Notas de POO: interfaces, excepções e input/output

2010/11 (1ª versão)
2016/17 (revisão)

Conteúdo

1 Considerações gerais

Algumas notas sobre os exercícios feitos até ao momento:

- a implementação do clone do Java não é útil, pelo que cada classe devolve um `clone()` do seu tipo;
- o nosso clone é *deep* e é feito com recurso ao construtor de cópia;
- não utilizar construções como `addAll(Collection c)`;
- `Strings` e Wrapper classes não necessitam de `clone()` pois são imutáveis;

2 Interfaces

A declaração de uma interface em Java é muito simples dado que a interface apenas poderá conter um identificador, um conjunto opcional de constantes (ou seja static e final) e um conjunto de assinaturas de métodos que são implicitamentes abstractos.

O código necessário para criar uma interface toma a seguinte forma:

```
public interface MinhaInterface  {

  public metodoA();
  public metodoB();
  ....
}
```

Uma classe pode declarar-se do tipo de dados de uma dada interface ao sinalizar tal através da seguinte construção:

```
public class MinhaClasse implements  MinhaInterface  {

  ....
  ....

  public metodoA() {.....}
  public metodoB(){.....}
  ....
}
```

A classe para correctamente implementar a interface, ou interfaces, declarada(s), tem de fornecer uma concretização para os métodos que estão declarados na interface.

Objectos instância dessa classe podem ser vistos como sendo do tipo de dados da interface, sendo que nesse caso o seu comportamento resume-se aos métodos declarados na interface.

2.1 Hierarquia das Interfaces

À semelhança das classes, as interfaces estão organizadas numa hierarquia. No entanto, a herança existente na hierarquia das interfaces é múltipla, o que implica que uma interface pode ser sub-interface de mais do que uma interface.

Podemos ter assim declarações como:

```
public interface MinhaInterface extends Interface1, Interface2 {

  public metodoA();
  public metodo B();
  ....
}
```

Quando os métodos herdados das possíveis super-interfaces forem idênticos é necessário tornar não ambígua a utilização do método recorrendo à sintaxe `NomeInterface.metodo()`.

Uma classe pode implementar mais do que uma interface, sendo que para tal deve explicitamente elencar a lista das interfaces.

```
public class MinhaClasse implements MinhaInterface1, MinhaInterface2  {

....
....
....
}
```

2.2 Interfaces e Classes Abstractas

Por vezes o mesmo conceito pode ser expresso através da utilização de classes abstractas ou de interfaces. Tal resulta do facto de que uma classe totalmente abstracta e uma interface terem o mesmo poder expressivo. No entanto existem algumas diferenças entre ambos os conceitos:

- uma classe abstracta pode não ser 100% abstracta;
- uma classe abstracta não impõe às suas subclasses a implementação dos métodos abstractos - uma interface obriga a quem a implementa a tornar os métodos concretos;
- uma classe abstracta pode ser utilizada para criar os mecanismos para a construção de software genérico, extensível e parametrizável. Uma interface não tem genericamente essas preocupações;
- classes e interfaces pertencem a duas hierarquias distintas, podendo no entanto o programador conjugar ambas por forma a aumentar a capacidade expressiva dos seus programas.

2.3 A classe Class

A classe `Class` permite que em tempo de execução se possam saber algumas propriedades importantes dos objectos que populam uma aplicação.

Instâncias da classe `Class` representam classes e interfaces num ambiente de execução Java. É possível por exemplo a uma instância desta classe enviar metódos como:

- public static Class forName(String classname)
- public Object newInstance()
- public boolean isInstance(Object obj) // equivalente dinâmico ao operador instanceOf
- public boolean isInterface()
- public Class getSuperClass()

- public Class[ ] getInterfaces()
- etc.

É por exemplo possível determinar quais as interfaces a que determinados objectos (pertencentes por exemplo a uma colecção) respondem.

O método `getInterfaces` pode ser invocado para determinar quais as interfaces com que uma determinada classe está comprometida. Este método devolve um array de objectos do tipo `Class`.

Vejam-se por exemplo as seguintes declarações, em que em tempo de execução se interroga quais as interfaces implementadas.

```
import java.lang.reflect.*;
import java.io.*;

class SampleInterface {

   public static void main(String[] args) {
      try {
          RandomAccessFile r = new RandomAccessFile("myfile", "r");
          printInterfaceNames(r);
      } catch (IOException e) {
          System.out.println(e);
      }
   }

   static void printInterfaceNames(Object o) {
      Class c = o.getClass();
      Class[] theInterfaces = c.getInterfaces();
      for (int i = 0; i < theInterfaces.length; i++) {
         String interfaceName = theInterfaces[i].getName();
         System.out.println(interfaceName);
      }
   }
}
```

3 Excepções

O mecanismo de excepções em Java, permite que o programador construa as suas próprias classes de excepção, de forma muito simples, por herança da superclasse das excepções, a classe Exception.

O código necessário para criar uma classe de excepções toma a seguinte forma:

```
class MinhaExcepcao extends Exception {
  MinhaExcepcao() {super();}
  MinhaExcepcao(String s) {super(s);}
}
```

3.1 Lançar uma excepção

Quando existe uma situação de erro, por exemplo retirar um elemento da colecção quando ele não existe, o programador pode explicitamente sinalizar o envio de um erro que deve ser posteriormente tratado.

Para tal apenas é necessário criar uma instância da excepção pretendida, da forma:

```
...
throw new <nomeClasseExcepcao>(); //construtor vazio
...
...
throw new <nomeClasseExcepcao>("mensagem que queira enviar");
//construtor parametrizado por uma String
```

A única forma que um programador tem de saber que um determinado método pode lançar uma excepção, é isso estar explicitamente reflectido na assinatura do método.

Para tal a assinatura do método deve ser feita de forma a indicar que o método pode "libertar alguma"excepção. Isso deve ser feito de acordo com a seguinte sintaxe:

```
public <tipoDados><nomeMetodo>(<listaParametros>)
throws <listaClassesExcepcao>
```

Um método que invoque outro que pode lançar uma excepção, deve precaver-se e ter disponível código de tratamento para essa excepção. Se não o fizer, então esse método deve explicitamente (na sua claúsula de throws) declarar que também pode libertar uma excepção desse tipo.

Note-se o exemplo de utilização de uma stack de Object com tamanho limitado. A operação de *push* deve verificar se ainda é possível adicionar mais algum elemento à stack. Caso esta já esteja cheia, não insere nada e sinaliza quem invocou o método de que aconteceu uma situação de excepção.

De igual forma, o método que faz *pop* de um elemento da stack, verifica primeiro se a stack não está vazia. Caso esteja, devove uma excepção.

```
public void push(Object elem) throws StackCheiaException {
  if (numElem == dim)
     throw new StackCheiaException("Stack Cheia !!");
  else
     { stack.add(elem); numElem++; }
}

public void pop() throws StackVaziaException {
  if (numElem == 0)
     throw new StackVaziaException("Stack Vazia !!");
  else
     numElem--;
}
```

3.2 Apanhar uma excepção

Como referido atrás ao invocar um método que pode lançar uma excepção, o programador deve prever a existência de código de tratamento do erro, por forma a que o programa não acabe de forma brusca, devido ao facto de a excepção não ter sido devidamente identificada e tratada.

A construção Java que permite apanhar as excepções e tratá-las tem a seguinte forma:

```
try{
  //código que liberta excepções
  ...
  ...
}
catch(<tipoExcepcao1>){
  //tratamento da excepcao
}
catch(<tipoExcepcao2>){
  //tratamento da excepcao
}
catch (...){...}

finally{
  //código que pode ser executado após as
  //clausulas de catch

}
```

Tenha em linha de conta que esta estrutura é muito semelhante a um switch, logo após entrar numa zona de tratamento a uma classe de excepção, sai da estrutura try/catch e só pode exectuar o código que estiver em finally.

A zona de finally é opcional e pode não existir.

O código que invoca os métodos de stack, vistos atrás, deve estar preparado para apanhar e tratar as excepções.

```
  try
    { stack1.push("anos");
      stack1.push(new Integer(20));
      stack1.push("tem");
      stack1.push("Maria");
      stack1.push("A");

      //...
      stack1.pop(); stack1.pop(); stack1.pop(); stack1.pop();
      stack1.pop();

      // ...
   }
```

```
catch(StackVaziaException e)
  { System.out.println("A Stack está vazia !!"); }
catch(StackCheiaException e)
  { System.out.println("A Stack está cheia !!"); }
}
```

4 Input e Output em Java

Em Java existe um conceito - stream - que abstrai as diferentes formas que podem assumir a leitura e escrita de dados.

Uma stream é uma abstracção para uma qualquer forma de entrada de dados ou escrita de dados. Permite que o programador se isole das particularidades de estar a escrever ou ler, de um ficheiro, de uma porta numa outra máquina, entre outras situações.

As streams podem ser catalogadas em dois grandes grupos:

- streams de caracteres (streams de texto)
- streams de bytes (streams binárias)

Do primeiro grupo interessa ao programador conhecer as sub-hierarquias do package java.io que se encontram delimitadas pelas classes `java.io.Reader` (para streams de leitura) e `java.io.Writer` (para streams de escrita).

4.1 Streams de texto

Para escrever e ler de streams de texto existem várias possibilidades. No entanto para simplificar a escolha e ter um bom desempenho sugere-se a utilização das classes:

- FileWriter - classe de conveniência para escrever ficheiros de texto. Oferece métodos normais como write para escrever sequências de caracteres, flush e close para gerir a disponibilidade do output.
- FileReader - comportamento idêntico ao da classe acima, mas para leitura.

A utilização destas classes para escrita e leitura de informação resultaria em código de baixo nível, uma vez que a gestão do interface com as fontes de informação teria de ser feita pelo utilizador.

Por isso utilizam-se classes com interface mais optimizado como as existentes em:

- PrintWriter - permite a escrita de texto, faz a gestão do buffer (logo optimizando os tempos de acesso) e disponibiliza métodos para a escrita de caracteres, arrays e strings.

  Exemplo de utilização de escrita em ficheiros de texto. No exemplo do Banco (visto nas aulas práticas), este método envia para ficheiro de texto a informação de todas as contas.

```
public void escreveEmFicheiroTxt(String filename) throws IOException {

  PrintWriter fich = new PrintWriter(filename);
  fich.println("------- Contas --------");
  for(Conta c: this.contas.values())
    fich.println(c.toString());

    fich.flush();
    fich.close();
}
```

- BufferedReader - permite a leitura de texto de uma stream de texto, efectuando a gestão do buffer de caracteres. Oferece métodos para ler eficientemente caracteres, arrays e linhas de texto.

  Para ler uma linha de texto, a classe `BufferedReader` tem um método importante: `readline()` que devolve a String retirada do ficheiro até ao fim da linha (até encontrar um newline).

  É necessário depois fazer o parsing dessa String para encontrar a informação nela contida. Se a String tiver separadores que se possam identificar é muito útil a classe `StringTokenizer`. Esta classe faz iterações sobre a String com recurso aos métodos `nextToken()` e `hasToken`.

  Exemplo de utilização:

```
try {

  String linha = ""; String txt = "";
  StringTokenizer st;
  String nome; int cod, quant, numCp;

  BufferedReader bin =
    new BufferedReader(new FileReader("fich.txt"));

  while (bin.ready()) { //enquanto houver que ler
    linha = bin.readline();
    //ir buscar a informacao de codigo
    //Codigo:
    st = new StringTokenizer(bin.readline(), "\t");
    txt = st.nextToken();
    cod = Integer.valueOf(st.nextToken().intValue();

    //Nome:
    st = new StringTokenizer(bin.readLine(),"\t");
    txt = st.nextToken();
    nome = st.nextToken();
```

```
        //...
        //...
        //...
    } catch(IOException e) {...}

    }
```

4.2 Streams binárias

Para escrever e ler sequências de bytes, guardando os valores em formato binário existem várias possibilidades. As sub-hierarquias de classes do package java.io que começam em OutputStream e InputStream disponibilizam várias classes para esse efeito.

Para eficientemente guardarmos objectos em formato binários precisamos apenas de utilizar as seguintes classes:

- FileOutputStream - permite criar uma stream de escrita, para escrever sequências de bytes.
- FileInputStream - uma instância desta classe serve para obter bytes de um File existente. A sua escolha deve ser feita sempre que se pretender ler bytes e não caracteres (caso em que se deveria utilizar uma instância de FileReader).

Tal como nas streams de texto, é necessário agora escolher classes que possibilitem um maior nível de abstracção na manipulação da escrita e leitura de bytes. Para tal utilizam-se as classes:

- ObjectOutputStream - escreve para uma stream binária objectos com toda a informação que permite posteriormente reconstruí-los. Para que seja possível a um objecto ser escrito a classe respectiva deve implementar a interface `Serializable`.

  Exemplo de utilização:

  ```
  FileOutputStream fos = new FileOutputStream("t.tmp");
  ObjectOutputStream oos = new ObjectOutputStream(fos);

  oos.writeInt(12345);
  oos.writeObject("Today");
  oos.writeObject(new GregorianCalendar());

  oos.close();
  ```

  O método mais importante desta classe é: `public final void writeObject(Object obj) throws IOException` , pois permite gravar um objecto de uma só vez.

  No projecto do Banco, o método para gravar em ficheiro de objectos uma instância de Banco, tem a seguinte forma:

```
public void gravaEmObjStream(String fich) throws IOException{

  ObjectOutputStream oout =
            new ObjectOutputStream(new FileOutputStream(fich));
  oout.writeObject(this);
  oout.flush(); oout.close();

}
```

- ObjectInputStream - uma instância desta classe lê dados e objectos gravados utilizando uma ObjectOutputStream. Só é possível ler objectos cujas classes implementem a interface Serializable.

  Exemplo de utilização:

```
FileInputStream fis = new FileInputStream("t.tmp");
ObjectInputStream ois = new ObjectInputStream(fis);

int i = ois.readInt();
String today = (String) ois.readObject();
GregorianCalendar date = (GregorianCalendar) ois.readObject();

ois.close();
```

  O método mais importante que esta classe oferece é: `public final Object readObject() throws IOException, ClassNotFoundException` , pois permite ler um objecto de uma só vez.

  Na maioria das vezes é vantajoso ler o conteúdo de um ficheiro de objectos no programa principal. Repare-se que no exemplo do Banco, o método de instância `gravaEmObjStream`, gravou uma instância de Banco (o this), pelo que não faz sentido nenhum que a leitura desse ficheiro seja também feita por um método de instância de Banco.

  A leitura deve ser feita no programa principal (normalmente no método `main`), como a seguir se exemplifica.

```
public static void main(String[] args) {

  //....

  Banco novoBanco;

  try {
    ObjectInputStream oin =
         new ObjectInputStream(new FileInputStream(filename));
```

```
    novoBanco = (Banco) oin.readObject();
    oin.close();
  }
  catch(IOException e)
    { System.out.println(e.getMessage()); }
  catch(ClassNotFoundException e)
    { System.out.println(e.getMessage()); }


   //...
}
```