Collections & Maps

Lecture 12

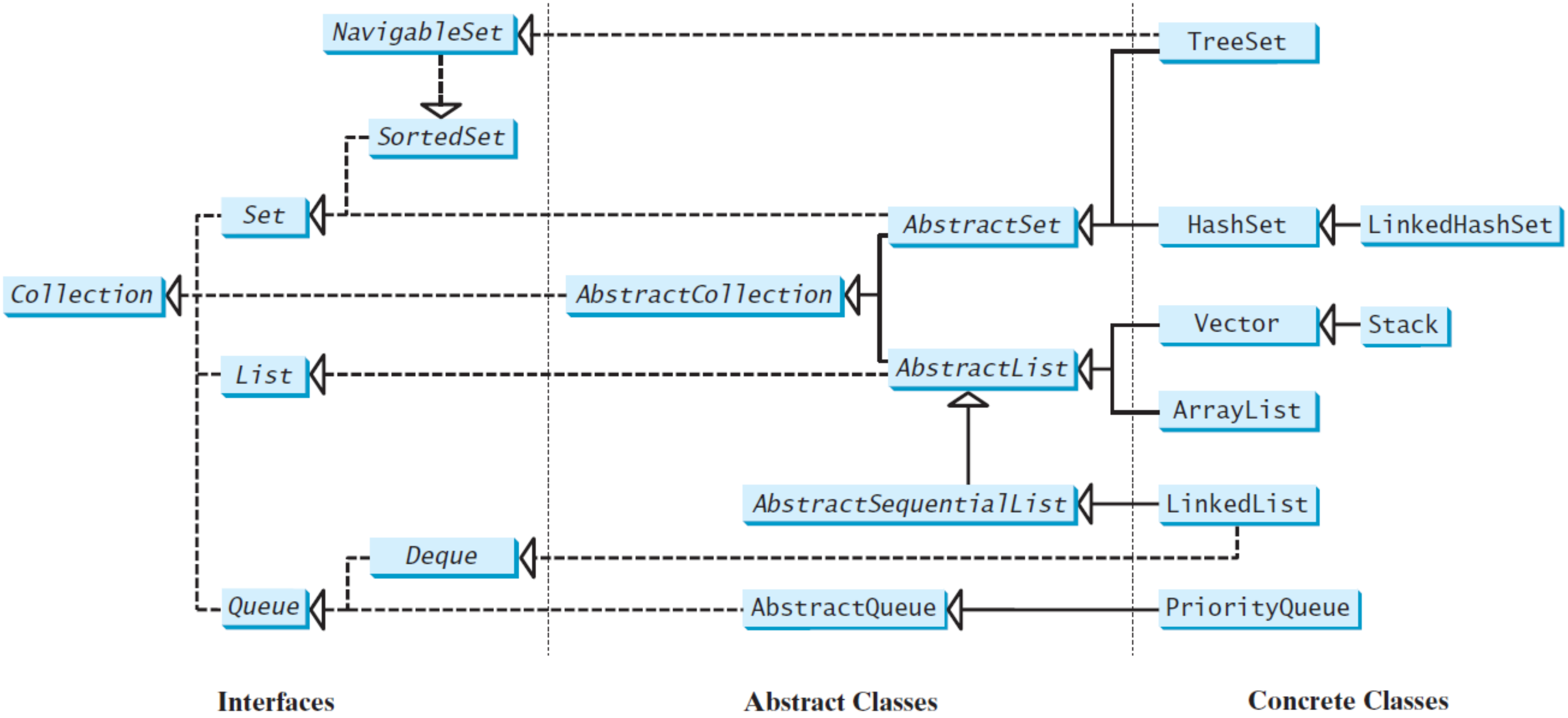
Review: **Data Structure**

- A **data structure** is a collection of data organized in some fashion
- The structure not only stores data but also supports operations for accessing/manipulating the data
- Java provides several data structures that can be used to organize/manipulate data efficiently in the **Java Collections Framework**

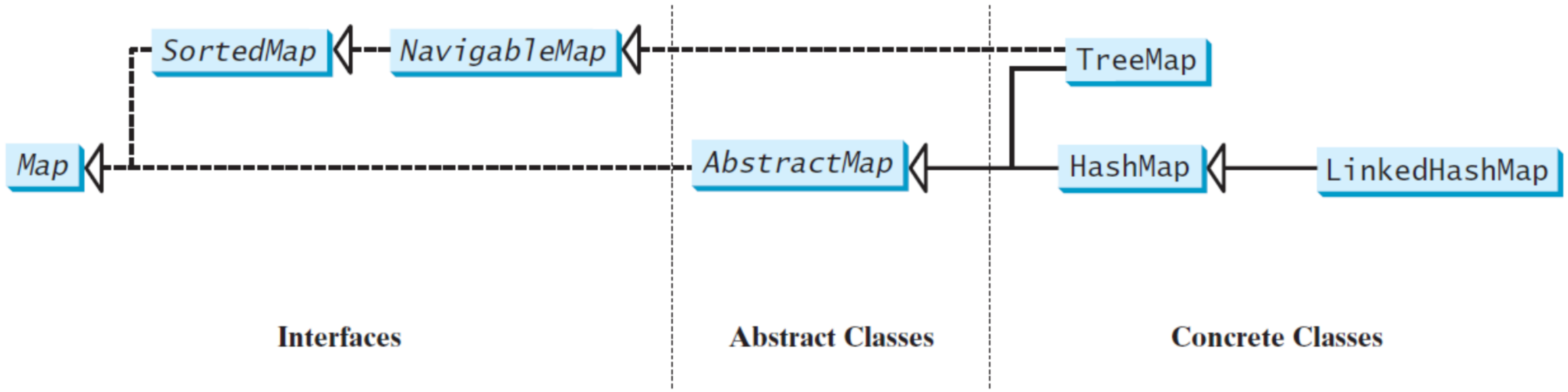
Review: JCF

- The Java Collections Framework supports two types of containers:
  - Storing a collection of elements (**collection**)
  - Storing key/value pairs (**map**)

Review: JCF Hierarchy (1)



Review: JCF Hierarchy (2)



All the Useful!

Name	What is it good for*?	Implementations
`List`	A sequence; access via index, search	`ArrayList`, `Vector`, `LinkedList`
	+ push, pop, peek (i.e. ***LIFO***)	`Stack`
`Set`	Distinct objects (i.e. no duplicates)	`Linked/HashSet`, `TreeSet`
`Queue`	Hold elements in line (i.e. ***FIFO***)	`LinkedList`
	+ custom ordering	`PriorityQueue`
`Map`	Associate two objects	`HashMap`
	+ ordered iteration	`LinkedHashMap`, `TreeMap`

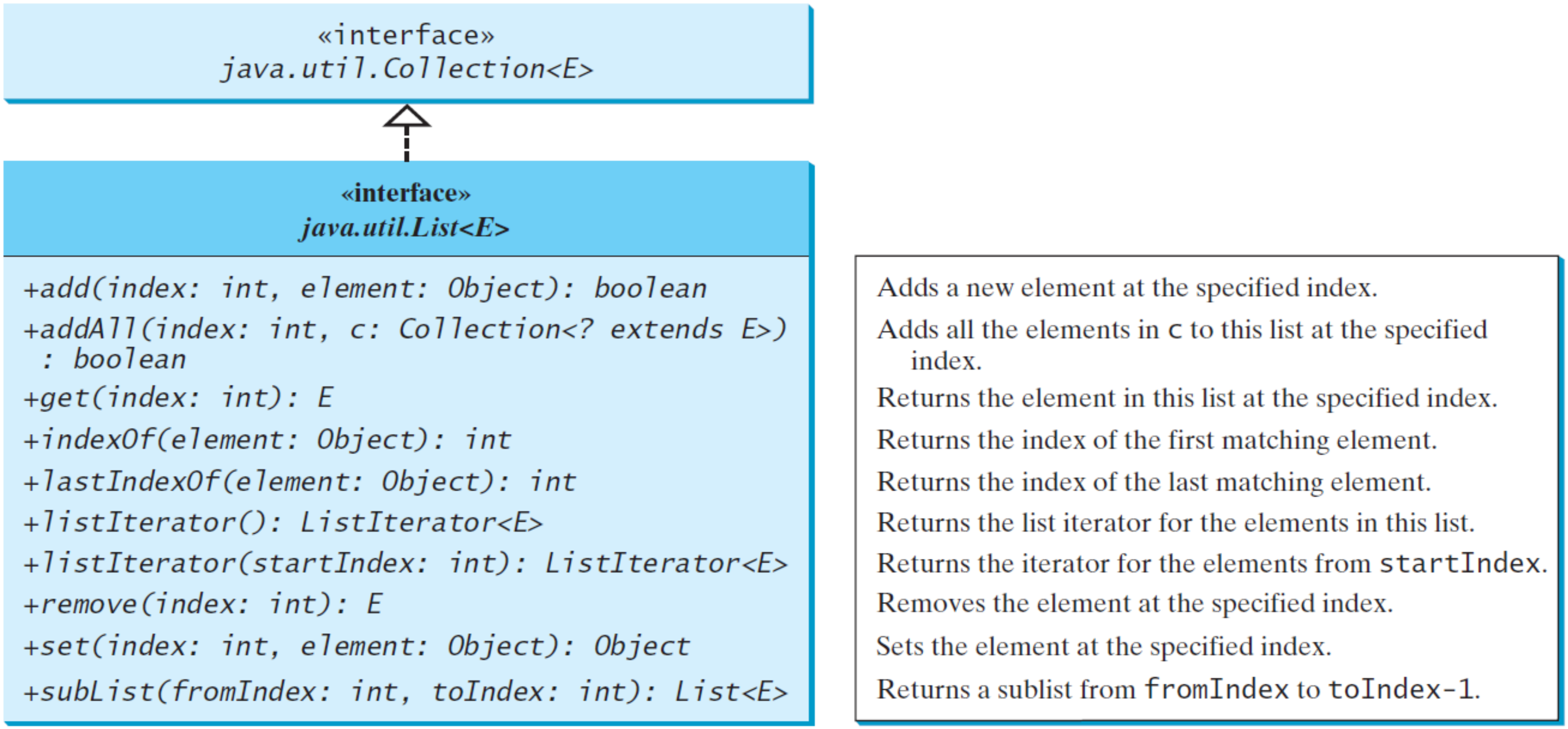
**War Absolutely nothing! Huh, Yeah*

Lists

- The **`List`** interface extends the **`Collection`** interface

- A list stores elements in a sequential order, and allows the user to specify where the element is store
  - The user can access the elements by index

The List Interface



«interface»
java.util.Collection<E>

«interface»
java.util.List<E>

+add(index: int, element: Object): boolean	Adds a new element at the specified index.
+addAll(index: int, c: Collection<? extends E>) : boolean	Adds all the elements in c to this list at the specified index.
+get(index: int): E	Returns the element in this list at the specified index.
+indexOf(element: Object): int	Returns the index of the first matching element.
+lastIndexOf(element: Object): int	Returns the index of the last matching element.
+listIterator(): ListIterator<E>	Returns the list iterator for the elements in this list.
+listIterator(startIndex: int): ListIterator<E>	Returns the iterator for the elements from startIndex.
+remove(index: int): E	Removes the element at the specified index.
+set(index: int, element: Object): Object	Sets the element at the specified index.
+subList(fromIndex: int, toIndex: int): List<E>	Returns a sublist from fromIndex to toIndex-1.

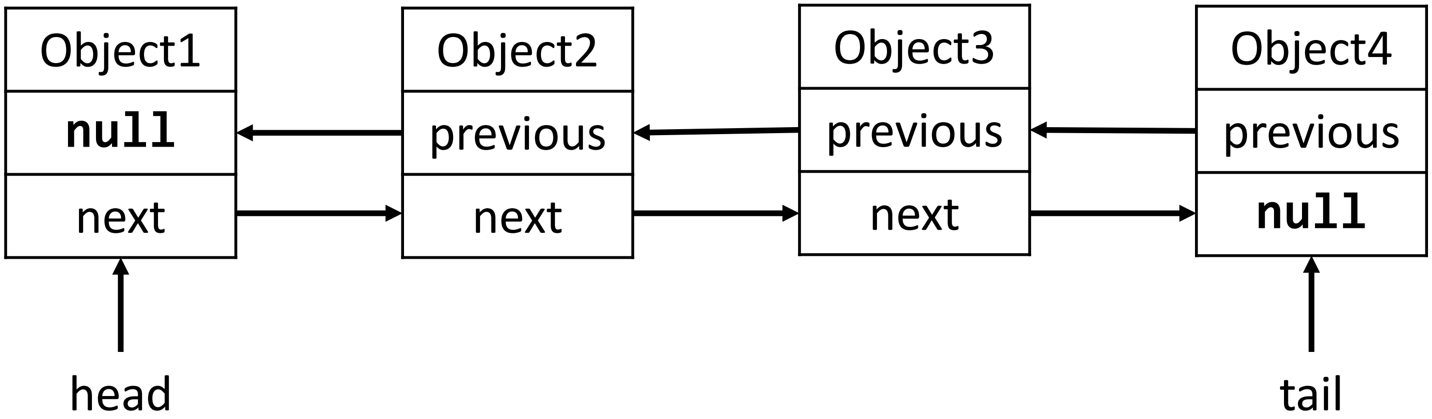
ArrayList vs. **Vector**

- The **ArrayList** and **Vector** classes both use arrays internally, which grow as needed
- The main difference: **Vector** is safe when multiple "threads" (think methods) access the data at the same time
  - Like **StringBuilder** vs. **StringBuffer**
- Because of this, by default, use **ArrayList** for faster operation

`LinkedList`

Whereas **`ArrayList`** and **Vector** encapsulate arrays, **`LinkedList`** operates via a chain of references between individual data “nodes”

- Time to access a node? Search?
- Time to add/remove a node?

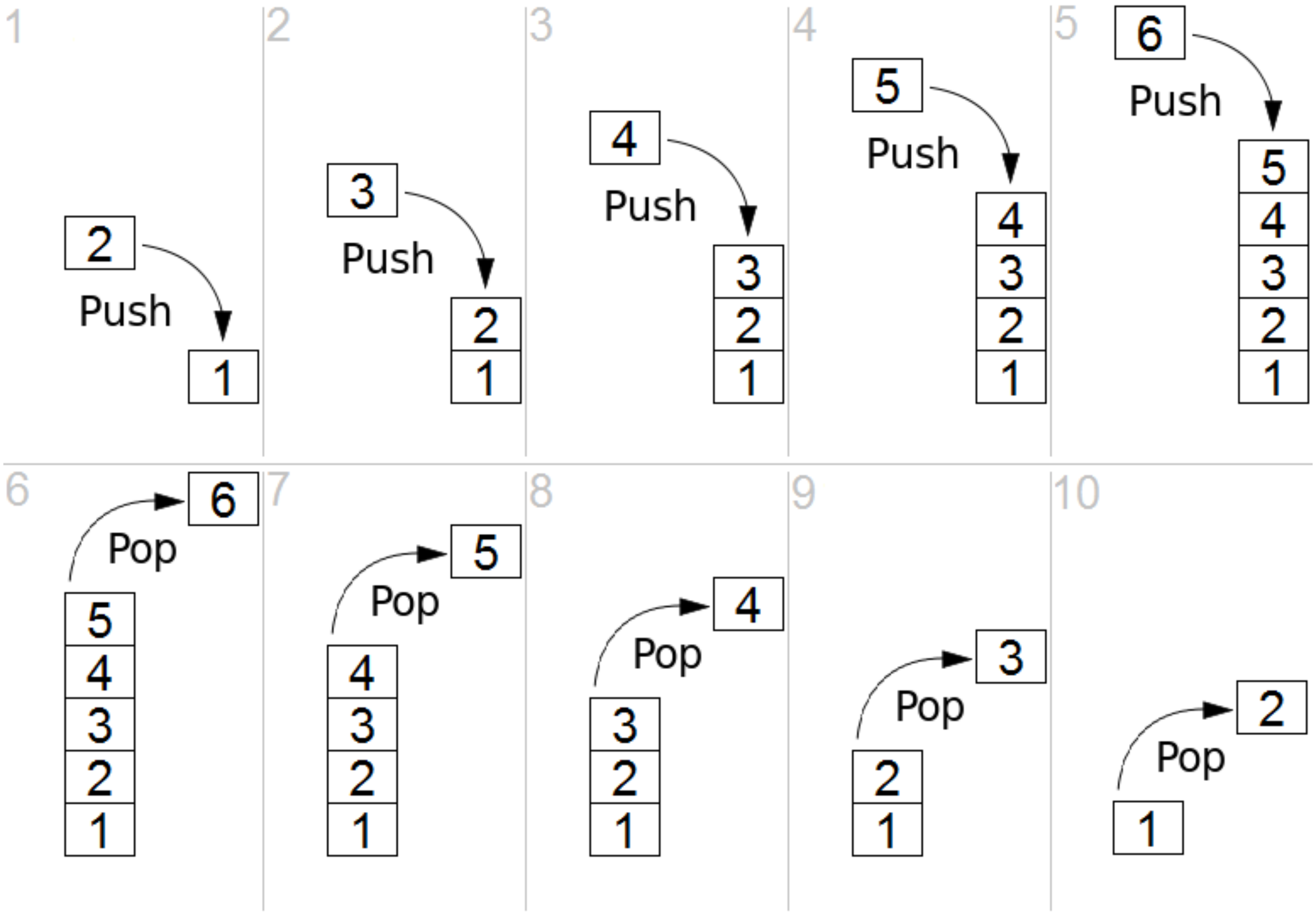


`ArrayList` vs. `LinkedList`

- **`ArrayList`** and **`LinkedList`** are both lists, but which you should use depends on what your program is *mostly* doing
  - Access/Search: **`ArrayList`**
  - Add/Remove: **`LinkedList`**

- An example of tradeoffs that occur throughout Computer Science :)

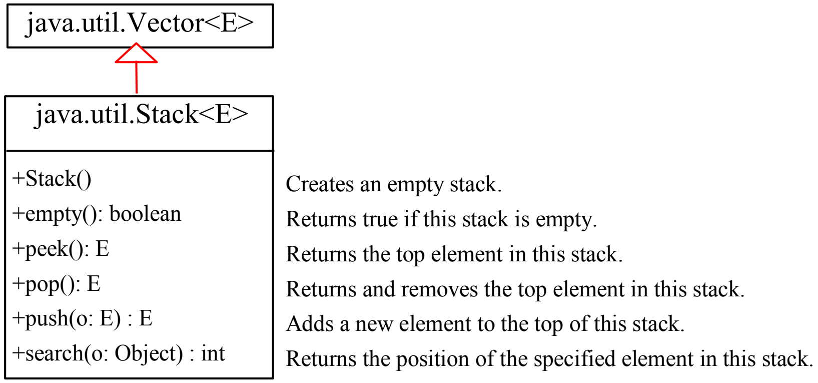
A Stack: Last in, First Out (LIFO)



The **`Stack`** Class

- Stacks come in quite handy in a variety of situations
  - See the book for an example of evaluating mathematical expressions

- The **`Stack`** class is a subclass of **`Vector`**



java.util.Vector<E>

java.util.Stack<E>

+Stack()	Creates an empty stack.
+empty(): boolean	Returns true if this stack is empty.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E) : E	Adds a new element to the top of this stack.
+search(o: Object) : int	Returns the position of the specified element in this stack.

Making Progress!

Name	What is it good for?	Implementations
`List`	A sequence; access via index, search	`ArrayList`, `Vector`, `LinkedList`
	+ push, pop, peek (i.e. ***LIFO***)	`Stack`
`Set`	Distinct objects (i.e. no duplicates)	`Linked/HashSet`, `TreeSet`
`Queue`	Hold elements in line (i.e. ***FIFO***)	`LinkedList`
	+ custom ordering	`PriorityQueue`
`Map`	Associate two objects	`HashMap`
	+ ordered iteration	`LinkedHashMap`, `TreeMap`

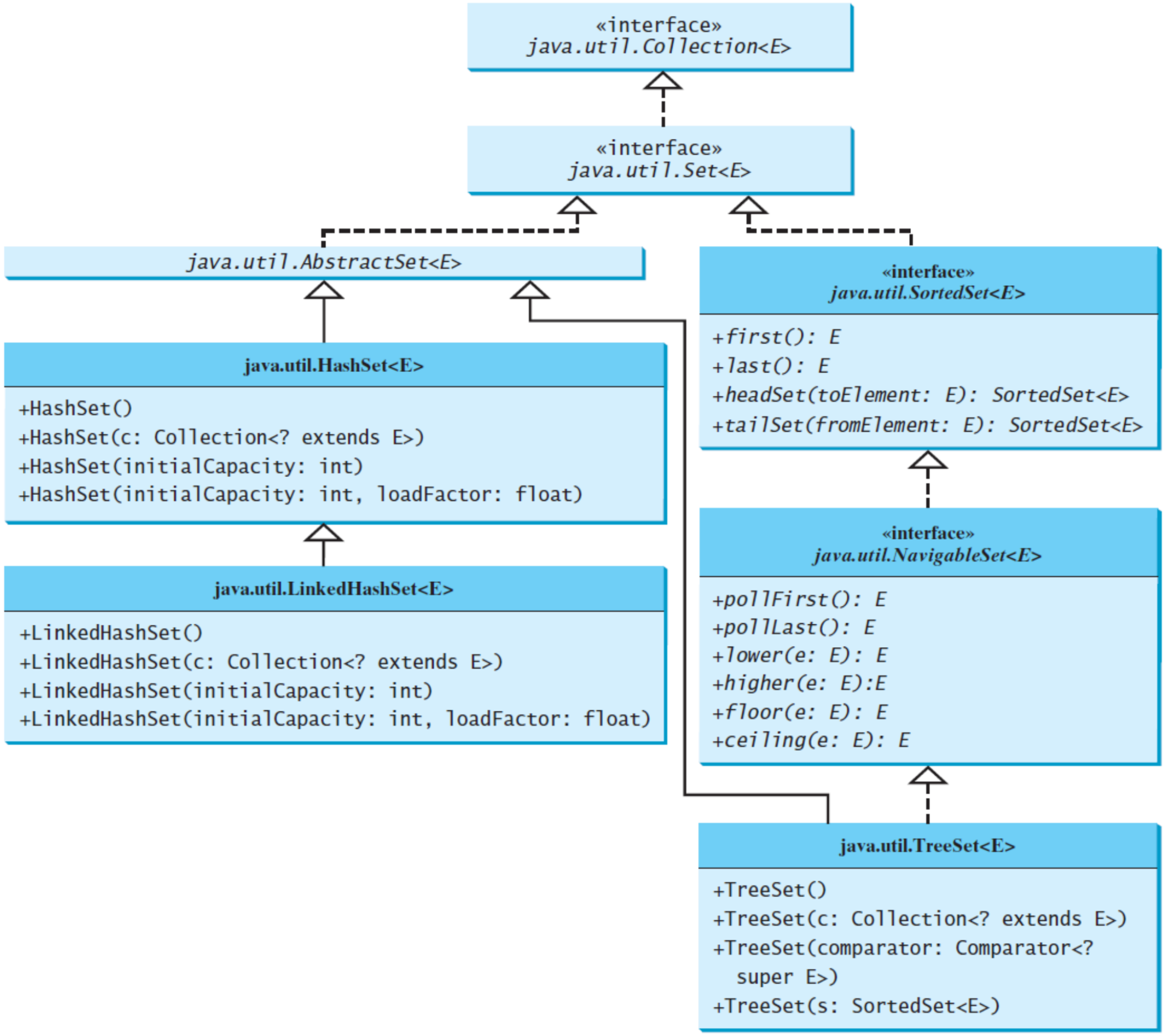
Sets

- The **`Set`** interface simply states that an instance contains no duplicates

- No two elements **`e1`** and **`e2`** can be in the set such that **`e1.equals(e2)`** is true

- Key implementation question: do you need an ordering when iterating elements?
  - If so, what kind?

Set Hierarchy



«interface»
java.util.Collection<E>

«interface»
java.util.Set<E>

java.util.AbstractSet<E>

java.util.HashSet<E>

+HashSet()
+HashSet(c: Collection<? extends E>)
+HashSet(initialCapacity: int)
+HashSet(initialCapacity: int, loadFactor: float)

java.util.LinkedHashSet<E>

+LinkedHashSet()
+LinkedHashSet(c: Collection<? extends E>)
+LinkedHashSet(initialCapacity: int)
+LinkedHashSet(initialCapacity: int, loadFactor: float)

«interface»
java.util.SortedSet<E>

+first(): E
+last(): E
+headSet(toElement: E): SortedSet<E>
+tailSet(fromElement: E): SortedSet<E>

«interface»
java.util.NavigableSet<E>

+pollFirst(): E
+pollLast(): E
+lower(e: E): E
+higher(e: E):E
+floor(e: E): E
+ceiling(e: E): E

java.util.TreeSet<E>

+TreeSet()
+TreeSet(c: Collection<? extends E>)
+TreeSet(comparator: Comparator<? super E>)
+TreeSet(s: SortedSet<E>)

HashSet

- The **HashSet** class guarantees a duplicate-free collection, but makes NO guarantees as to iteration order
  – In particular, the order may not remain constant over time

- Time for basic operations (add, remove, contains, size) does NOT depend on the number of elements (i.e. very fast)!

Example

```
public static void p(Set<?> s) {
     for (Object o : s) {
          System.out.printf("%s ", o);
     }
     System.out.printf("%n");
}

public static void main(String[] args) {
     final Set<String> s = new HashSet<>();
     s.add("alpha");
     p(s);
     s.add("beta");
     p(s);
     s.add("gamma");
     p(s);
     s.add("delta");
     p(s);
     s.add("alpha");
     p(s);
}
```

```
alpha
alpha beta
alpha beta gamma
alpha delta beta gamma
alpha delta beta gamma
```

`LinkedHashSet`

- If you need a *predictable* ordering, the **`LinkedHashSet`** internally maintains a linked list of entries

- Iteration over entries reflects the order in which they were inserted into the set ("insertion order")

Example

```
public static void p(Set<?> s) {
     for (Object o : s) {
          System.out.printf("%s ", o);
     }
     System.out.printf("%n");
}

public static void main(String[] args) {
     final Set<String> s = new LinkedHashSet<>();
     s.add("alpha");
     p(s);
     s.add("beta");
     p(s);
     s.add("gamma");
     p(s);
     s.add("delta");
     p(s);
     s.add("alpha");
     p(s);
}
```

```
alpha
alpha beta
alpha beta gamma
alpha beta gamma delta
alpha beta gamma delta
```

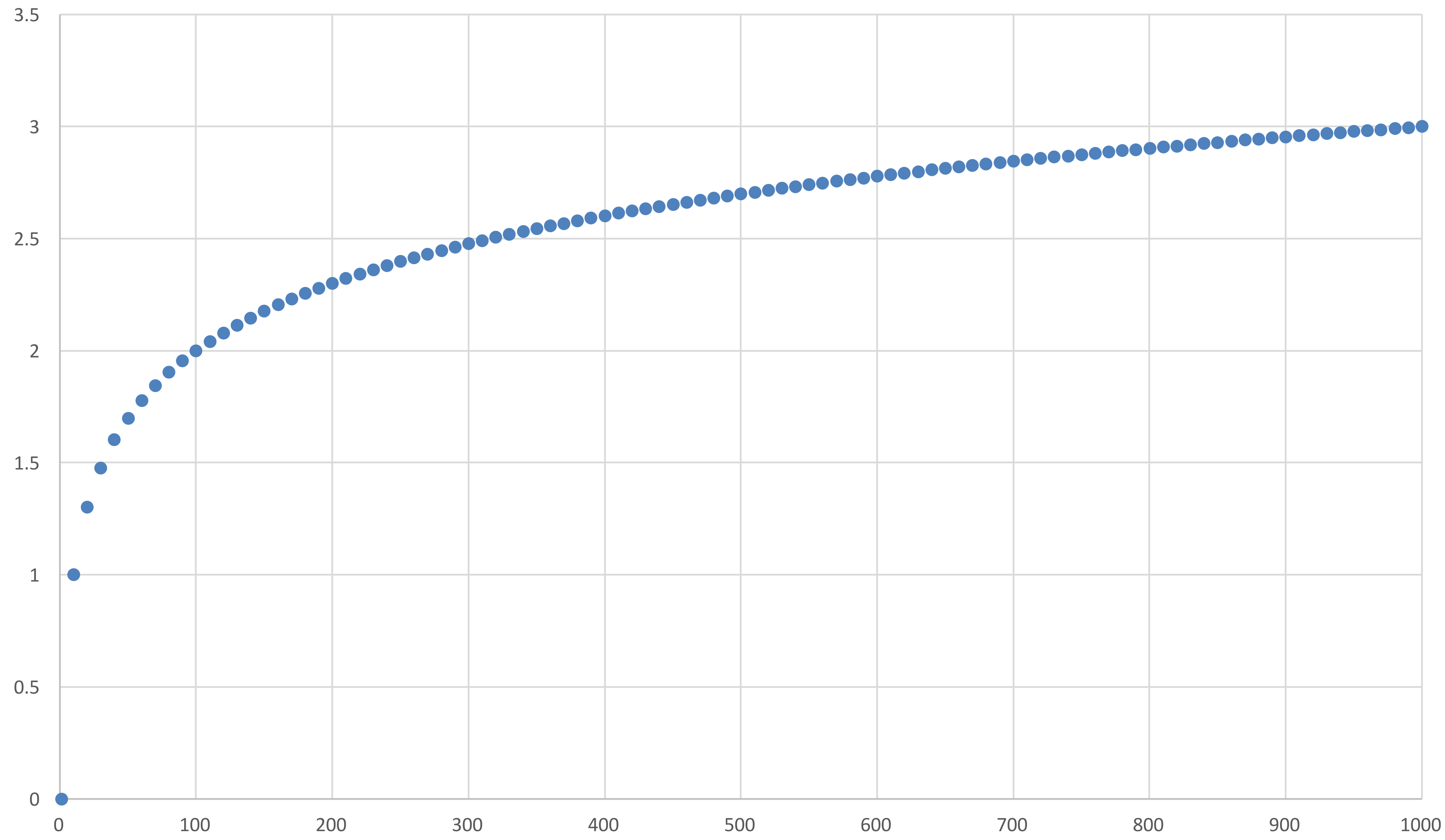
`TreeSet`

- Use a **`TreeSet`** if you need to iterate over elements in a particular order

- Basic operations are slightly slower (*logarithmic* in number of elements), so don't use by default

- You can either use the "natural" ordering (i.e. **`Comparable`**) or supply your own ordering via a **`Comparator`**

Logarithmic Growth



Example

```
public static void p(Set<?> s) {
     for (Object o : s) {
          System.out.printf("%s ", o);
     }
     System.out.printf("%n");
}

public static void main(String[] args) {
     final Set<String> s = new TreeSet<>();
     s.add("alpha");
     p(s);
     s.add("beta");
     p(s);
     s.add("gamma");
     p(s);
     s.add("delta");
     p(s);
     s.add("alpha");
     p(s);
}
```

```
alpha
alpha beta
alpha beta gamma
alpha beta delta gamma
alpha beta delta gamma
```

The Comparator Interface

- Sometimes there are multiple ways in which to compare elements
- The **Comparator** interface allows you to express a way of comparing two elements of the same type via a single method (i.e. functional!)

```
public interface Comparator<T> {
   // negative if o1 < o2
   // 0 if o1 == o2
   // positive if o1 > o2
   int compare(T o1, T o2);
}
```

Example

```
public static void p(Set<?> s) {
     for (Object o : s) {
          System.out.printf("%s ", o);
     }
     System.out.printf("%n");
}

public static void main(String[] args) {
     final Set<String> s = new TreeSet<>((o1, o2)->-o1.compareTo(o2));
     s.add("alpha");
     p(s);
     s.add("beta");
     p(s);
     s.add("gamma");
     p(s);
     s.add("delta");
     p(s);
     s.add("alpha");
     p(s);
}
```

```
alpha
beta alpha
gamma beta alpha
gamma delta beta alpha
gamma delta beta alpha
```

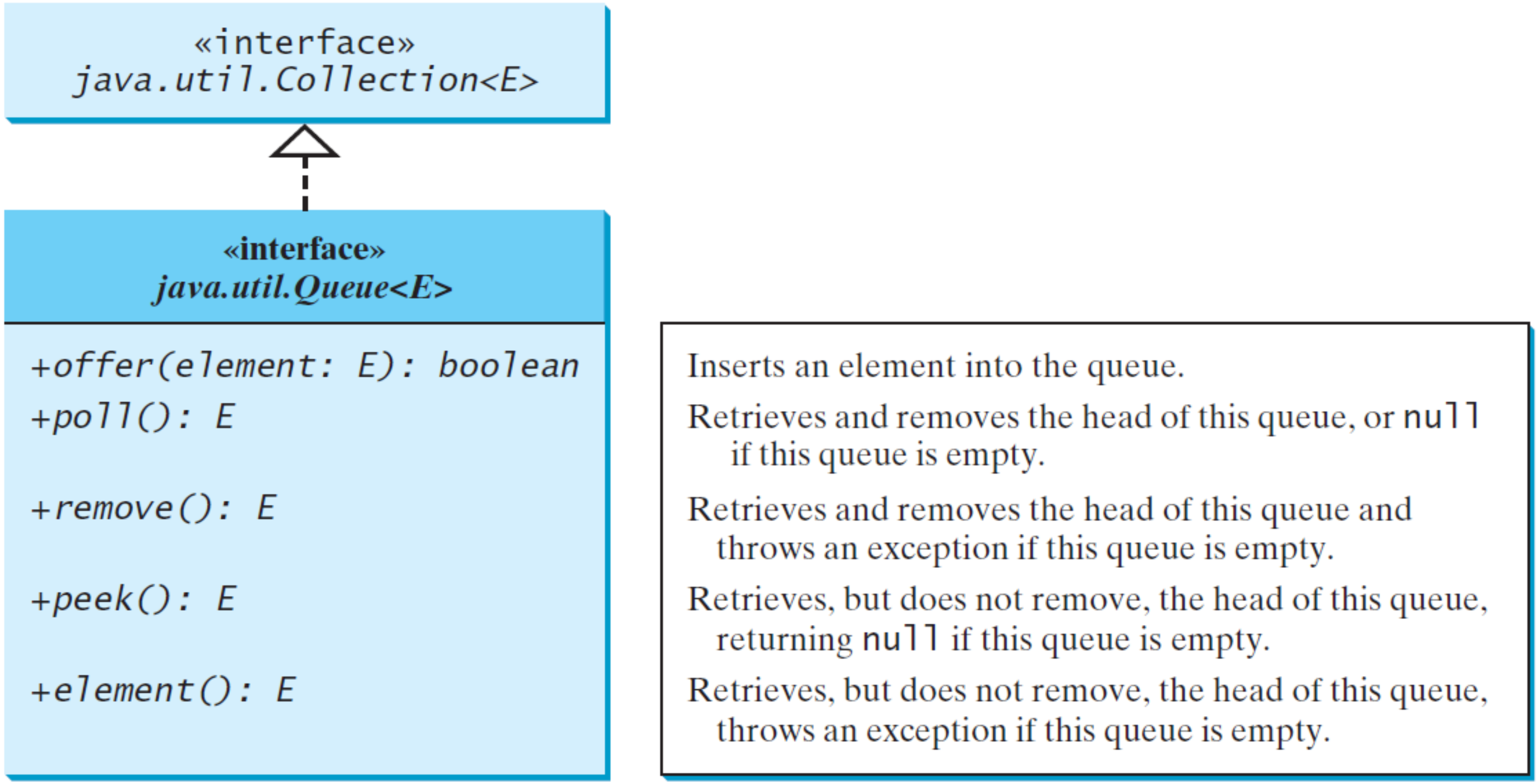
More Progress!

Name	What is it good for?	Implementations
`List`	A sequence; access via index, search	`ArrayList`, `Vector`, `LinkedList`
	+ push, pop, peek (i.e. ***LIFO***)	`Stack`
`Set`	Distinct objects (i.e. no duplicates)	`Linked/HashSet`, `TreeSet`
`Queue`	Hold elements in line (i.e. ***FIFO***)	`LinkedList`
	+ custom ordering	`PriorityQueue`
`Map`	Associate two objects	`HashMap`
	+ ordered iteration	`LinkedHashMap`, `TreeMap`

Queues

- The **`Queue`** interface represents a first-in/first-out (FIFO) data structure
  - Elements are appended to the end of the queue and are removed from the beginning
- The **`PriorityQueue`** interface represents a queue in which elements are assigned *priorities*
  - When accessing elements, the element with the highest priority is removed first
- Come in handy for storing items to process (e.g. work queues)

`Queue` Interface

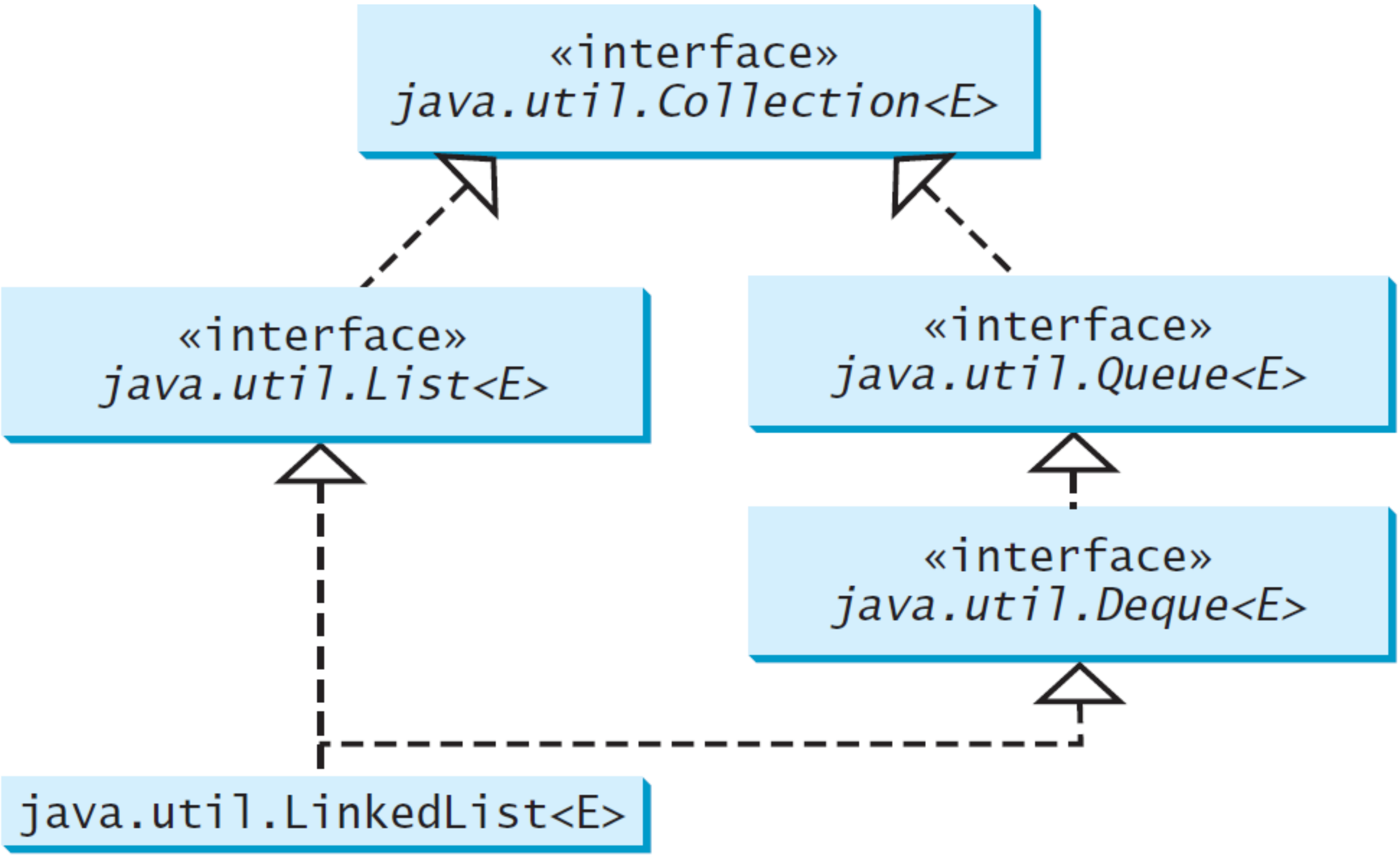


«interface»
java.util.Collection<E>

«interface»
java.util.Queue<E>

+offer(element: E): boolean	Inserts an element into the queue.
+poll(): E	Retrieves and removes the head of this queue, or `null` if this queue is empty.
+remove(): E	Retrieves and removes the head of this queue and throws an exception if this queue is empty.
+peek(): E	Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
+element(): E	Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

LinkedList as a Queue



Queue Example

```
private static class Passenger {
      final public String name;
      final public int zone;

      public Passenger(String name, int zone) {
            this.name = name;
            this.zone = zone;
      }

      @Override
      public String toString() {
            return String.format("%s (%d)",
                                        name, zone);
      }
}

public static void p(Queue<?> q) {
      while (!q.isEmpty()) {
            System.out.printf("%s%n", q.poll());

      }
}
```

```
public static void main(String[] args) {
      Queue<Passenger> q = new LinkedList<>();

      q.offer(new Passenger("Alice", 2));
      q.offer(new Passenger("Bob", 3));
      q.offer(new Passenger("Carol", 1));
      q.offer(new Passenger("Dan", 2));
      q.offer(new Passenger("Bob", 2));
      p(q);
}
```

```
Alice (2)
Bob (3)
Carol (1)
Dan (2)
Bob (2)
```

Priority Queue Example (1)

```
private static class Passenger {
	final public String name;
	final public int zone;

	public Passenger(String name, int zone) {
		this.name = name;
		this.zone = zone;
	}

	@Override
	public String toString() {
		return String.format("%s (%d)",
		                     name, zone);
	}
}

public static void p(Queue<?> q) {
	while (!q.isEmpty()) {
		System.out.printf("%s%n", q.poll());

	}
}
```

```
public static void main(String[] args) {
	Queue<Passenger> q = new PriorityQueue<>();

	q.offer(new Passenger("Alice", 2));
	q.offer(new Passenger("Bob", 3));
	q.offer(new Passenger("Carol", 1));
	q.offer(new Passenger("Dan", 2));
	q.offer(new Passenger("Bob", 2));
	p(q);
}
```

```
Exception in thread "main"
java.lang.ClassCastException:
Foo$Passenger cannot be cast
to java.lang.Comparable
```

Priority Queue Example (2)

```
private static class Passenger
        implements Comparable<Passenger> {
    final public String name;
    final public int zone;

    public Passenger(String name, int zone) {
        this.name = name;
        this.zone = zone;
    }

    @Override
    public String toString() {
        return String.format("%s (%d)",
                            name, zone);
    }

    @Override
    public int compareTo(Passenger o) {
        return Integer.compare(zone, o.zone);
    }
}

public static void p(Queue<?> q) {
    while (!q.isEmpty()) {
        System.out.printf("%s%n", q.poll());
    }
}
```

```
public static void main(String[] args) {
    Queue<Passenger> q = new PriorityQueue<>();

    q.offer(new Passenger("Alice", 2));
    q.offer(new Passenger("Bob", 3));
    q.offer(new Passenger("Carol", 1));
    q.offer(new Passenger("Dan", 2));
    q.offer(new Passenger("Bob", 2));
    p(q);
}
```

```
Carol (1)
Bob (2)
Dan (2)
Alice (2)
Bob (3)
```

Priority Queue Example (3)

```
private static class Passenger {
      final public String name;
      final public int zone;

      public Passenger(String name, int zone) {
            this.name = name;
            this.zone = zone;
      }

      @Override
      public String toString() {
            return String.format("%s (%d)",
                                    name, zone);
      }
}

public static void p(Queue<?> q) {
      while (!q.isEmpty()) {
            System.out.printf("%s%n", q.poll());

      }
}
```

```
public static void main(String[] args) {
      Queue<Passenger> q = new PriorityQueue<>(
         (e1,e2)->Integer.compare(e1.zone, e2.zone));

      q.offer(new Passenger("Alice", 2));
      q.offer(new Passenger("Bob", 3));
      q.offer(new Passenger("Carol", 1));
      q.offer(new Passenger("Dan", 2));
      q.offer(new Passenger("Bob", 2));
      p(q);
}
```

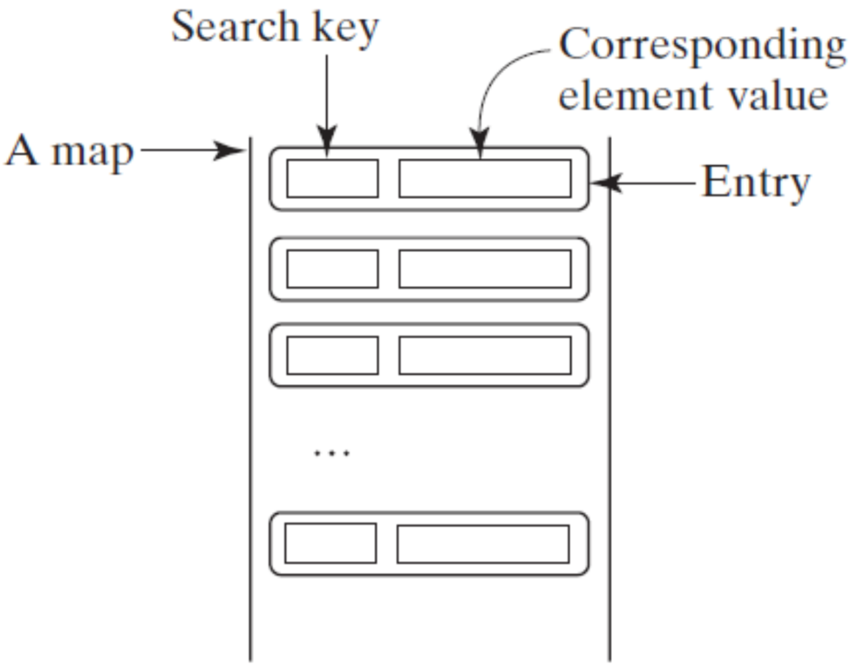
```
Carol (1)
Bob (2)
Dan (2)
Alice (2)
Bob (3)
```

Now with Even More Progress!

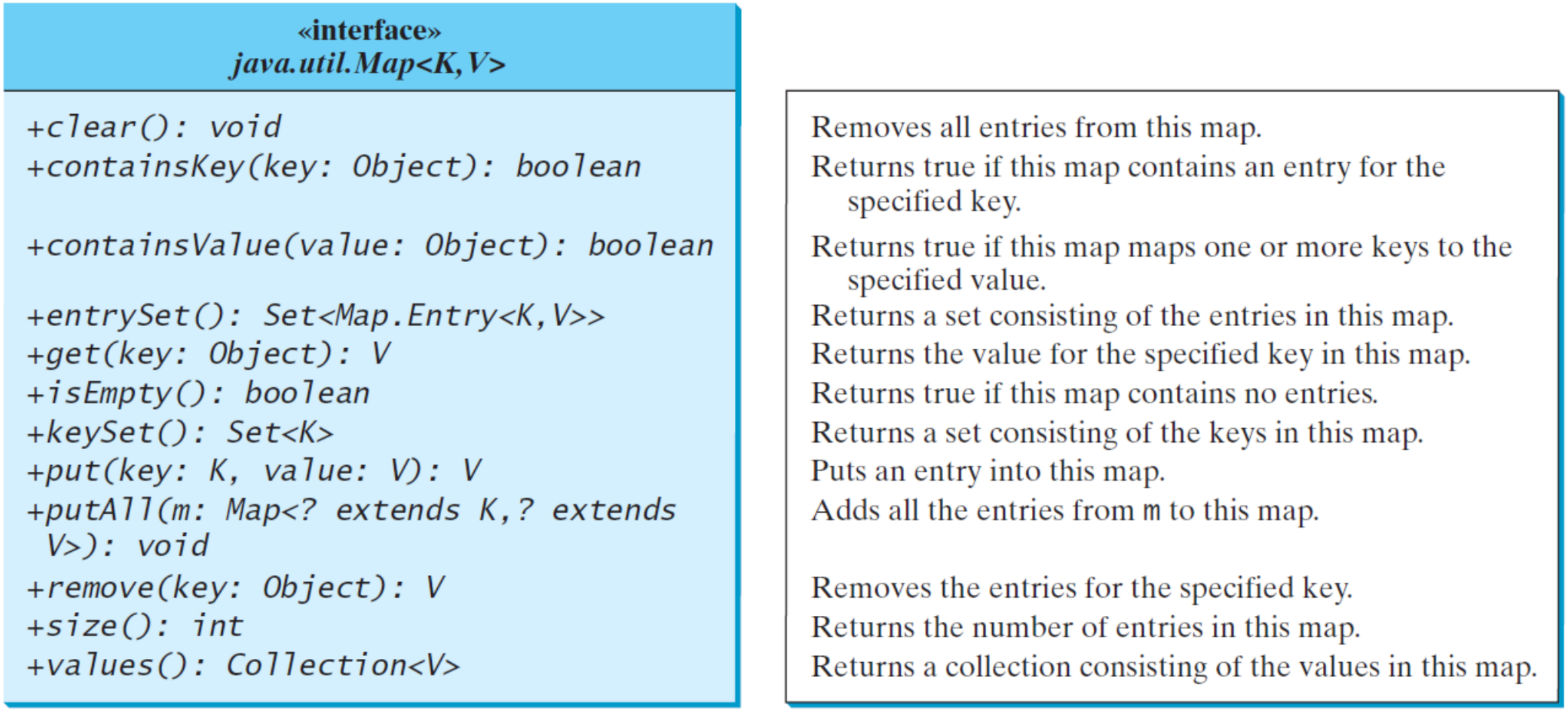
Name	What is it good for?	Implementations
`List`	A sequence; access via index, search	`ArrayList`, `Vector`, `LinkedList`
	+ push, pop, peek (i.e. ***LIFO***)	`Stack`
`Set`	Distinct objects (i.e. no duplicates)	`Linked/HashSet`, `TreeSet`
`Queue`	Hold elements in line (i.e. ***FIFO***)	`LinkedList`
	+ custom ordering	`PriorityQueue`
`Map`	Associate two objects	`HashMap`
	+ ordered iteration	`LinkedHashMap`, `TreeMap`

What are Maps?

- Think of a map as an array… but where the index can be anything!
- Technically a map is a set of **`Entry`** elements, each with a "key" and a "value"
- Each key must be unique – this can be used to get/set the corresponding value

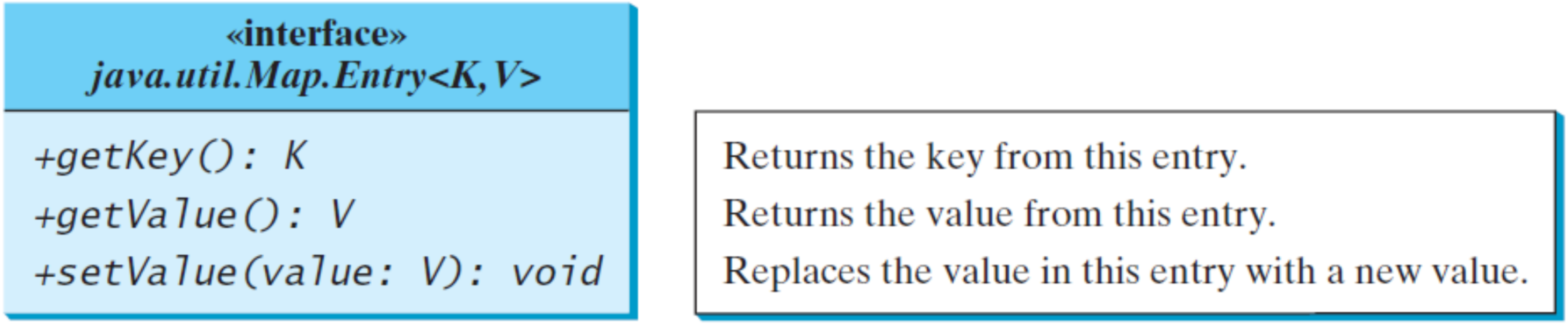


The `Map` Interface



«interface» *java.util.Map<K,V>*	
`+clear(): void`	Removes all entries from this map.
`+containsKey(key: Object): boolean`	Returns true if this map contains an entry for the specified key.
`+containsValue(value: Object): boolean`	Returns true if this map maps one or more keys to the specified value.
`+entrySet(): Set<Map.Entry<K,V>>`	Returns a set consisting of the entries in this map.
`+get(key: Object): V`	Returns the value for the specified key in this map.
`+isEmpty(): boolean`	Returns true if this map contains no entries.
`+keySet(): Set<K>`	Returns a set consisting of the keys in this map.
`+put(key: K, value: V): V`	Puts an entry into this map.
`+putAll(m: Map<? extends K,? extends V>): void`	Adds all the entries from `m` to this map.
`+remove(key: Object): V`	Removes the entries for the specified key.
`+size(): int`	Returns the number of entries in this map.
`+values(): Collection<V>`	Returns a collection consisting of the values in this map.

The `Entry` Interface



«interface»
java.util.Map.Entry<K,V>

```
+getKey(): K
+getValue(): V
+setValue(value: V): void
```

Returns the key from this entry.
Returns the value from this entry.
Replaces the value in this entry with a new value.

Key Ordering

As with sets, the primary difference in concrete implementations comes down to key ordering

- **`HashMap`**: fastest, no key ordering
- **`LinkedHashMap`**: very fast, insertion order
- **`TreeMap`**: fast, order via **`Comparable`**/**`Comparator`**

Example (1)

```
public static <K, V> void inMap(Map<K,V> m, K key) {
        System.out.printf("%s: %s%n", key,
                m.containsKey(key)?m.get(key):"not in map");
}

public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<>();
        m.put("Mount Everest", 29029);
        m.put("K2", 28251);
        m.put("Kangchenjunga", 28169);
        m.put("Lhotse", 27940);
        m.put("Makalu", 27838);

        inMap(m, "K2");
        inMap(m, "Manaslu");
        System.out.printf("%nContents:%n");
        for (Entry<String, Integer> e : m.entrySet()) {
                System.out.printf(" %s=%d%n",
                        e.getKey(), e.getValue());
        }
}
```

```
K2: 28251
Manaslu: not in map

Contents:
 K2=28251
 Mount Everest=29029
 Kangchenjunga=28169
 Lhotse=27940
 Makalu=27838
```

Example (2)

```java
public static <K, V> void inMap(Map<K,V> m, K key) {
      System.out.printf("%s: %s%n", key,
            m.containsKey(key)?m.get(key):"not in map");
}

public static void main(String[] args) {
      Map<String, Integer> m = new LinkedHashMap<>();
      m.put("Mount Everest", 29029);
      m.put("K2", 28251);
      m.put("Kangchenjunga", 28169);
      m.put("Lhotse", 27940);
      m.put("Makalu", 27838);

      inMap(m, "K2");
      inMap(m, "Manaslu");
      System.out.printf("%nContents:%n");
      for (Entry<String, Integer> e : m.entrySet()) {
            System.out.printf(" %s=%d%n",
                  e.getKey(), e.getValue());
      }
}
```

```
K2: 28251
Manaslu: not in map

Contents:
 Mount Everest=29029
 K2=28251
 Kangchenjunga=28169
 Lhotse=27940
 Makalu=27838
```

Example (3)

```
public static <K, V> void inMap(Map<K,V> m, K key) {
      System.out.printf("%s: %s%n", key,
            m.containsKey(key)?m.get(key):"not in map");
}

public static void main(String[] args) {
      Map<String, Integer> m = new TreeMap<>();
      m.put("Mount Everest", 29029);
      m.put("K2", 28251);
      m.put("Kangchenjunga", 28169);
      m.put("Lhotse", 27940);
      m.put("Makalu", 27838);

      inMap(m, "K2");
      inMap(m, "Manaslu");
      System.out.printf("%nContents:%n");
      for (Entry<String, Integer> e : m.entrySet()) {
            System.out.printf(" %s=%d%n",
                  e.getKey(), e.getValue());
      }
}
```

```
K2: 28251
Manaslu: not in map

Contents:
 K2=28251
 Kangchenjunga=28169
 Lhotse=27940
 Makalu=27838
 Mount Everest=29029
```

Example (4)

```
public static <K, V> void inMap(Map<K,V> m, K key) {
      System.out.printf("%s: %s%n", key,
            m.containsKey(key)?m.get(key):"not in map");
}

public static void main(String[] args) {
      Map<String, Integer> m = new TreeMap<>(
                                    (k1,k2)->-k1.compareTo(k2));
      m.put("Mount Everest", 29029);
      m.put("K2", 28251);
      m.put("Kangchenjunga", 28169);
      m.put("Lhotse", 27940);
      m.put("Makalu", 27838);

      inMap(m, "K2");
      inMap(m, "Manaslu");
      System.out.printf("%nContents:%n");
      for (Entry<String, Integer> e : m.entrySet()) {
            System.out.printf(" %s=%d%n",
                  e.getKey(), e.getValue());
      }
}
```

```
K2: 28251
Manaslu: not in map

Contents:
 Mount Everest=29029
 Makalu=27838
 Lhotse=27940
 Kangchenjunga=28169
 K2=28251
```

Exercise

- Use a **`LinkedHashMap`** to create a menu with the following items, care of Bertie Bott
  - Booger: $1
  - Rotten Egg: $1.50
  - Vomit: $1
  - Cherry: $2
- Write a method that updates the map to increase the price of all items by $0.50

Solution

```
public static void p(Map<?,? extends Number> m) {
      for (Entry<?, ? extends Number> e : m.entrySet()) {
            System.out.printf("%s=%.2f%n", e.getKey(), e.getValue().doubleValue());
      }
      System.out.printf("%n");
}

public static <K> void inc(Map<K, ? super Double> m, double amt) {
      for (K k : m.keySet()) {
            m.replace(k, (double) m.get(k) + amt);
      }
}

public static void main(String[] args) {
      Map<String, Double> m = new LinkedHashMap<>();

      m.put("Booger", 1.);
      m.put("Rotten Egg", 1.5);
      m.put("Vomit", 1.);
      m.put("Cherry", 2.);

      p(m);
      inc(m, 0.5);
      p(m);
}
```

```
Booger=1.00
Rotten Egg=1.50
Vomit=1.00
Cherry=2.00

Booger=1.50
Rotten Egg=2.00
Vomit=1.50
Cherry=2.50
```

Take Home Points

Name	What is it good for*?	Implementations
`List`	A sequence; access via index, search	`ArrayList`, `Vector`, `LinkedList`
	+ push, pop, peek (i.e. ***LIFO***)	`Stack`
`Set`	Distinct objects (i.e. no duplicates)	`Linked/HashSet`, `TreeSet`
`Queue`	Hold elements in line (i.e. ***FIFO***)	`LinkedList`
	+ custom ordering	`PriorityQueue`
`Map`	Associate two objects	`HashMap`
	+ ordered iteration	`LinkedHashMap`, `TreeMap`

- The JCF has many useful data structures
- Choosing the correct interface and concrete implementation requires you to understand your data, the important operations, and efficiency tradeoffs