Instituto Superior de Engenharia de Coimbra

Programação Avançada (2020/21)

Exercícios

ArrayList

Um objecto do tipo ArrayList<T> representa uma sequência ordenada de objectos do tipo T. Implementa o interface List<T>.

A seguinte instrução cria um *ArrayList* vazio de elementos do tipo `Produto`:

```
List<Produto> produtos= new ArrayList<Produto>();
```

Alguns métodos:

`public boolean add(E e)` acrescenta o elemento **`e`** ao fim da lista.

`public E remove(int index)` remove o elemento da posição **`index`**, ajustando os índices seguintes; retorna o elemento removido.

`public boolean remove(Object o)` remove a primeira ocorrência de **`o`**, se existir, ou seja remove o elemento com menor índice i tal que (o==null ? get(i)==null : o.equals(get(i))) (se esse elemento existir); retorna true se o elemento foi encontrado e eliminado.

`public void clear()` remove todos os elementos da lista.

`public int size()` retorna o numero de elementos da lista.

`public int indexOf(Object o)` retorna o índica da primeira ocorrência de **`o`** na lista, ou -1 se não existir.he element, ou seja, retorna o elemento com menor índice i tal que (o==null ? get(i)==null : o.equals(get(i))) ou -1 se este índice não existir.

`public E get(int index)` retorna o elemento da lista na posição **`índex`**.

Percorrer um ArrayList:

```
for( int i = 0 ; i < produtos.size(); i++){
            // …  produtos.get(i)
}

for(Produto produto: produtos){
            // …  produto;
}
```

Interface Iterator

Um *iterator* é um objecto que lhe permite percorrer uma colecção.

As colecções têm um método:

```
Iterator<E> iterator();
```

Que retorna um *iterator* para a colecção que o invoca.

```
public interface Iterator<E> {
    boolean hasNext();  // retorna true se existem mais elementos
    E next();           // retorna o próximo elemento da colecção
    void remove();
}
```

Interface Set

Um set é uma colecção que não pode ter elementos repetidos.

Uma das implementações do interface `Set` é **`HashSet.`**

A seguinte instrução cria um *HashSet* vazio de elementos do tipo `Entidade`:

```
Set< Entidade > c = new HashSet< Entidade >();
```

Percorrer uma colecção com `iterator`:

```
Iterator<Entidade> it = v.iterator();
      while (it.hasNext()){
            // … it.next();
      }
  }
```

Para verificar se um determinado objecto pertence a um *HashSet*, é calculado o *hashCode* do objecto. O objecto é comparado com `equals()` só com os elementos do *HashSet* que têm o mesmo *hashCode* do objecto que está a ser pesquisado.

O objecto é considerado como pertencente ao *HashSet* se tiver o mesmo *hashCode* de um elemento e se também em relação a esse elemento a comparação com `equals()` der verdadeira.

Os objectos que se colocam num *HashSet* devem dispor de funções `equals()` e `hashCode()` devidamente definidas (**se `x.equals(y)` for `true`, então `x.hashCode() == y.hashCode()`**).

Ao adicionar um elemento a um *HashSet*, esta operação só tem sucesso (retornando `true` e adicionando de facto) se o novo elemento não pertencer ainda ao *HashSet*.

Interface Map

Um objecto do tipo Map representa uma correspondência chave-valor. Não pode haver chaves repetidas.

```
public interface Map<K,V> {

    V put(K key, V value);      // insere a correspondência de (key, value)
    V get(Object key);          // retorna o valor associado a key
    V remove(Object key);       // remove a associação cuja chave é key
    boolean containsKey(Object key); // contém uma associação com chave key
    boolean containsValue(Object value); // contém uma associação com valor value
    int size();
    boolean isEmpty();          // estar vazio
    void clear();               // remove todas as associações
    public Set<K> keySet();     // conjunto das chaves
    public Collection<V> values(); // colecção dos valores
}
```

Uma das implementações do interface `Map` é **`HashMap`.**

A seguinte instrução cria um *HashMap* vazio de elementos do tipo `Entidade`:

```
Map<String,Entidade> map = new HashMap<String, Entidade>();
```

Percorrer um map:

```
Set<String> chaves = map.keySet();
Iterator<String> iter = chaves.iterator();
while (iter.hasNext()) {
    String key = iter.next();
    str += " \n" + map.get(key);
}
```

Interfaces Comparable e Comparator

Implementando o interface **`Comparable`** a classe define uma ordem natural entre os seus elementos.

```
public abstract class Cartao implements Comparable<Cartao>{

    protected List<Chamada> chamadas = new ArrayList<Chamada>();
    private int numero;
    private double saldo;
    // …
    public int compareTo(Cartao o) {
        return numero - o.getNumero();
    }
// …
}

public class Telefonica {
    private String nome;
    private Map<Integer, Cartao> cartoes = new HashMap<Integer, Cartao>();
    // …
    public String toStringPorNumero(){
        List<Cartao> listaOrdenada =
                new ArrayList<Cartao>(cartoes.values()); //Faz a cópia
        Collections.sort(listaOrdenada);
        return "Operadora de telemoveis ***" + nome + "***\n"
                + "Lista ordenada por numero:\n" + listaOrdenada;
    }
    // …
}
```

O interface **`Comparator`** permite ordenar a mesma colecção segundo diversos critérios.

É preciso definir um objecto do tipo `Comparator`, neste caso, um objecto da classe `OrdenaPorSaldo` que vai ser o segundo argumento da função:

```
Collections.sort(listaOrdenada, new OrdenaPorSaldo());
```

```
public class Telefonica {
    private String nome;
    private Map<Integer, Cartao> cartoes = new HashMap<Integer, Cartao>();
// …
class OrdenaPorSaldo implements Comparator<Cartao> {
        public int compare(Cartao o1, Cartao o2) {
            if (o1.getSaldo() < o2.getSaldo())
                return -1;
            else if (o1.getSaldo() > o2.getSaldo())
                return 1;
            else return 0;
        }
    }
    public String toStringPorSaldo(){
        List<Cartao> listaOrdenada =
                new ArrayList<Cartao>(cartoes.values()); //Faz a cópia
        Collections.sort(listaOrdenada, new OrdenaPorSaldo());
        return "Operadora de telemoveis ***" + nome + "***\n"
                + "Lista ordenada por saldo:\n" + listaOrdenada;
    }
// …
}
```

Exercícios

1. Pretende-se uma aplicação para gerir os produtos de uma fábrica.

a) Os produtos são identificados por um número de série (inteiro) que deve ser único e não deve poder ser modificado a partir do exterior da classe. São ainda caracterizados pela data de fabrico e também por um estado (string) que inicialmente possui o valor "nao testado". Um objecto da classe `Produto` só pode ser criado com a indicação do seu número de série. A data de fabrico corresponde à data do momento de criação do objecto. Esta classe deverá ter um método booleano `testaUnidade()` que simula o controlo de qualidade. Quando esta função é invocada por um objecto, se este estiver no estado “não testado”, em 90% dos casos estará OK (o estado passará de "nao testado" para "aprovado"). Caso não esteja OK, o estado passará para "reprovado". A função `testaUnidade()` não altera o estado caso este seja “aprovado” ou “reprovado”, retornando true se o estado é aprovado e false se não é. Os membros variáveis devem ser privados, podendo ser acedidos para através de funções *get* e *set*. Nesta classe deve implementar as funções:

i) `toString()` que retorna uma *string* com a descrição do objecto;

ii) `equals()` que representa o critério de identificação de um produto (dois produtos são o mesmo se tiverem o mesmo numero de série).

iii) `hashCode()` que retorna o hash code do objecto.

b) Defina a classe `Fabrica` que gere um conjunto de produtos. Para além de uma colecção de produtos, a fabrica tem um nome e o número de produtos criados até ao momento (que pode ser diferente do número de elementos da colecção de produtos, uma vez que podem já ter sido alguns eliminados). Ao ser criado um objecto da classe `Fabrica` deve ser dado o seu nome, ficando, à partida, sem registo de nenhum produto. Esta classe deve ter as funções:

i) `acrescentaProduto()` que cria o produto e acrescenta-o à fabrica;

ii) `pesquisaProduto()` que recebe o número de série de um produto e retorna uma referência para o produto se o encontrar ou `null` se não encontrar;

iii) `eliminaProduto()` que recebe o número de série do produto e o elimina se o encontrar; retorna o valor lógico do sucesso desta operação;

iv) `eliminaReprovados()` que elimina todos os produtos cujo estado é "reprovado".

v) `testaUnidades()` que faz o controlo de qualidade a todos os produtos da fábrica.

vi) `toString()` que retorna uma *string* com a descrição do objecto;

2. Pretende-se uma aplicação para gerir os livros de uma biblioteca. Os livros são identificados por um código (um número inteiro que representa a ordem de criação do registo dos livros na biblioteca). O registo de um livro, para além do referido código, tem obrigatoriamente informação sobre o título e os autores.

a) Defina a classe `Livro` que representa este conceito de registo de um livro nesta biblioteca. Deve ser possível criar objectos da classe Livro, dando informação sobre o título e autores, sendo, neste caso o código gerado automaticamente como um inteiro que representa a ordem de criação do registo). Os membros variáveis devem ser privados, podendo ser acedidos para através de funções *get* e *set*. O membro *código* não deve poder ser modificado a partir do exterior da classe. Nesta classe deve implementar as funções:

i) `toString()` que retorna uma *string* com a descrição do objecto;

ii) `equals()` que representa o critério de identificação de um livro (dois livros são o mesmo se tiverem o mesmo código).

iii) `hashCode()` que retorna o hash code do objecto

b) Defina a classe `Biblioteca` que representa um conjunto de livros. Para além dos livros, a biblioteca tem um nome. Ao ser criado um objecto da classe `Biblioteca` deve ser dado o seu nome, ficando, à partida, sem registo de nenhum livro. Esta classe deve ter as funções:

i) `acrescentaLivro()` que recebe toda a informação que permite criar o registo de um livro, cria o registo e acrescenta-o à biblioteca;

ii) `pesquisaLivro()` que recebe o código de um livro e retorna uma referência para o livro se o encontrar ou `null` se não encontrar;

iii) `eliminaLivro()` que recebe o código do livro e o elimina se o encontrar; retorna o valor lógico do sucesso desta operação;

iv) `toString()` que retorna uma *string* com a descrição do objecto.

3. Pretende-se construir uma classe **`Sistemas`**, que oferece a seguinte funcionalidade, obtida através da utilização adequada de mapas (e eventualmente outras estruturas de dados):

```
    public static void main(String[] args) {
        Sistemas d = new Sistemas();
        d.newSystem("Sistema Solar"); //nome do sistema
        d.addStar("Sistema Solar", "Sol"); // acrescenta estrela a sistema
        d.addPlanet("Sistema Solar", "Mercurio"); //acrescenta primeiro
                                                  // planeta do sistema solar...
        d.addPlanet("Sistema Solar", "Venus"); //acrescenta segundo planeta
                                                  // do sistema solar...
        d.addPlanet("Sistema Solar", "Terra"); //acrescenta terceiro planeta
                                                  // do sistema solar...
        d.newSystem("Alfa Centauri");
        d.addStar("Alfa Centauri", "Proxima Centauri");
        d.addStar("Alfa Centauri", "Alfa Centauri A");
        d.addStar("Alfa Centauri", "Alfa Centauri B");
        d.addPlanet("Alfa Centauri", "Alfa Centauri Bb");

        System.out.println(d.getPlanet("Sistema Solar", 2)); // venus
        System.out.println(d.getStars("Alfa Centauri"));
                    // [Proxima Centauri , Alfa Centauri A, Alfa Centauri B]
        System.out.println(d.existsSystem("Xanadu")); //false
        System.out.println(d.existsSystem("Sistema Solar")); //true
    }
}
```

4. Pretende-se construir uma classe Dicionário, que oferece a seguinte funcionalidade, obtida através da utilização adequada de mapas:

```
    public static void main(String[] args) {
        Dicionario d = new Dicionario();
        d.add("ingles", "LIVRO", "Book");
        d.add("frances", "LIVRO", "livre");
        d.add("portugues", "LIVRO", "livro");
        d.add("ingles", "ANO", "year");
        d.add("frances", "ANO", "an");
        d.add("portugues", "ANO", "ano");

        d.defineLingua("ingles");
        System.out.println(d.get("ANO")); //year
        d.defineLingua("portugues");
        System.out.println(d.get("ANO"));//ano
        d.defineLingua("frances");
```

```
        System.out.println(d.get("LIVRO"));//livre
    }
```

5. Pretende-se construir uma classe que permite a gestão das notas de uma turma. Esta classe deve utilizar mapas de forma adequada (sem prejuízo da utilização de outras estruturas de dados que se revelem necessárias) .

```
    public static void main(String[] args) {
        Turma t = new Turma();
        t.addAluno("José", 201301); //nome, nº de aluno
        t.addAluno("Luis", 201303);
        t.addAluno("Ana", 201302);

        t.addNota("Luis", 65); //nota do 1º teste do Luis
        t.addNota(201303, 80); //nota do 2º teste do Luis
        t.addNota("Luis", 85); //nota do 3º teste do Luis

        System.out.println(t.getNotaTeste(201303, 3));
                                                //nota do 3º teste do Luis
        System.out.println(t.getNotaTeste("Luis", 2));
                                                //nota do 2º teste do Luis
    }
```

6. Pretende-se construir uma classe que permite a gestão de um inventário de produtos. Esta classe deve utilizar mapas de forma adequada para fornecer as funcionalidades descritas (sem prejuízo da utilização de outras estruturas de dados que se revelem necessárias).

```
  public static void main(String[] args) {
        Inventario t = new Inventario();
        t.addProduto("Coca", 1234, 1); //nome, código, preço
        t.addProduto("Cola", 1235, 2);
        t.addProduto("Chipi", 1236, 3);

        System.out.println(t.getPreco(1235)); //2
        System.out.println(t.getPreco("Coca"));  //1
        System.out.println(t.getCodigo("Chipi")); //1236

        System.out.println(t.getNomes()); //mostra todos os nomes:
                                          //A ORDEM NÃO É RELEVANTE!!!!
    }
}
```

7. Pretende-se um programa que registe medições horárias de temperatura. Quando se tenta introduzir no sistema um registo de um dia com as mesmas temperaturas máximas e mínimas que outro já anteriormente introduzido, isso deve ser indicado. Verifique se o seguinte programa funciona. Caso isso não aconteça, efectue as correcções necessárias.

```
public class TemperaturaDiaria {

    private int temperaturasHorarias[];
    private String responsavelMedicoes;
    private String localMedicao;

    TemperaturaDiaria(int[] temps, String resp, String local) {
        responsavelMedicoes = resp;
        localMedicao = local;
        temperaturasHorarias = new int[temps.length];
        for (int i = 0; i < temperaturasHorarias.length; i++) {
            temperaturasHorarias[i] = temps[i];
        }
    }
    public boolean equals(Object o) {
        if (!(o instanceof TemperaturaDiaria)) {
            return false;
        }
        TemperaturaDiaria outro = (TemperaturaDiaria) o;
        return getMaximo() == outro.getMaximo()
                && getMinimo() == outro.getMinimo();
    }
    public int hashCode() {
        int s = 0;
        for (int t : temperaturasHorarias) {
            s += t;
        }
        return s;
    }

    int getMaximo() {
        int max = temperaturasHorarias[0];
        for (int t : temperaturasHorarias) {
            if (t > max) {
                max = t;
            }
        }
        return max;
```

```
    }
    int getMinimo() {
        int min = temperaturasHorarias[0];
        for (int t : temperaturasHorarias) {
            if (t < min) {
                min = t;
            }
        }
        return min;
    }
}

public class Temperaturas {

    private HashSet<TemperaturaDiaria> temps = new HashSet<>();

    public void acrescenta(TemperaturaDiaria td) {
     if (temps.contains(td)) {
         System.out.println("Já está registado um dia com máximos e mínimos
similares");
     } else {
          System.out.println("Dia com novos máximos e mínimos");
         temps.add(td);
     }
    }
}
public class Main {
   public static void main(String[] args) {
      Temperaturas temperaturas = new Temperaturas();

      int [] t1 = { 1, 2, 3, 4, 5, 6, 7}; // min 1 max 7
      int [] t2 = { 1, 0, 2, 0, 2, 0, 7}; // min 0 max 7
      int [] t3 = { 1, 4, 2, 5, 2, 3, 7}; // min 1 max 7
      System.out.println("1");
      temperaturas.acrescenta(new TemperaturaDiaria(t1, "Aaa", "Coimbra") );
      System.out.println("2");
      temperaturas.acrescenta(new TemperaturaDiaria(t2, "Bbb", "Lisboa") );
      System.out.println("3");
      temperaturas.acrescenta(new TemperaturaDiaria(t3, "Ccc", "Porto") );
   }
}
```