Módulo 7 - Coleções

Objetivos

- Conhecer e saber utilizar as abstrações de tipos de dados oferecidas nas bibliotecas do Java: fila, fila prioritária, conjunto ordenado, dicionário, tabela.

Motivação: poupar esforço e maximizar fiabilidade

A construção eficiente de software complexo requer a reutilização de componentes de software (i.e., "peças" já desenvolvidas por terceiros). Diz-se habitualmente que temos de evitar "reinventar a roda", significando que quando já existe um pedaço de software que já faz (bem) aquilo que necessitamos, então devemos utilizá-lo e não desenvolver a nossa versão. As motivações são diversas, desde o esforço que é poupado (por não termos que desenvolver), bem como a fiabilidade, pois os componentes reutilizáveis tipicamente já foram muito testados e otimizados, e logo, oferecem garantias que dificilmente alcançaremos (sem despender tempo considerável). Neste módulo o foco é a utilização de estruturas de dados, componentes reutilizáveis disponíveis nas bibliotecas de todas as linguagens de programação modernas, entre as quais o Java..

Imaginando que temos um canivete Suíço, com ele podemos fazer qualquer obra (em teoria), pois tem faca, serra, chave de fendas, saca-rolhas, etc. Porém, quando precisamos de uma ferramenta destas para um trabalho mais a sério (que não seja abrir uma garrafa de vinho), iremos certamente ter a vida mais facilitada se utilizarmos uma ferramenta dedicada para o efeito. Fazendo o paralelo para a programação, conseguimos resolver qualquer problema só com vetores/listas, mas iremos ser muito mais produtivos e ter soluções mais eficientes se utilizarmos estruturas de dados adequadas ao problema. Em problemas exigentes poderá até ser apropriado construir estruturas específicas com base nas existentes para uma melhor adequação às necessidades. Saber utilizar as estruturas certas no contexto certo é uma competência muito importante (que pode demorar anos a dominar). Citando Linus Trovalds (inventor do Linux e do Git), *"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."*

Tipos de dados abstratos

Um tipo de dados abstrato é uma noção de estrutura de dados que disponibiliza um conjunto de operações que oferecem determinadas garantias. Consoante o problema em questão, deveremos averiguar quais os tipos abstratos de dados mais adequados. Por um lado, boas escolhas poderão simplificar significativamente a resolução do problema, poupando esforço de desenvolvimento de código. Por outro, ainda que o código tenha uma complexidade semelhante, uma alternativa aparentemente inocente entre duas estruturas pode implicar diferenças de performance muito significativas.

A tabela seguinte resume os principais tipos abstratos de dados, a sua representação nas interfaces do Java, as classes que as implementam, e menciona a técnica de concretização utilizada nas mesmas. Todas as interfaces e classes pertencem ao pacote java.util. Alguns destes elementos já foram utilizados em módulos anteriores (Pilha, Lista), os restantes serão introduzidos neste módulo.

Tipos de dados abstratos	**Interface**	**Classes**	**Concretização**
Pilha	-	Stack (mau desenho)	Vetor
Lista	List	ArrayList	Vetor
		LinkedList	Nós ligados
Fila com duas pontas (*double-ended queue)*	Deque		
		ArrayDeque	Vetor circular
Fila Prioritária	Queue	PriorityQueue	Amontoado (*binary heap*)
Conjunto Não-Ordenado	Set	HashSet	Tabela de dispersão (*hashtable*)
Conjunto Ordenado	SortedSet	TreeSet	Árvore de pesquisa (*red-black tree*)
Tabela	Map	HashMap	Tabela de dispersão (*hashtable*)
Dicionário	SortedMap	TreeMap	Árvore de pesquisa (*red-black tree*)

Interfaces java.util.Queue e java.util.Deque (Fila)

A noção elementar de Fila baseia-se na analogia com as "filas" do mundo real, e é representada na interface java.util.Queue<E> que declara as seguintes operações elementares:

- offer(E), adiciona um elemento no fim da fila
- peek() : E, consulta o elemento que está na frente da fila (caso exista)
- poll() : E, remove (e devolve) o elemento que está na frente da fila (caso exista)
- size() : int, devolve o número de elementos na fila

Esta noção de Fila apenas permite que sejam adicionados elementos no fim da fila, e removidos elementos do início. Uma noção estendida consiste na Fila com duas pontas, onde é possível também acionar elementos na frente da fila, bem como remover do final. Esta noção é representada na interface java.util.Deque<E>, a qual declara, para além das operações acima, as seguintes:

- offerFirst(E), adiciona um elemento na frente da fila
- peekLast() : E, consulta o elemento que está no fim da fila (caso exista)
- pollLast() : E, remove (e devolve) o elemento que está no fim da fila (caso exista)

A classe java.util.ArrayDeque consiste numa implementação de Deque que se baseia num vetor circular. A classe java.util.LinkedList, já abordada anteriormente por ser uma implementação da interface java.util.List, também implementa Deque, e baseia-se em nós ligados. ArrayDeque é mais económica em termos de memória e desempenho, e caso seja necessário manter apenas uma fila, será a escolha mais adequada por esta razão. Em situações em que a fila também tem que ser manipulada com remoções no meio, LinkedList poderá ser mais adequada.

Exemplo:

```
Deque<String> deque = new ArrayDeque<>();
deque.offer("b");             // [b]
deque.offer("c");             // [b, c]
deque.offer("d");             // [b, c, d]
deque.offerFirst("a")         // [a, b, c, d]
String poll = deque.poll();   // "a"
String peek = deque.peek();   // "b"
```

Interface java.util.Comparable

Num módulo anterior foi introduzida a interface java.util.Comparator para representar critérios de comparação para efeitos de ordenação. As bibliotecas do Java utilizam também java.util.Comparable, que é uma interface relacionada com Comparator. A interface tem a apenas uma operação que se assemelha à da interface Comparator. A diferença consiste estar desenhada para ser definido um método de instância do objeto que será comparado.

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

Por exemplo, a class java.lang.Integer, cujos objetos representam inteiros, implementa Comparable. Desta forma é possível fazer:

```
Integer i = new Integer(1);
Integer j = new Integer(3);
int comp = i.compareTo(j);          // -2 (i antecede j)
```

O propósito desta interface é definir classes de objetos comparáveis, adequado nos casos onde o tipo de objetos tem uma ordem natural (intrínseca). Este tipo de objetos são muitas vezes numéricos, sendo que a ordem natural típica será a ordem crescente.

Comparador externo

```
Asc comp = new Asc();
int c = comp.compare(5, 7);
```

```
public class Asc implements Comparator<Integer> {
  public int compare(Integer a, Integer b) {
    return a - b;
  }
}
```

Comparador interno

```
Integer i = 5;
int c = i.compareTo(7);
```

```
public class Integer implements Comparable<Integer> {
  private int value;
  // ...
  public int compareTo(Integer integer) {
    return value - integer.value;
  }
}
```

Perante um vetor ou uma lista de objetos comparáveis, podemos utilizar os métodos de ordenação do Java sem fornecer um objecto comparador externo (java.util.Collections), e dessa forma será utilizado o comparador interno da classe.

```
List<Integer> list = ...
Collections.sort(list);
```

Se o objetivo for utilizar outro comparador que não o interno, então utiliza-se um externo, sendo o interno ignorado. Se objetivo for utilizar o comparador interno inverso (p.e. no caso dos inteiros, representa a ordem decrescente), podemos fazer:

```
Collections.sort(list, Collections.reverseOrder());
```

A seguinte classe, para representar valores monetários com unidades e parte decimal, é um exemplo de implementação da interface Comparable. O método comparador verifica se as unidades do objeto são iguais às unidades do argumento, em caso afirmativo é devolvida a diferença entre os valores decimais, e em caso negativo é devolvida a diferença entre as unidades.

```
public class MoneyAmount implements Comparable<MoneyAmount> {
  private int units;
  private int decimals;
  // ...
  public int compareTo(MoneyAmount a) {
    return units == a.units ? decimals - a.decimals : units - a.units;
  }
}
```

Quando estamos perante um tipo de objetos que não tem um critério intrínseco de comparação (ordem natural), então a utilização desta interface na sua classe não será a solução adequada. Ao fazê-lo a classe ficará acoplada a um único critério de comparação, quando na verdade há vários possíveis. Nestes casos deverão ser utilizados comparadores externos (Comparator).

Exercício 7.1 - Tempos (duração) comparáveis

A classe Time do Exercício 4.2 tinha o propósito de representar tempos. Entre dois tempos existe uma ordem natural (crescente), e logo fará sentido que estes objetos sejam comparáveis. Complete a classe Time por forma a implementar a interface java.util.Comparable. Teste a ordenação crescente e decrescente de uma lista de tempos desordenada.

Classe java.util.PriorityQueue (Fila Prioritária)

Fila Prioritária é outra noção de fila que também se baseia numa analogia com as "filas prioritárias" do mundo real. Neste tipo de filas, o critério de ordem não é o de chegada (como nas filas regulares), mas sim outro critério variável que define a prioridade dos elementos. Em termos da estrutura de dados, esta garante que a ordem de saída da fila está de acordo com o critério de prioridade fornecido. Quando dois elementos têm a mesma prioridade, não há garantias em relação a qual deles sairá primeiro. Atenção que uma Fila Prioritária não corresponde a um vetor ordenado, embora seja possível obter um efetuando o esvaziamento da fila.

As bibliotecas do Java fornecem a classe java.util.PriorityQueue para filas prioritárias, a qual pode ser utilizada de duas formas: com elementos comparáveis (classes que implementam java.util.Comparable), ou com elementos (comparáveis ou não) e um comparador (java.util.Comparator) para fornecer o critério de prioridade.

Neste primeiro exemplo, assumindo que a classe Time implementa Comparable, o objeto PriorityQueue é inicializado sem argumento, e será utilizado o comparador interno de Time.

```
PriorityQueue<Time> pq = new PriorityQueue<>();
pq.offer(new Time(3, 30));
pq.offer(new Time(0, 20));
pq.offer(new Time(1, 50));
while(!pq.isEmpty())
  System.out.println(pq.poll());
```

```
> 0:20
> 1:50
> 3:30
```

Neste segundo exemplo em baixo, embora a classe String implemente Comparable (ordem alfabética), o objeto PriorityQueue é inicializado com um comparador que dá prioridade às strings com maior comprimento.

```
Comparator<String> comp = (a,b) -> b.length() - a.length();
PriorityQueue<String> pq = new PriorityQueue<>(comp);
pq.offer("Viriato");
pq.offer("Claudio Unimano");
pq.offer("Tito Livio");
while(!pq.isEmpty())
  System.out.println(pq.poll());

> Claudio Unimano
> Tito Livio
> Viriato
```

Exercício 7.2 - Horário de dúvidas

Suponha que a ordem de atendimento no horário de dúvidas de um professor obedece às seguintes regras:

1. Quem fez marcação prévia tem prioridade sobre quem não tem marcação, sendo que entre quem fez marcação a prioridade é por ordem de marcação.
2. Para quem não tem marcação, a prioridade é inversa à antiguidade na UC (os alunos com menos inscrições têm prioridade sobre os que têm mais)

Desenvolva uma classe para fazer a gestão da ordem de atendimento. Inclua operações para:

- Efetuar marcação de um aluno
- Assinalar chegada de um aluno
- Obter o próximo aluno a ser atendido (se não houver alunos em espera, devolve null)

`class Aluno {`
`  String nome;`
`  int numeroInscricoes;`
`  ...`
	`class HorarioDuvidas {`
	`  ...`
	`  void marcacao(Aluno a) { ...`
	`  void chegada(Aluno a) { ...`
	`  Aluno proximo() { ...`

Exemplo (número de inscrições entre parênteses):
Marcações: Viriato, Claudio
Chegada: Caio (2), Tito (4), Claudio (2), Fabio (1), Tautalo (1), Viriato (5)
Atendimento: Viriato, Claudio, Fabio, Tautalo, Caio, Tito

Padrão Objeto de Valor (*Value Object*)

Alguns tipos de objetos representam valores que não correspondem a uma entidade no sistema que estamos a desenvolver. Por exemplo, num sistema de gestão de inscrição em disciplinas, cada *aluno* e cada *disciplina* são entidades que não se pretendem duplicadas em objetos idênticos em termos de conteúdo. Por outro lado, objetos para representar valores neste contexto, tais como número de inscrições ou datas, não têm identidade. Podemos ter vários objetos idênticos para uma data, porém ao efetuar uma comparação gostaríamos que fossem considerados iguais. A este tipo de objetos chamamos *objetos de valor*. Existem diversas classes de objetos de valor no Java, uma para cada tipo primitivo (Integer, Double, Boolean, etc), String, etc. Outros exemplos típicos de objetos de valor são tempos, datas, dias da semana, cores.

Um objeto de valor deve ser imutável, pois ao fazer uma atribuição é desejável que tenha a semântica de uma cópia, tal como nos tipos primitivos. Por outro lado, para efeitos de verificação de igualdade, as classes de objetos de valor devem sobrepor o método equals(Object), bem como o hashCode(), mantendo o seu contrato (ver caixa).

Contrato equals(...) / hashCode()

Ao ler a documentação de java.lang.Object, é exigido como *contrato* do objeto que se equals(...) devolver verdadeiro para dois objetos *a* e *b*, então os valores retornados pelo método hashCode() de ambos terão que ser iguais. Desta forma, ao sobrepor o método equals(...) será "obrigatório" sobrepor também hashCode() de forma a que a propriedade seja mantida (a.equals(b) ⇒ a.hashCode() == b.hashCode()). Ao violar o contrato destas operações ocorrerão *bugs* difíceis de detetar, nomeadamente quando são utilizadas estruturas de dados baseadas em tabelas de dispersão (*hashtable*). Estas fazem uso do método hashCode() para fazer o espalhamento na tabela, e a violação do contrato dá origem a resultados inesperados (sem que ocorra um erro).

Importante: Na prática, estes métodos só devem ser sobrepostos com o propósito de concretizar Objetos de Valor, não havendo outra razão óbvia para o fazer.

Quando os objetos são compostos por um único valor numérico, esse será o valor retornado por hashCode(). Caso contrário, deverá ser feito um cálculo com base nos diversos valores que compõem o objeto (seguindo uma "receita", p.e. a que está incorporada no gerador de código do Eclipse). As classes de objetos de valor não devem permitir ser derivadas. O seguinte exemplo consiste numa classe para representar cores como objetos de valor, onde a comparação é feita em função do atributo, e dado que este é inteiro, corresponde ao valor retornado por hashCode().

```
public final class Color {
   private final int value;
   ...
   public boolean equals(Object o) {
      return o != null && o instanceof Color && value == ((Color) o).value;
   }

   public int hashCode() {
     return value;
   }
}

Color a = new Color(255, 0, 0);
Color b = new Color(255, 0, 0);
boolean equals = a.equals(b);                  // true
int hashA = a.hashCode();                      // 16711680
boolean hashEquals = hashA == b.hashCode();    // true
```

Exercício 7.3 - Tempos (duração) como objetos de valor

Faça evoluir a classe Time do exercício 4.2 por forma a que os seus objetos sejam de valor, sobrepondo equals(...) e hashCode(). Experimente:

- Verificar a igualdade entre objetos Time;
- Fazer pesquisas numa lista, comparando a versão anterior da classe com a nova. O comportamento deverá ser diferente, pois a operação contains(...) faz uso do método equals(...).

```
List<Time> list = ...
list.add(new Time(1, 20, 0));
...
boolean b = list.contains(new Time(1, 20, 0));  // ?
```

Interface java.util.Set (Conjunto Não-Ordenado)

As bibliotecas do Java têm a interface java.util.Set para representar Conjuntos Não-Ordenados. Os elementos que compõem um conjunto são todos distintos, não sendo possível haver duplicados. Esta propriedade é garantida na inserção, pois adicionar um elemento já existente na coleção não terá efeito, ficando a mesma inalterada.

A classe java.util.HashSet consiste numa implementação de Set baseada numa tabela de dispersão (*hashtable*). Devido a este facto, há que ter em especial atenção o tipo dos elementos do conjunto, pois a tabela de dispersão utiliza o método hashCode(). Os objetos do conjunto devem ser entidades (que não sobrepõem equals(...) / hashCode()) ou Objetos de Valor imutáveis (p.e. String, Integer). Por outras palavras, o valor retornado por hashCode() tem que ser constante. O exemplo seguinte ilustra a criação de um conjunto a partir de uma lista de strings (com duplicações). Cada elemento da lista de letras é adicionado ao conjunto, sendo depois impresso cada elemento do conjunto.

```
List<String> letters = Arrays.asList("I", "E", "I", "A", "U", "O", "A");
Set<String> set = new HashSet<>();
letters.forEach(l -> set.add(l));
set.forEach(e -> System.out.println(e));

> A
> E
> U
> I
> O
```

Como é possível ver pelo resultado, as letras diferentes aparecem todas e sem repetições, porém, a ordem pela qual ocorrem na iteração não é relativa à ordem da lista inicial, nem tão pouco de acordo com a comparação de strings (a classe String implementa java.util.Comparable). Isto é devido à forma de funcionamento da tabela de dispersão, que "espalha" os elementos acordo com uma função não inversível, não sendo possível recuperar a ordem de inserção. Por outro lado, a organização dos elementos também não considera nenhum tipo de ordenação intrínseca dos elementos. Se para efeito do problema que estamos a resolver utilizando um conjunto nenhuma destas propriedades é relevante, então HashSet será uma estrutura adequada. A sua performance é muito superior quando comparada com a utilização de uma lista onde é restringida a inserção de duplicados "manualmente" (verificando previamente a existência de um elemento com a operação contains(...)).

Exercício 7.4 - Contagem de vocabulário

Desenvolva um programa que dado um ficheiro de texto, indique quantas palavras diferentes são utilizadas. Converta as palavras encontradas para minúsculas, por forma a considerar que palavras como "Ramalhete" e "ramalhete" correspondem à mesma.

```
java WordCount OsMaias-Cap1.txt

> 2788
```

Para utilizar a classe Scanner por forma a ignorar números e pontuação é possível definir um delimitador para dividir as palavras que considera outros carateres (e não só os espaços). O código seguinte permite iterar sobre as palavras de um ficheiro de texto ignorando pontuação e números.

```
Scanner scanner = new Scanner(new File(...));
scanner.useDelimiter("[,\\.;:\\-\\?!»«\\(\\)0-9\\s]+");
while(scanner.hasNext()) {
  String word = scanner.next();
  // ...
}
```

Interface java.util.SortedSet (Conjunto Ordenado)

As bibliotecas do Java têm a interface java.util.SortedSet para representar Conjuntos Ordenados. Este tipo de dados é uma especialização de Conjunto Não-Ordenado, garantindo que a iteração pelos elementos é feita de forma ordenada, quer de acordo com a ordem intrínseca dos elementos (quando a sua classe implementa java.util.Comparable), ou de acordo com um comparator externo.

A classe java.util.TreeSet consiste numa implementação de SortedSet baseada numa árvore de pesquisa (*red-black tree*). Dado que os conjuntos de elementos são ordenados, torna-se necessário fornecer um critério de comparação, ou então para objetos comparáveis utilizar o comparador interno, tal como no caso da Fila Prioritária.

O seguinte exemplo demonstra a utilização de TreeSet, onde podemos observar que a partir de uma lista desordenada de strings com repetições podemos obter o conjunto de strings distintas de forma ordenada. O facto dos elementos estarem ordenados permite também aceder ao primeiro e último elemento, bem como efectuar pesquisas de intervalos.

```
List<String> letters = Arrays.asList("I", "E", "I", "A", "U", "O", "A");
SortedSet<String> set = new TreeSet<>();
letters.forEach(l -> set.add(l));
set.forEach(e -> System.out.println(e));
System.out.println("primeiro: " + set.first());
System.out.println("ultimo: " + set.last());

> A
> E
> I
> O
> U
> primeiro: A
> ultimo: U
```

As árvores de pesquisa mantêm a ordem dos elementos a cada inserção, consoante o comportamento do comparador. Caso um elemento já inserido seja modificado, e essa modificação implique uma alteração de ordem, a ordem da árvore ficará incoerente porque a mesma não se reajustará automaticamente quando o objeto muda. Por essa razão, os elementos de um conjunto ordenado TreeSet deverão ser objetos imutáveis.

Exercício 7.5 - Pesquisa de palavras

Desenvolva um programa que recebe o nome de um ficheiro como argumento e processa esse ficheiro por forma a indicar as palavras alfabeticamente compreendidas entre dois argumentos dados (o segundo exclusive). As palavras deverão ser tratadas uniformemente apenas utilizando minúsculas, e deverão ser consideradas apenas palavras com mais de três letras.

```
java WordSearch OsMaias-Cap1.txt lisboa livre

> lisboa
> lisboeta
> literatura
> livraria
```

Utilize um objeto TreeSet para reunir o conjunto de palavras diferentes. Para obter o intervalo de palavras utilize a operação subSet(...).

Interface java.util.Map (Tabela)

As bibliotecas do Java têm a interface java.util.Map para representar Tabelas (também chamadas de Vetores Associativos). Esta abstração consiste num conjunto não-ordenado de associações (chave → valor). Neste tipo de dados abstrato as inserções consistem em pares (chave, valor), ao passo que as pesquisas e remoções são feitas com base na chave. As chaves são únicas, sendo que uma inserção com uma chave que já existe irá substituir a entrada existente. Porém, pode dar-se o caso de duas chaves distintas estarem associadas ao mesmo valor.

A classe java.util.HashMap é uma implementação de Map baseada numa tabela de dispersão. Tal como java.util.HashSet, a sua utilização requer o mesmo especial com o tipo das chaves utilizadas. O seguinte exemplo ilustra a utilização de uma tabela que associa strings a inteiros.

```
Map<String, Integer> map = new HashMap<>();
map.put("A", 1);                              // {"A" → 1}
map.put("B", 3);                              // {"B" → 3, "A" → 1}
map.put("C", 1);                              // {"B" → 3, "A" → 1, "C" → 1}
map.put("A", 2);                              // {"B" → 3, "A" → 2, "C" → 1}
Integer i = map.get("A");                     // 2
boolean containsA = map.containsKey("A");     // true
boolean containsD = map.containsKey("D");     // false
map.remove("A");                              // {"B" → 3, "C" → 1}
```

Exercício 7.6 - Contagem de palavras (top-n)

Desenvolva um programa para exibir as *top-n* ocorrências de palavras num ficheiro. O programa deverá receber como argumentos o nome de um ficheiro, um número *m* para indicar o número de letras mínimo das palavras a considerar, e um número *n* para indicar o valor para o *top-n*. Utilize um objeto HashMap<String, Integer> para manter a associação entre as palavras encontradas e contagem de ocorrência das mesmas. Para a cada palavra *p*, será verificado se a mesma existe na tabela: caso não exista é inserido o par chave-valor (*p*, 1), caso contrário a entrada será substituída pelo valor existente mais um.

```
java WordCount OsMaias-Cap1.txt 5 7

> afonso: 59
> pedro: 35
> vilaça: 28
> depois: 26
> lisboa: 24
> quando: 20
> benfica: 19
```

Interface java.util.SortedMap (Dicionário)

As bibliotecas do Java têm a interface java.util.SortedMap para representar Dicionários, o que pode ser visto como uma Tabela onde as associações (chave → valor) são mantidas ordenadas por chave. A classe TreeMap é uma implementação de SortedMap baseada numa árvore de pesquisa para as chaves. Pelas mesmas razões de TreeSet, as chaves deverão ser objetos imutáveis.

```
SortedMap<Integer, Aluno> map = new TreeMap<>();
map.put(3, new Aluno(3, "Tito Livio"));     // [3->Tito Livio]
map.put(1, new Aluno(1, "Viriato"));        // [1->Viriato,3->Tito Livio]
map.put(2, new Aluno(2, "Tautalo"));        // [1->Viriato,2->Tautalo,3->Tito Livio]

for(Entry<Integer, Aluno> e : map.entrySet())
  System.out.println(e.getKey() + ": " + e.getValue());

> 1: Viriato
> 2: Tautalo
> 3: Tito Livio
```

Exercício 7.7 - Contagem de palavras (intervalo)

Nos mesmos moldes do Exercício 5.4, desenvolva um programa que dado o nome de um ficheiro como argumento e dois argumentos, apresente o número ocorrências de cada palavra no intervalo.

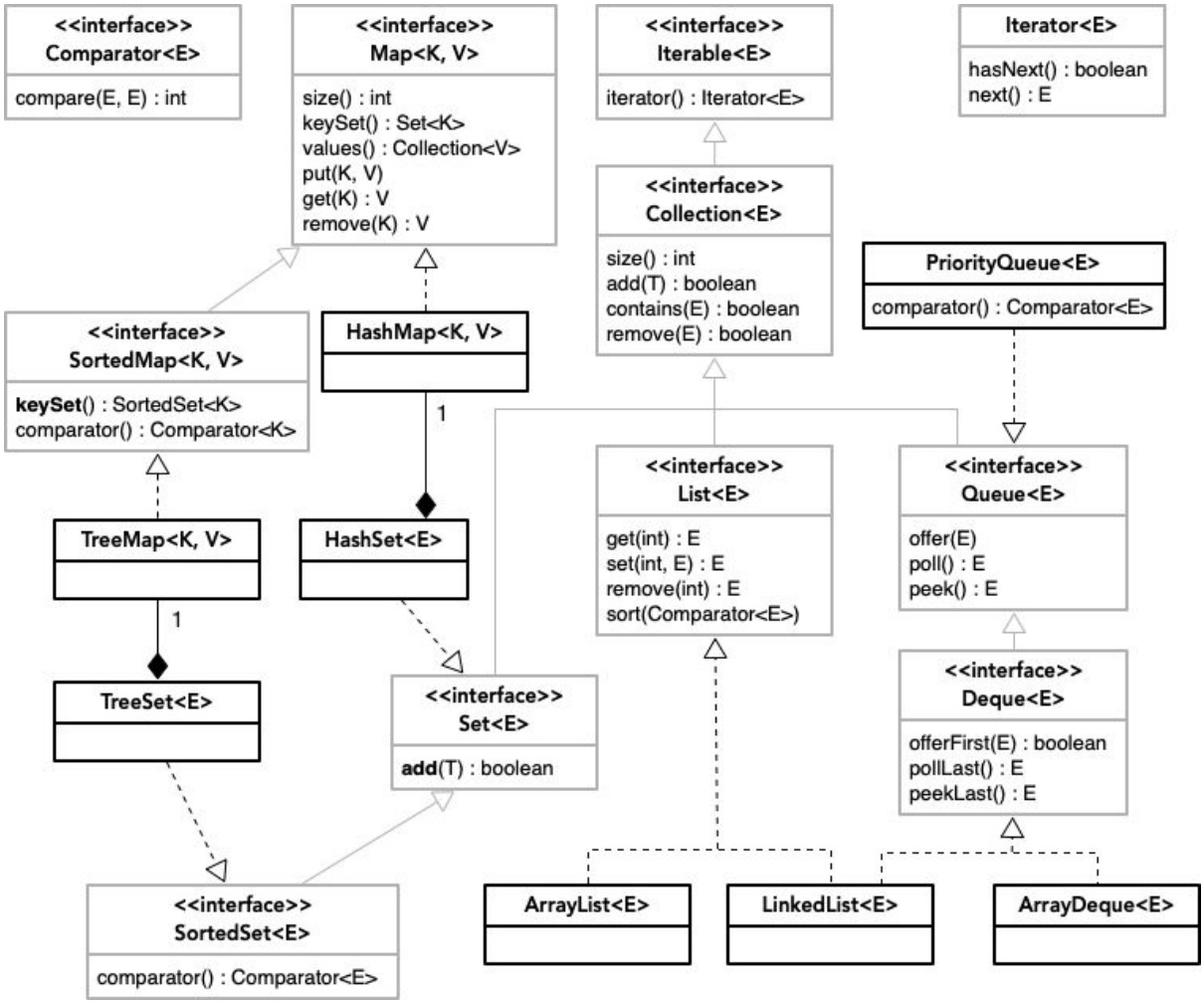
```
java WordCount OsMaias-Cap1.txt lisboa livre
> lisboa: 24
> lisboeta: 1
> literatura: 2
> livraria: 2
```

Para obter o intervalo de associações entre duas chaves utilize a operação subMap(...).

Hierarquia de coleções

O diagrama seguinte apresenta um resumo das principais estruturas de dados oferecidas pelas bibliotecas do Java. Note que as operações listadas são apenas as mais importantes, representando apenas características essenciais que distinguem as diferentes abstrações. É importante entender as propriedades de cada tipo de dados abstrato e saber relacioná-lo com outros, por forma a conseguir fazer uma escolha informada aquando da resolução de um problema. Geralmente, a escolha do tipo de dados (interface) é feita em função das características inerentes ao problema que estamos a resolver. Por outro lado, a escolha de

uma implementação concreta (classe) será relativa a questões de desempenho (velocidade/memória) com influência do conjunto de dados que se vai manipular, as restrições do ambiente de execução, e a frequência das operações a executar.



Observações:

- Todas as coleções são iteráveis (Iterable);
- Há várias alternativas para uma mesma interface (p.e. ArrayList e LinkedList)
- A interface Set é igual à Collection, mas o método add(...) tem outra semântica, pois não permite duplicados;
- Comparator é utilizada nas estruturas que envolvem ordenação (PriorityQueue, SortedSet, SortedMap, List);
- As chaves de Map e SortedMap são conjuntos (Set e SortedSet, não permitem duplicados), ao passo que os valores são coleções (Collection)
- TreeSet é implementada com base em TreeMap, e HashSet é implementada com base em HashMap.

Exercício 7.8 - Pilha (interface e implementação)

Aproveitando o facto do Java não ter uma interface para representar pilhas, bem como da implementação de java.util.Stack não ter o melhor desenho, neste exercício o objetivo é desenvolver tanto a interface para representar o tipo abstrato de dados, bem como uma classe que a implementa.

a) Defina uma interface genérica Stack<E> para pilhas com as seguintes operações:
   i) push(E), insere um elemento no topo da pilha.
   ii) peek() : E, consulta o elemento que está no topo da pilha (caso exista).
   iii) pop() : E, remove (e devolve) o elemento que está no topo da pilha (caso exista).
   iv) size() : int, devolve o número de elementos da pilha.
   v) isEmpty() : boolean, devolve verdadeiro caso a pilha esteja vazia, falso caso contrário.

b) Desenvolva uma classe ArrayStack que implementa a interface, sendo concretizada com base num objeto ArrayList.

c) Altere a implementação do Exercício 1.5 (avaliador de expressões, revista no Exercício 3.4) por forma a utilizar a classe desenvolvida. As alterações no código deverão ser mínimas caso o nome das operações da pilha seja igual ao proposto na alínea (a).