Colecções (JCF)

Infra-estrutura aplicacional de colecções do Java

- Infra-estrutura aplicacional englobando interfaces, classes abstractas e concretas, e algoritmos que disponibilizam vários tipos de colecção em Java

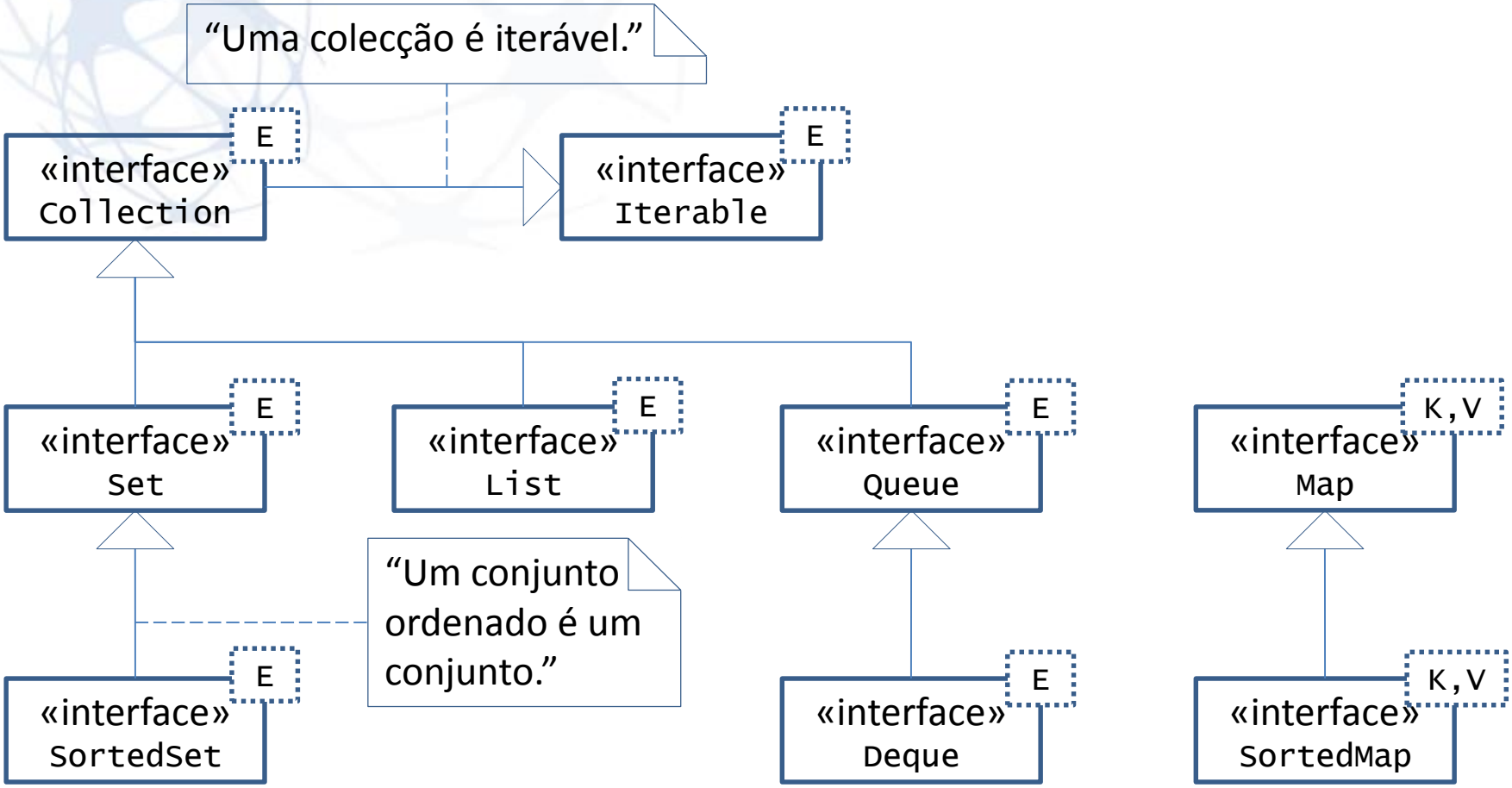
- Colecções
  - Agregados estruturados de elementos
  - Cada tipo de colecção tem propriedades específicas
  - Têm diferentes eficiências a realizar operações equivalentes

JCF: tipos de colecção

Tipo	Natureza	Repetições	Ordenado	Tipo de ordem
`Set<E>`	Conjunto	não	?	?
`List<E>`	Sequência	sim	sim	de inserção
`Queue<E>`	Fila de espera	sim	sim	extração: sim, internamente: ?
`Stack<E>`	Pilha	sim	sim	extração: sim, internamente: ?
`Map<K,V>`	Mapeia chaves em valores	não (chaves) sim (valores)	?	?

Legenda:
`E` – tipo dos elementos
`K` – tipo das chaves de um mapa
`V` – tipo dos valores de um mapa
? – característica depende do tipo concreto

JCF: principais interfaces



JFC: estruturas de dados subjacentes

Nome	Nome (inglês)	Descrição
Vector	*Array*	Sequência de elementos contíguos em memória, com indexação muito rápida mas inserção de novos elementos lenta (excepto nos extremos quando não é necessário um aumento da capacidade).
Lista ligada	*Linked list*	Sequência de elementos ligados, com indexação e pesquisa lentos mas inserção rápida em qualquer local.
Árvore	*Tree*	Sequência de elementos organizados em árvore, com todas as operações essenciais razoavelmente rápidas.
Tabela de dispersão	*Hash(ing) table*	Elementos espalhados em matriz usando índices obtidos aplicando-lhes uma função de endereçamento, com todas as operações essenciais muito rápidas (troca mais velocidade por maior consumo de memória).

JCF: elementos, chaves e valores

- Têm de implementar
  - `boolean equals(Object another)`
  - `int hashCode()`

Para procurar.

Para tabelas de dispersão.

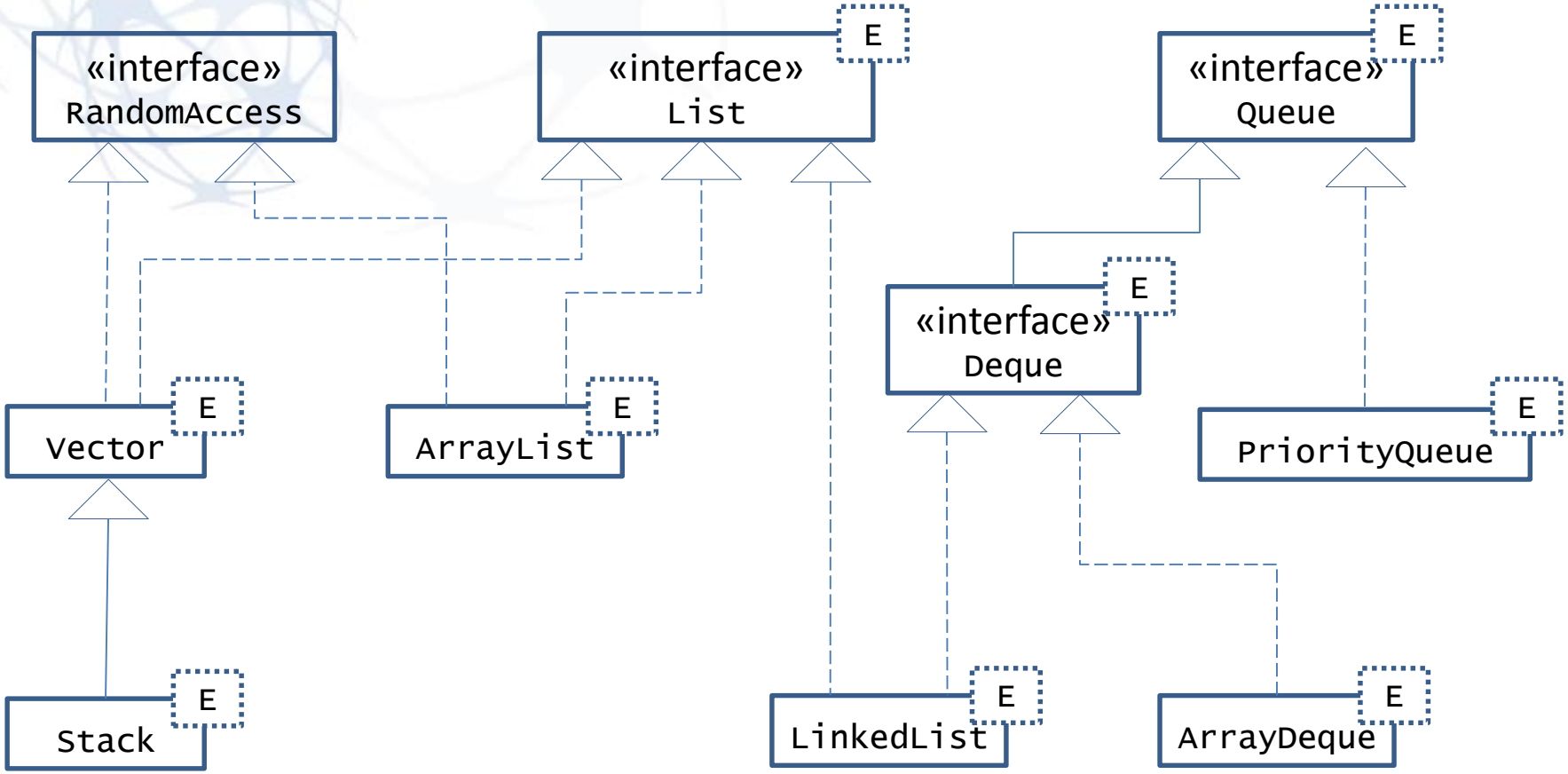
- Operações são fornecidas pela classe `Object`!

- Podem ser sobrepostas (*com cuidado*)
  - Se
    ```
    one.equals(another)
    ```
    então
    ```
    one.hashCode() == another.hashCode()
    ```
  - Outras restrições

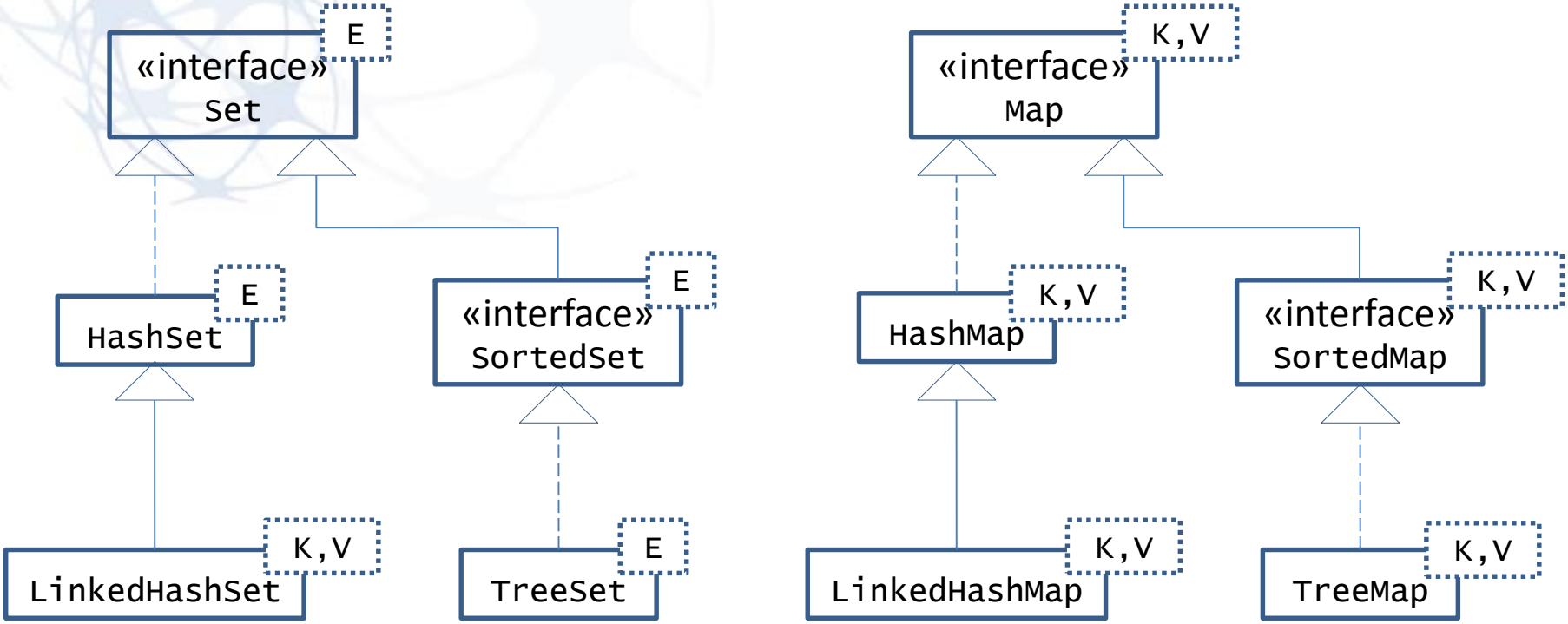
JCF: classes concretas

Tipo	Representação interna	Restrições adicionais
`ArrayList<E>`	Vector	-
`Vector<E>`	Vector	-
`LinkedList<E>`	Lista ligada	-
`ArrayDeque<E>`	Vector	-
`Stack<E>`	Vector (via `Vector<E>`)	-
`PriorityQueue<E>`	Vector (organizada como árvore)	`E` implementa `Comparable<E>`
`TreeSet<E>`	Árvore	`E` implementa `Comparable<E>`
`TreeMap<K,V>`	Árvore	`K` implementa `Comparable<K>`
`HashSet<E>`	Tabela de dispersão	-
`HashMap<K,V>`	Tabela de dispersão	-

JCF: classes concretas



JCF: classes concretas



JCF: `one.compareTo(another)`

Relação entre `one` e `another`	Resultado da operação
`one < another`	< 0
`one = another`	= 0
`one > another`	> 0

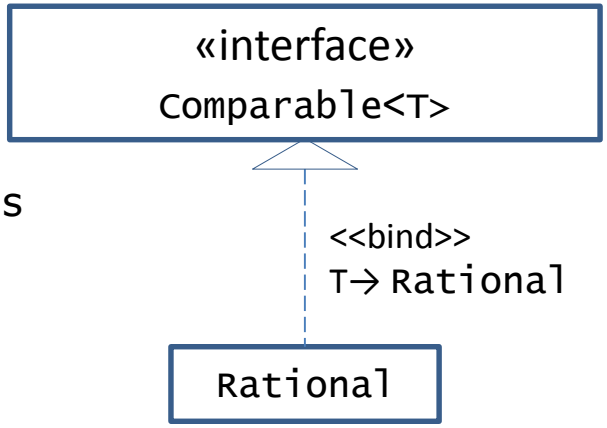
JCF: Boas práticas

- Classe implementa `compareTo`? Então é *de valor*
- Logo, *deve sobrepor* a sua especialização de `equals`…
- …pois por omissão `equals` compara *identidade* e não *igualdade*!
- As operações `compareTo` e `equals` devem ser consistentes…
- …ou seja, `one.compareTo(another) == 0` deve resultar no mesmo que `one.equals(another)`

Aplicação à classe Rational

```
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();
    }
    …
}
```

Esta implementação só está correta se convencionarmos que o denominador é sempre positivo. Neste caso, isso deveria fazer parte da condição invariante.



Aplicação à classe `Rational`

```
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;
    } …
}
```

Aplicação à classe `Rational`

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

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

    …
}
```

Aplicação à classe Aluno

Aluno não tem de estar ordenado sempre da mesma forma (não tem uma ordem natural)

Para ter uma ordem alfabética por nome podemos definir:

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

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

}
```

Classe pacote `collections`

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

sort(List), Ordenar segundo a ordem natural (**Comparable**)

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

A classe **Collections** tem outros métodos úteis, tais como **shuffle(List)** , **reverse(List)**, **min(Collection)**, **max(Collection)**

sort(List, Comparator) , Ordenar segundo um critério

JCF: List e 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);
```

É comum usar um tipo mais genérico para aceder a uma colecção do que a classe real do objecto referenciado. Dessa forma pode-se alterar essa classe alterando apenas uma linha de código.

Fará sentido indexar uma lista? E se se mudar a classe real para `LinkedList`?

Remoção fora do ciclo? O.K.
Remoção dentro do ciclo? Bronca!

JCF: Vector

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

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

courses.add(ip);  // adiciona ao fim
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);  // adiciona no topo
courses.push(poo);
while (!courses.isEmpty()) {
    out.println(courses.peek());
    courses.pop();
}
```

JCF: List, LinkedList e 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();
}
```

Quando possível deve usar-se o interface e não o tipo específico.

Dois em um: avança e devolve. Muito discutível!

Remoção segura: É removido o último elemento devolvido por `next()`.

JCF: Queue e 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 e 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);
}
```

Mais uma vez, dois em um…

JCF: LinkedList e 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);
}
```

Mais uma vez, dois em um…

Ciclo *for-each*

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

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

Modo de iteração compacto, sem usar iterador, mas … de utilização limitada (não se pode alterar a colecção, não se pode facilmente percorrer subsequências da colecção, etc.).

JCF: Iteração e alteração concorrentes

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

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

Alterações durante o ciclo produzem resultados inesperados. Pode mesmo ser lançada a excepção `ConcurrentModificationException`.

JCF: Map e 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 e 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 e 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: Boas práticas na utilização de colecções

- Não usar colecções de `Object`
- Usar o tipo de colecção mais adequado
- Atentar na diferente eficiência das mesmas operações em diferentes tipos de colecção (consultar a documentação)
- Não alterar uma colecção durante uma iteração ao longo dos elementos (ou usar o iterador para o fazer)

JCF: Boas práticas na utilização de colecções

- Alteração de elementos de colecções com ordem intrínseca pode ter efeitos inesperados
- Usar sempre classes (de valor) imutáveis quando for necessária ordem intrínseca
- Ter atenção à documentação: nem todas as colecções permitem a inserção de elementos nulos

Mais informação / Referências

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

Sumário

- Colecções (JCF)