudub järjestus
- Puudel põhinevad andmestruktuurid – O(log *n*)
  - võrdlusoperatsiooni põhine järjestus
- Massiiv, dünaamiline massiiv, lingitud list
  - mõned lihtsamad operatsioonid O(1)
  - teised operatsioonid O(*n*)

Oluline on osata leida sobiv andmestruktuur