Collections (JCF)

Application infrastructure of the Java Collections

- Application infrastructure encompassing abstract and concrete classes, interfaces, and algorithms that provide various types of collections in Java

Java Collections Framework (JCF)

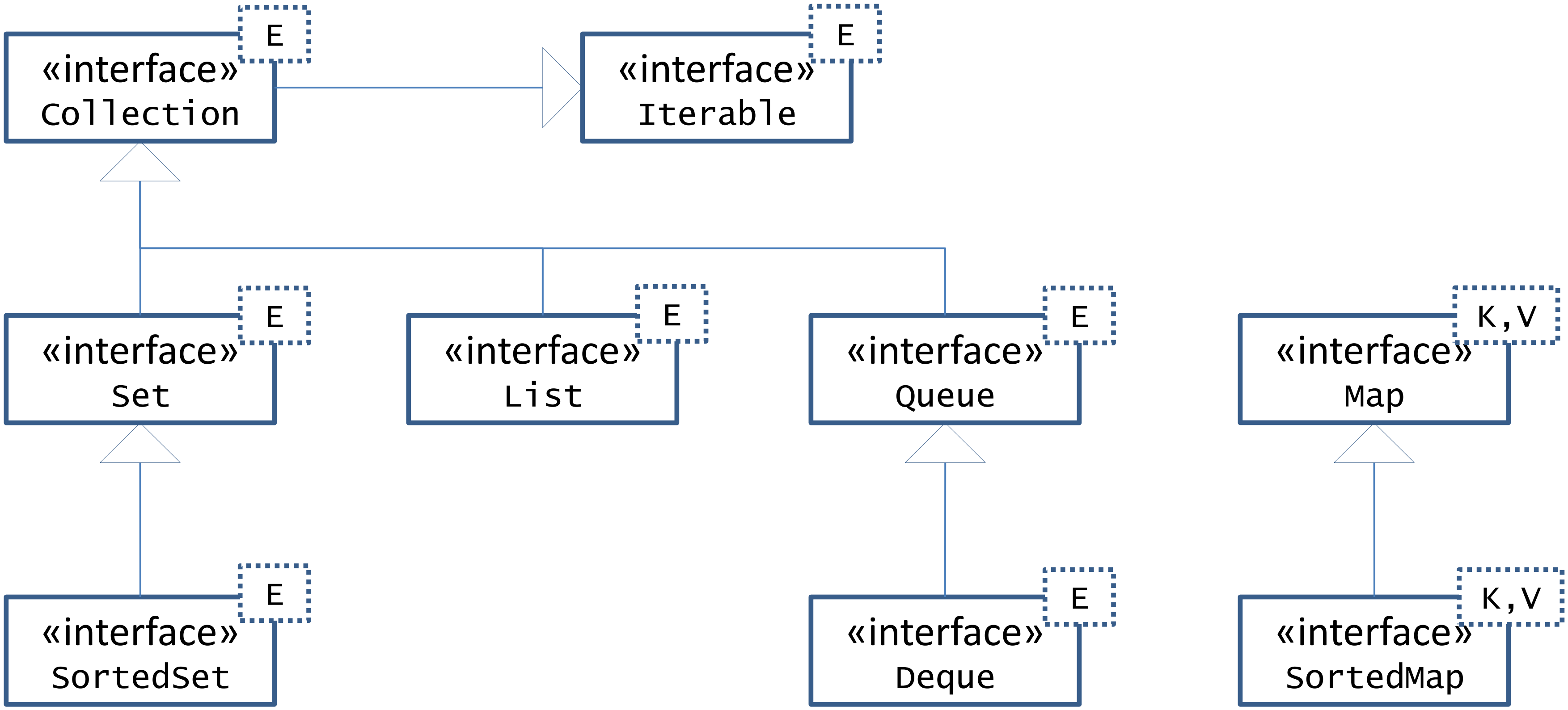
- Collections
  - Aggregate of structured elements
  - each type of collection has specific properties
  - All have different efficiencies to perform equivalent operations

JCF: tipos de colecção

Type	Abstract type	Repetitions	Ordered	Order
`Set<E>`	Set	no	?	?
`List<E>`	Sequence	yes	yes	insertion
`Queue<E>`	Queue	yes	yes	extraction: yes, internaly: ?
`Stack<E>`	Stack	yes	yes	extraction: yes, internaly: ?
`Map<K,V>`	Map	não (keys) yes (values)	?	?

Legend:
`E` – type of elements
`K` – type of map keys
`V` – type of map values
? – dependent of the concrete type

JCF: main interfaces



JFC: The data-structures

Name	Description
Array	Sequence of contiguous elements in memory, fast indexation but slow insertion in the middle and capacity increase.
Linked list	Sequence of linked elements, slow indexation and search, but quick insertions.
Tree	Sequence of elements organized in a tree-structure, all basic operations are reasonably fast (requires element ordering).
Hash(ing) table	Elements spread in a large matrix using indexes that are a function of the element's value. All essential operations are fast (more memory used).

JCF: elements, keys and values

- Must implement
  - `boolean equals(Object another)`
  - `int hashCode()`

To search

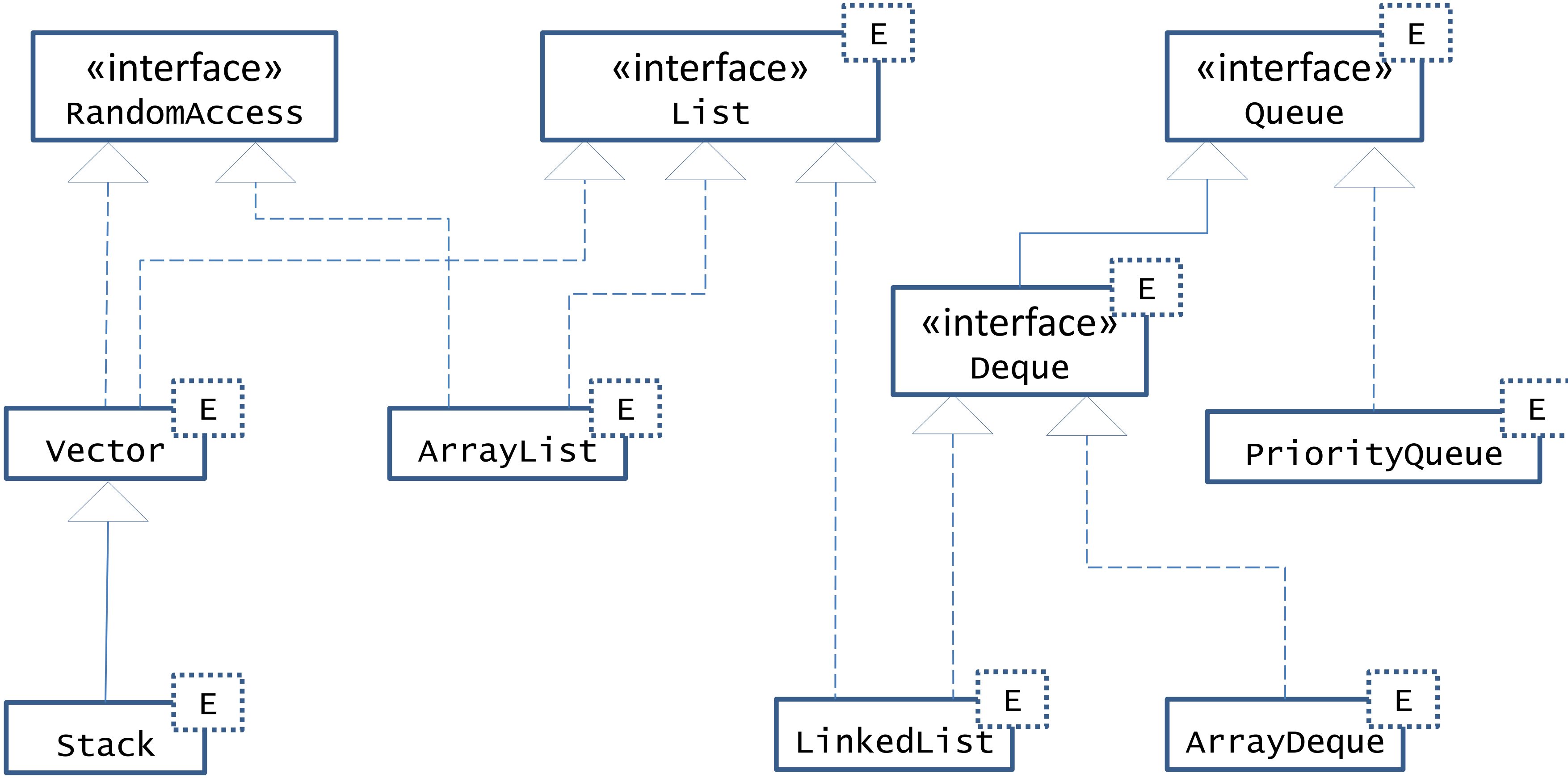
For hash maps

- Operations supplied by `Object`!
- Can be overloades (*with great care*)
  - If
    ```
    one.equals(another)
    ```
    then
    ```
    one.hashCode() == another.hashCode()
    ```
  - Other restrictions may need to be considered ...

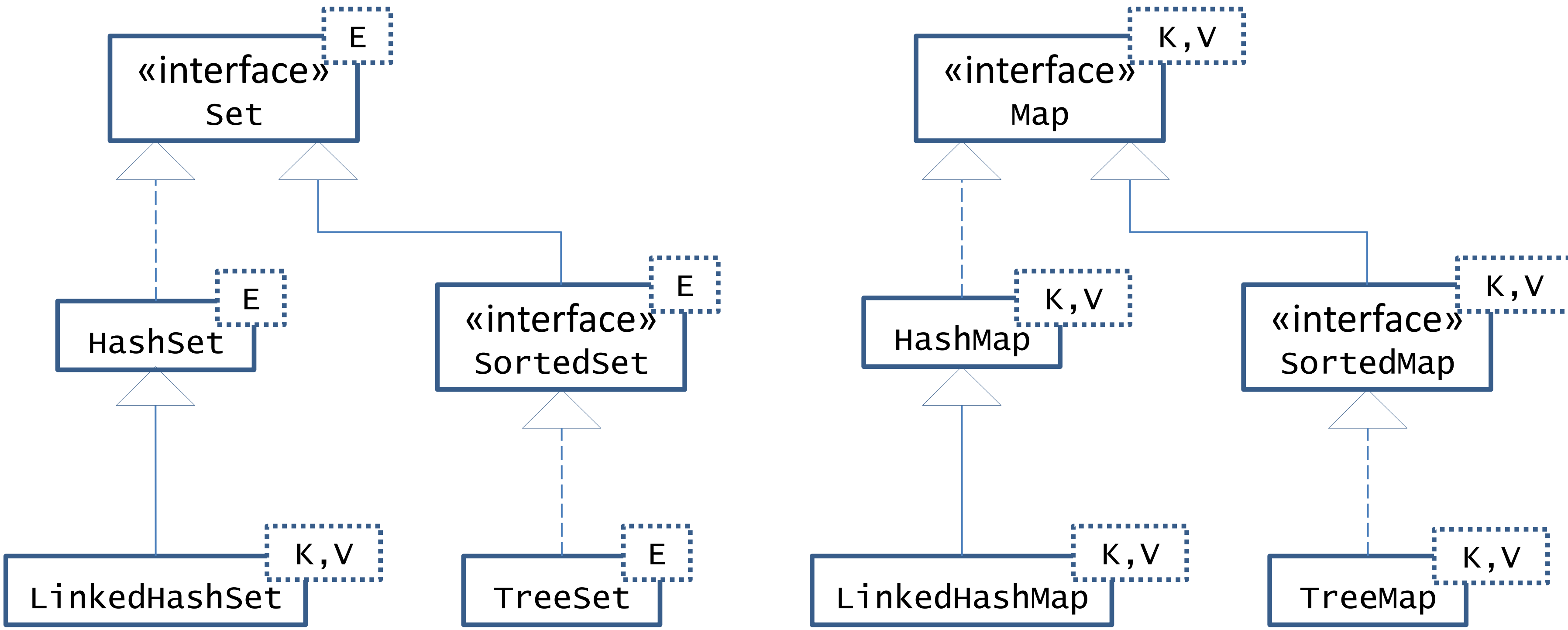
JCF: concrete classes

Type	Internal representation	Restrictions
`ArrayList<E>`	Vector	-
`Vector<E>`	Vector	-
`LinkedList<E>`	Linked List	-
`ArrayDeque<E>`	Vector	-
`Stack<E>`	Vector (via `Vector<E>`)	-
`PriorityQueue<E>`	Vector (organized as a tree)	`E` implements `Comparable<E>`
`TreeSet<E>`	Tree	`E` implements `Comparable<E>`
`TreeMap<K,V>`	Tree	`K` implements `Comparable<K>`
`HashSet<E>`	Hash map	-
`HashMap<K,V>`	Hash map	-

JCF: concrete classes



JCF: concrete classes



JCF: `one.compareTo(another)`

Relation between `one` and `another`	**Result**
`one` < `another`	< 0
`one` = `another`	= 0
`one` > `another`	> 0

JCF: Good practices

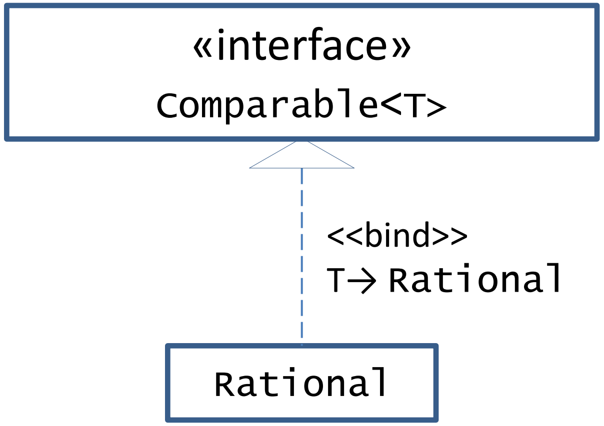
- Class implements `compareTo`? Its a *value-type*
- ... so, *it should overload* `equals`...
- ... because `equals` default behavior compares *identity* not *equality*!
- Operations `compareTo` and `equals` must be consistent ...
- ...i.e., `one.compareTo(another) == 0` must be the same as `one.equals(another)`

The Rational example

```
public class Rational implements Comparable<Rational> {
    private final int numerator;
    private final int denominator;
    …

    public int compareTo(final Rational another){
        return getNumerator() * another.getDenominator()
             - another.getNumerator() * getDenominator();
    }

    …
}
```



This implementation requires the denominator to be positive.

That should be garantied by a class invariant.

The `Rational` example

```
public class Rational implements Comparable<Rational> {
    …

    public boolean equals(final Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;

        final Rational other = (Rational) obj;

        return denominator == other.denominator
            && numerator == other.numerator;
    }

    …
}
```

The `Rational` example

```
public class Rational implements Comparable<Rational> {
    private final int numerator;
    private final int denominator;
    …

    public int hashCode() {
        return (getNumerator() + getDenominator())
            * (getNumerator() + getDenominator() + 1)
            + getDenominator();
    }

    …
}
```

The `Student` example

Student does not have to be ordered the same way all the time (it does not have a natural order)

For an alphabetical order one can define:

```
public class ComparadorDeAlunos implements Comparator<Aluno> {

    public int compare(Aluno aluno1, Aluno aluno2) {
        return aluno1.getNome().compareTo(aluno2.getNome());
    }

}
```

Collections

```
List<Rational> racionais = new ArrayList<Rational>();
...
Collections.sort(racionais);
```

sort(List), Order according natural order (**Comparable**)

```
List<Aluno> alunos = new LinkedList<Aluno>();
...
Collections.sort(alunos, new  ComparadorDeAlunos());
```

sort(List, Comparator) , Order according to criteria

Collections has more useful methods, such as **shuffle(List)** , **reverse(List)**, **min(Collection)**, **max(Collection)**

JCF: `List` and `ArrayList`

```
 List<Course> courses =
    new ArrayList<Course>();
Course ip = new Course("IP");
Course poo = new Course("POO");
courses.add(ip);  // adiciona ao fim
courses.add(poo);

int indexOfCourseToRemove = -1;
for (int i = 0; i != courses.size(); i++)
    if (courses.get(i) == poo)
        indexOfCourseToRemove = i;

if (indexOfCourseToRemove != -1)
    courses.remove(indexOfCourseToRemove);
courses.remove(ip);
```

It is common to use a more generic type to use a colection. this way one can keep th flexibility in changing the implementation by changing just one line of code.

Is is sensible to index a list? What if this is a `LinkedList`?

Removing elements outside the cicle? O.K.
Removing within the cicle? Not a good idea!

JCF: Vector

```
Vector<Course> courses = new Vector<Course>();

Course ip = new Course("IP");
Course poo = new Course("POO");

courses.add(ip);   // add in the end
courses.add(poo);

for (int i = 0; i != courses.size(); i++)
    out.println(courses.get(i));
```

JCF: Stack

```
Stack<Course> courses = new Stack<Course>();

Course ip = new Course("IP");
Course poo = new Course("POO");

courses.push(ip);  // add on top
courses.push(poo);

while (!courses.isEmpty()) {
    out.println(courses.peek());
    courses.pop();
}
```

JCF: `List`, `LinkedList` and `Iterator`

```
List<Course> courses =
    new LinkedList<Course>();
…
Course esi = new Course("ES I");
…

Iterator<Course> iterator =
    courses.iterator();

while (iterator.hasNext()) {
    Course course = iterator.next();
    if (course == esi)
        iterator.remove();
}
```

When possible use the interface

Two in one: return and advance, arguably a good idea.

Safe removal, last element return by `next()` is removed.

JCF: Queue and LinkedList

```
Queue<String> courseNames =
    new LinkedList<String>();

courseNames.add("POO");
courseNames.add("ES I");
courseNames.add("IP");

while(!courseNames.isEmpty()) {
    out.println(courseNames.element());
    courseNames.remove();
}
```

JCF: Queue and LinkedList

```
Queue<Course> courses = new LinkedList<Course>();

Course ip = new Course("IP");
Course poo = new Course("POO");

courses.add(ip);  // adiciona ao início
courses.add(poo);  // adiciona ao início

out.println(courses.element());
out.println(courses.element());

Iterator<Course> iterator = courses.iterator();

while (iterator.hasNext()) {
    Course course = iterator.next();
    out.println(course);
}
```

JCF: LinkedList and Deque

```
Deque<Course> courses = new LinkedList<Course>();

Course ip = new Course("IP");
Course poo = new Course("POO");

courses.addFirst(ip); // adiciona ao início
courses.addLast(poo); // adiciona ao fim

out.println(courses.getFirst());
out.println(courses.getLast());

Iterator<Course> iterator = courses.iterator();

while (iterator.hasNext()) {
    Course course = iterator.next();
    out.println(course);
}
```

for-each

```
List<Course> courses =
    new LinkedList<Course>();

for (Course course : courses)
    out.println(course);
```

Compact iteration mode, but ... collection cannot be altered, harder to cicle over subsequences, etc..

JCF: Iteration and modififcation

```
List<Course> courses =
    new LinkedList<Course>();
…
Course poo = new Course("POO");
…

for (Course course : courses) {
    courses.remove(poo);
    out.println(course);
}
```

Changing the collection in mid-cicle can have unexpected effects, usually `ConcurrentModificationException` is thrown.

JCF: Map and HashMap

```
Map<String, Course> courses =
    new HashMap<String, Course>();
…
courses.put("IP", new Course("Introdução à …"));
…

if (courses.containsKey("IP"))
    out.println(courses.get("IP"));
for (String key : courses.keySet())
    out.println(key);
for (Map.Entry<String, Course> entry : courses.entrySet())
    out.println(entry);

for (Course course : courses.values())
    out.println(course);
```

JCF: Map and TreeMap

```
Map<String, Course> courses =
    new TreeMap<String, Course>();
…
courses.put("IP", new Course("Introdução à …"));
…

if (courses.containsKey("IP"))
    out.println(courses.get("IP"));
for (String key : courses.keySet())
    out.println(key);
for (Map.Entry<String, Course> entry : courses.entrySet())
    out.println(entry);

for (Course course : courses.values())
    out.println(course);
```

JCF: Queue and PriorityQueue

```
Queue<String> courseNames =
    new PriorityQueue<String>();

courseNames.add("POO");
courseNames.add("ES I");
courseNames.add("IP");

while(!courseNames.isEmpty()) {
    out.println(courseNames.element());
    courseNames.remove();
}
```

JCF: Good practices

- Never use collections of `Object`
- Pick the most suitable type for your intended usage
- Check the efficiency of the different operations in each concrete class
- Don’t modify a collection while cycling through it, unless when using iterator

JCF: Good practices

- Changing elements of a collection that relies on the intrinsic order of elements may have unexpected results
- Always use value-types when intrinsic order is required
- Check the documentation
- Not all collections allow `null` elements

References

- Y. Daniel Liang, *Introduction to Java Programming*, 7.ª edição, Prentice-Hall, 2008.

Summary

- Collections(JCF)