Instituto Politécnico de Beja — Escola Superior de Tecnologia e Gestão

Guia Prático 5
Modularidade, Composição, Delegação, Diagramas de Classes e Listas de objectos

João Paulo Barros
Programação Orientada por Objectos
Licenciatura em Engenharia Informática
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Beja

22 de abril de 2019

Objectivos: modularizar código existente, (*refactoring*), aplicar o método para o desenvolvimento de software: utilizar programação conduzida por testes; reconhecer diagramas de classe e de sequência e a diferença sintática e semântica entre objectos e classes na linguagem UML; definir e utilizar listas dinâmicas de objectos para implementação de algoritmos de pesquisa e operações de acumulação. Utilizar ciclos *foreach*. Perceber e gerar métodos `equals`.

1 Uma estrutura para o modelo da máquina dispensadora

Vamos agora aperfeiçoar um pouco a nossa classe `Dispenser`. Dado que se trata de um modelo de uma máquina dispensadora é fácil verificar que tem pelo menos duas partes principais:

1. O mealheiro;
2. Os produtos para venda.

O resto são os botões que em rigor fazem parte da interface com o utilizador. Assim podemos já identificar os vários componentes principais do nosso programa:

- A interface com o utilizador que se divide em ***view*** e ***controller***;
- O código principal a que também se chama ***model***;
- O código de teste.

Model-View-Controller (MVC)

Os três componentes nos dois primeiros pontos constituem a arquitectura **MODEL-VIEW-CONTROLLER (MVC)**. Assim, iremos desenvolve programas com uma variante da arquitectura MVC e com código de teste para o *Model*. Esta estrutura será utilizada em (provavelmente) todos os programas que iremos fazer nesta unidade curricular. A mesma permite diminuir as dependências entre componentes diferentes do programa. Ou seja aumenta-se o **ENCAPSULAMENTO** (*coupling*). Tal consegue-se procurando que cada um desses componentes tenha uma interface mínima (exponha um mínimo de dados). Tal tem o nome de **OCULTAÇÃO DE INFORMAÇÃO** (*information hiding*).

encapsulamento

ocultação de informação

As duas partes da máquina (mealheiro e produtos) são na verdade parte do *Model* que é a parte que estamos a fazer e testar. À classe dos mealheiros iremos chamar `MoneyBox`. Assim, cada objecto `Dispenser` irá conter e utilizar um objecto da classe `MoneyBox`.

link

A Fig. 1 mostra um diagrama de objectos que ilustra este tipo de ligação (**LINK**) entre um objecto da classe `Dispenser` e um objecto da classe `MoneyBox`. Os triângulos dão nome à ligação no sentido em que apontam: o objecto `oneDispenser` contém o objecto `oneMoneyBox`; o objecto `oneMoneyBox` pertence ao objecto `oneDispenser`.

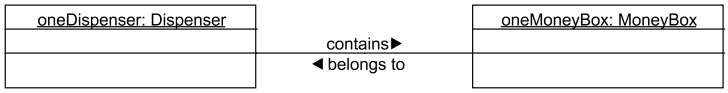


Figura 1: Diagrama de objectos que especifica uma relação de composição entre um objecto `oneDispenser` da classe `Dispenser` e um objecto `oneMoneyBox` da classe `MoneyBox`.

agregação

composição

Quando faz sentido dizer que um objecto é parte de outro, dizemos que existe uma **AGREGAÇÃO** entre os dois: um é o todo (o composto) e o outro é a parte (o componente). Na agregação, cada objecto contém uma ou mais referências para outros objectos. Quando para além de ser "parte de" o componente existe durante o mesmo tempo em que existe o composto (têm ciclos de vida coincidentes) então dizemos que se trata de uma relação de **COMPOSIÇÃO**. É o que sucede entre cada objecto da classe `Dispenser` e um objecto da classe `MoneyBox`. Técnicamente apenas a criação é coincidente pois em Java™ não temos controlo sobre a destruição dos objectos — o componente (o objecto da classe `MoneyBox`) é criado no construtor do composto (o objecto da classe `Dispenser`) — mas na prática o "final de vida" de ambos também coincide pois como apenas o objecto da classe `Dispenser` acede ao objecto da classe `MoneyBox` quando não há objecto `Dispenser` é já como se não houvesse objecto `MoneyBox`. A Fig. 2 mostra um diagrama de classes que ilustra este tipo de relação entre objectos da classe `Dispenser` e objectos da classe `MoneyBox`. O losango a cheio significa composição.

diagrama de classes

links

associações

O diagrama de objectos especifica objectos específicos. Normalmente, as relações entre objectos são as mesmas para todos os objectos dessa classe. Por essa razão é mais usual representar classes e as relações entre todos os objectos dessas classes. Tal é feito no chamado **DIAGRAMA DE CLASSES** (*class diagram*). Note que o diagrama de classes especifica relações entre objectos! Como são relações que são verdadeiras para todos os objectos de cada classe, cada relação representa-se por uma linha entre rectângulos que representam classes e modela as ligações entre os vários objectos. Assim, ligações (**LINKS**) entre objectos são modeladas por **ASSOCIAÇÕES**. Estas incluem as agregações e composições.

A composição entre objectos das classes `Dispenser` e `MoneyBox` pode ser lida da

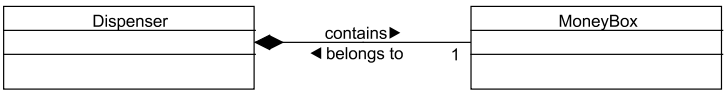


Figura 2: Diagrama de classes que especifica uma relação de composição entre cada objecto da classe `Dispenser` e um objecto da classe `MoneyBox`.

seguinte forma: cada objecto da classe `Dispenser` contém uma referência para um objecto da classe `MoneyBox`. Como foi dado dado um nome à associação entre `Dispenser` e `MoneyBox`, podemos ler de outra forma menos "tecnicista"e de mais alto-nível: *Cada objecto da classe `Dispenser` contains um objecto da classe `MoneyBox`.*

Tal como já referido, a composição é representada utilizando um losango a cheio e é implementada através de um atributo no objecto composto que é a referência para o objecto componente. Este é criado no construtor do composto.

A linguagem UML admite os seguintes indicadores de multiplicidade[1], entre os quais está o "1" utilizado na Fig. 2:

Indicador	Multiplicidade
`0..1`	zero ou um
`1`	um
`0..*`	zero ou mais
`1..*`	um ou mais
`n`	apenas n (com $n > 1$)
`*`	muitos
`0..n`	zero a n (com $n > 1$)
`1..n`	um a n (com $n > 1$)
`n..m`	n a m (com $n > 1$ e $m > 1$)
`n..*`	n ou mais (com $n > 1$)

2 Criação do mealheiro — o objecto da classe `MoneyBox`

Os testes já feitos deverão funcionar para objectos da nova classe Dispenser agora contendo cada um o seu mealheiro. A diferença estará apenas no interior da classe `Dispenser`. As estas alterações que não alteram as funcionalidades do código (do ponto de vista externo, este faz o mesmo) mas que melhoram a sua estrutura interna chamamos **REFACTORING** (refactorização). *refactoring*

Para efectuar o refactoring da classe `Dispenser` deverá seguir os seguinte passos:

1. Modificar o corpo dos métodos `insertCoin`, `buyProduct` e `cancel` na classe `Dispenser`; a modificação consistirá em chamar os métodos com o mesmo nome na classe `MoneyBox`, por outras palavras, os objectos `Dispenser` irão delegar as tarefas relacionadas com gestão de dinheiro no respectivo objecto da classe `MoneyBox` (*vide* **DELEGATION**); *delegation*
2. Definir a classe `MoneyBox`;
   (a) Mudar os atributos `productPrice`, `insertedMoney` e `salesMoney` para a classe `MoneyBox`;

[1] *in* Ambler, Scott W., "The Elements of UML 2.0 Style", Cambridge University Press, 2005, pág. 63.

(b) Definir os métodos `insertCoin`, `buyProduct` e `cancel`, na classe `MoneyBox`.

Segue-se a resolução para o caso do método `insertCoin` em ambas as classes e mais algum código:

```
1
class Dispenser
3 {
    private MoneyBox moneyBox;
5
    ...
7   public Dispenser(int productPrice)
    {
9       this.moneyBox = new MoneyBox(productPrice);
    }
11
    public insertCoin(int coin)
13  {
        return this.moneyBox.insertCoin(coin);
15  }
    ...
17 }
```

```
1
class MoneyBox
3 {
    private int insertedMoney;
5   private int salesMoney;
    ...
7   public MoneyBox()
    {
9       this.insertedMoney = 0;
        this.salesMoney = 0;
11  }

13  public insertCoin(int coin)
    {
15      this.insertedMoney += coin;
        return this.insertedMoney;
17  }
    ...
19 }
```

Verifique se o teste do método `insertCoin` continua a funcionar. Se não funciona, tente descobrir a razão.

Agora já deverá conseguir fazer o mesmo para os métodos `buyProduct` e `cancel`. Ou seja, deve fazer as alterações (na classe `Dispenser`) e adições (na classe `MoneyBox`) necessárias para que todos os testes passem.

3 Dados da Máquina

Considere agora que pretendemos guardar os dados relativos ao registo da máquina. Estes correspondem a um conjunto de informação sobre a própria máquina, nomeadamente, modelo, fabricante e data de fabrico. Em lugar de colocarmos todos estes dados como atributos de cada objecto `Dispenser` vamos considerar que a máquina **tem um** (em inglês ***has-a***) registo. E o que é um registo? Naturalmente, mais um objecto. Assim vamos necessitar de uma classe `Register`.

3.1 A classe `Register`

Esta será uma classe pouco interessante pois define objectos pouco interessantes. Isto porque, por agora, apenas contêm dados. Estes objectos `Register` não fazem grande coisa. Mesmo assim, estes objectos são muito frequentes pois muitas vezes é necessário guardar dados que depois são consultados e, eventualmente, alterados.

1. Defina então uma classe `Register` ainda vazia.
2. Defina uma classe de teste para esta classe e defina um pequeno teste que cria um objecto `Register` e verifica, utilizando um método *get* (um *getter*) se todos os atributos ficaram correctamente inicializados. Por exemplo:

```
1
class RegisterTest
3 {
    ...
5   public void testInit()
    {
7       Register r = new Register("XYZ" /*modelo*/,
                                  "Bolhas Company", 2006);
9       assertEquals("XYZ", r.getModel());
        // identico para os restantes atributos
11  }
}
```

3. Defina agora o construtor e os métodos *get* para cada um dos atributos.
4. Execute o teste para verificar se os *getters* funcionam correctamente.

3.2 `Dispenser` com `Register`

Vamos agora alterar a classe `Dispenser` de forma a que cada objecto desta utilize um objecto da classe `Register`.

1. Comece por alterar o teste da classe `Dispenser`, de forma a que o construtor aceite também um parâmetro `Register`. Por exemplo:

```
class DispenserTest
{
    ...
    public void testInsertCoin()
    {
        Register r = new Register("XYZ" /*modelo*/,
                                  "Bolhas Company", 2006);
        Dispenser d = new Dispenser(40 /*productPrice*/, r);
        int i = d.insertCoin(20);
        assertEquals(20, i);
    }
}
```

Após a execução deste código, que cria dois objectos da classe `String`, um objecto da classe `Register` e um objecto da classe `Dispenser`, as ligações entre os objectos podem ser representadas pelo diagrama da Fig. 3. Note que este não é um diagrama UML e omite o objecto da classe `MoneyBox`.

2. Agora altere o construtor da classe `Dispenser` de forma a que este passe no teste. Tal implica definir esse atributo. Por exemplo:

```
class Dispenser
{
    private Register register;
    ...

    public Dispenser(int productPrice, Register r....)
    {
        ...
        this.register = r;
        ...
    }
}
```

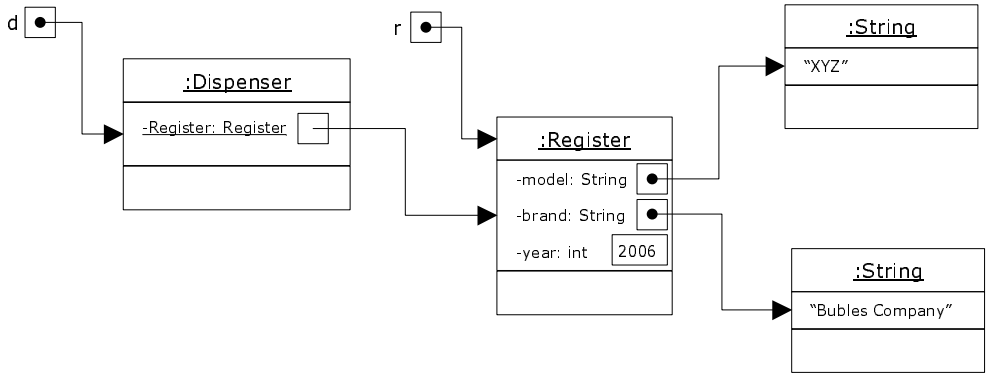


Figura 3: Objectos criados pelo método `testInsertCoin` e respectivas interligações.

Implementámos assim uma relação para a qual não faz sentido falar em "parte de": um registo não é parte da máquina, da mesma forma que o mealheiro. No entanto, um registo está obviamente associado à máquina.

3. Cada objecto da classe `Dispenser` pode utilizar o objecto `register`. Por exemplo, pode pedir ao objecto `register` para lhe dar o seu modelo. Desta forma podemos também pedir a um objecto da classe `Dispenser` que nos indique o seu modelo.

```
class DispenserTest
{
    ...
    public void testGetModel()
    {
        Register r = new Register("XYZ" /*modelo*/,
                                  "Bolhas Company", 2006);
        Dispenser dispenser = new Dispenser(40 /*productPrice*/, r);
        assertEquals("XYZ", dispenser.getModel());
    }
}
```

Para tal, o objecto da classe `Dispenser` **delega** esta tarefa no objecto `register` nele contido. Vamos então definir um método `getModel`, na classe `Dispenser`, que devolve o modelo da máquina. Este método permitirá que o teste acima tenha sucesso. Para tal o método pede ao objecto `this.register` essa informação chamando o respectivo método `getModel`:

```
class Dispenser
{
    ...
    public String getModel()
    {
       return this.register.getModel();
    }
}
```

4. Adicione os testes para os restantes atributos.

5. De forma a passar os testes definidos no ponto anterior, adicione funções idênticas para os restantes atributos da classe `Register`.

No exemplo acima, o objecto da classe `Dispenser` foi criado recebendo um objecto, previamente criado, da classe `Register`. Ficou assim com uma referência para um objecto que já existia.

O próximo passo será a modelação dos produtos à venda na nossa máquina. Para tal teremos de utilizar uma estrutura de dados que seja capaz de guardar vários objectos, por exemplo uma lista ou quadro.

4 Uma Lista de Produtos

Vamos continuar a desenvolver o programa proposto no guia anterior que simula uma máquina dispensadora. Agora vamos aproximar a nossa máquina mais um pouco da realidade. A máquina vende vários produtos pelo que teremos de guardar vários produtos na máquina. Assim, cada objecto `Dispenser` deve poder conter vários produtos. Para criar estes objectos teremos de definir uma classe que represente produtos. Vamos denominá-la `Product`. Depois, já poderemos criar produtos.

1. Defina então a classe `Product` de forma a que o seguinte teste seja bem sucedido. O teste especifica a forma de criar objectos da classe `Product` e testa se os atributos ficaram com os valores correctos. Naturalmente, deve também definir a classe `ProductTest` onde deve definir um teste à criação de objectos da classe `Product`.

```
class ProductTest
{
    ...
    public void testCreateProduct()
    {
        Product product = new Product(/*name*/ "Chocolat 50g", /*price*/ 100);
        assertEquals("Chocolat 50g", product.getName());
        assertEquals(100, product.getPrice());
    }
}
```

Agora necessitamos de uma lista de produtos. Poderíamos utilizar um quadro (*array*), mas os quadros têm uma característica que por vezes não é conveniente: têm sempre o mesmo tamanho. Além disso são objectos algo limitados pois não têm métodos. Uma verdadeira lista é um objecto muito útil pelo que a biblioteca da linguagem Java™ disponibiliza já classes que permitem criar listas dinâmicas, ou seja, listas cuja dimensão, a que corresponde a quantidade de elementos, muda ao longo do tempo. Na verdade, existe mais do que uma classe para representar listas, mas nós iremos falar apenas da mais utilizada e que se chama `ArrayList`. O seu nome completo é `java.util.ArrayList`. Podemos utilizar a forma abreviada (`ArrayList`) desde que coloquemos o `import` certo no início no ficheiro, antes da definição da classe:

```
import java.util.ArrayList;
```

classe genérica

A partir da versão 5 da linguagem Java™, a classe `ArrayList` é uma **classe genérica**. As classes genéricas, por vezes também denominadas **templates**, ou classes parametrizadas, são classes que recebem parâmetros. No caso que nos interessa, a classe `ArrayList` recebe como parâmetro a classe dos objectos que irão fazer parte da lista. Por exemplo, a classe para representar uma lista de `Strings` toma o nome de `ArrayList<String>`. Os símbolos < e > (sinais de *menor que* e *maior que*) funcionam aqui apenas como um tipo de parêntesis.

Veja também a documentação da classe (*vide* Fig. 4).

Na verdade, como iremos utilizar variáveis referência para listas (`List`), devemos ver a documentação da interface `List` (`http://docs.oracle.com/javase/8/docs/api/java/util/List.html`). Neste contexto, uma interface é um tipo especial de de classe que apenas inclui cabeçalhos de métodos. Assim, a interface `List` especifica quais os métodos que podemos pedir a objectos que sejam listas (por exemplo `ArrayLists`) para executar.

listas indexadas

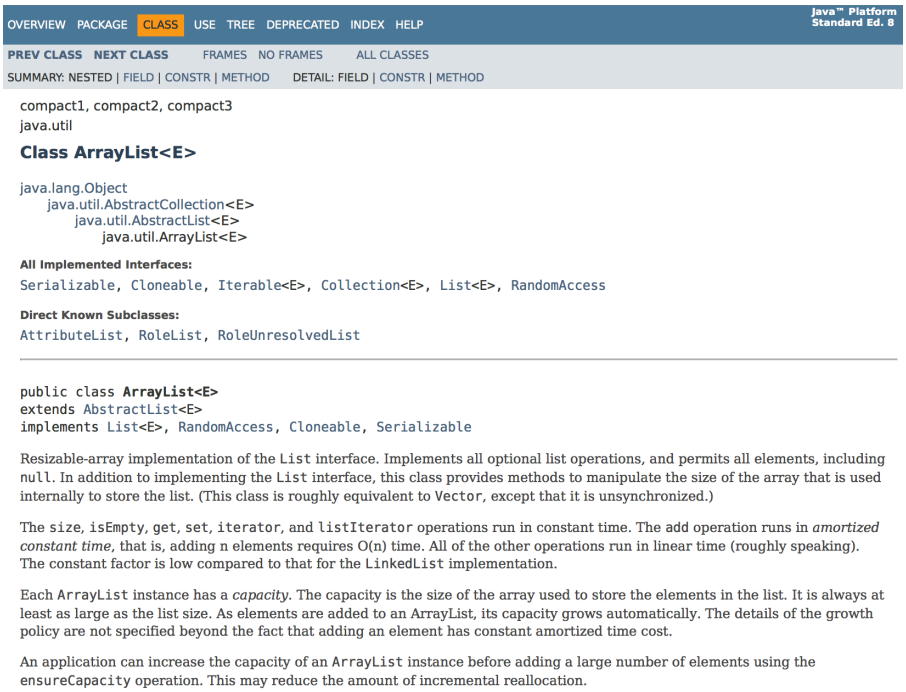
Estas listas são indexadas, ou seja, cada um dos objectos é colocado numa posição numerada. A numeração começa sempre em zero pelo que uma lista com um objecto tem apenas a posição zero ocupada. De forma semelhante, uma lista com dois objectos, tem as posições zero e um ocupadas. Note que tal implica que a última posição é sempre igual à quantidade de objectos menos um: por exemplo, três objectos estarão nas posições `0`, `1` e `2`, respectivamente.

Como os objectos `ArrayList` são listas (`List`), iremos definir variáveis referência para `List`. Estas variáveis podem guardar nomes (referências) de objectos de qualquer tipo de listas (que sejam `Lists`). Por exemplo:

```
java.util.List<Product> products = new java.util.ArrayList<Product>();
```

ou

```
// se tivermos colocado o import adequado:
List<Product> products = new ArrayList<Product>();
```



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

---

public class **ArrayList<E>**
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.

Figura 4: Documentação da classe `ArrayList`.

O seguinte método de teste, mostra o funcionamento do método `get`, da classe `ArrayList`, para objectos que são listas.

```
class ProductTest
{
    ...
     public void testProductList()
     {
         java.util.List<Product> products =
             new java.util.ArrayList<Product>();

         products.add(new Product(/*name*/ "Chocolat␣50g", /*price*/ 100));
         products.add(new Product(/*name*/ "Cookies", /*price*/ 150));

         assertEquals("Chocolat␣50g", products.get(0).getName());
         assertEquals(100, products.get(0).getPrice());
         assertEquals("Cookies", products.get(1).getName());
         assertEquals(150, products.get(1).getPrice());

         // needs an equal method to pass
         assertEquals(new Product(/*name*/ "Cookies", /*price*/ 150),
                      products.get(1)); // equals method needed!!     }
    ...
}
```

A lista é criada vazia, ou seja sem objectos (*vide* Fig. 5).

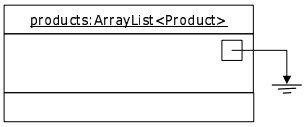


Figura 5: Estado da lista `products` definida no método `testProductList` imediatamente após a criação do objecto (primeiro **new**).

Seguidamente, os objectos são adicionados utilizando o método `add`, o qual adi-

ciona, no final da lista, o objecto que recebe por parâmetro . Note que, em rigor, o método adiciona, no final da lista, uma referência para o objecto. Isto porque cada objecto da classe `ArrayList<Product>` é uma lista de referências para objectos da classe `Product`. A Fig. 6 ilustra um objecto da classe `ArrayList<Product>` com `N` objectos da classe `Product`. O valor de `N` pode ser obtido pelo método `size` (`products.size()`) que devolve quantos elemento se encontram na lista (cujo nome está em `products`).

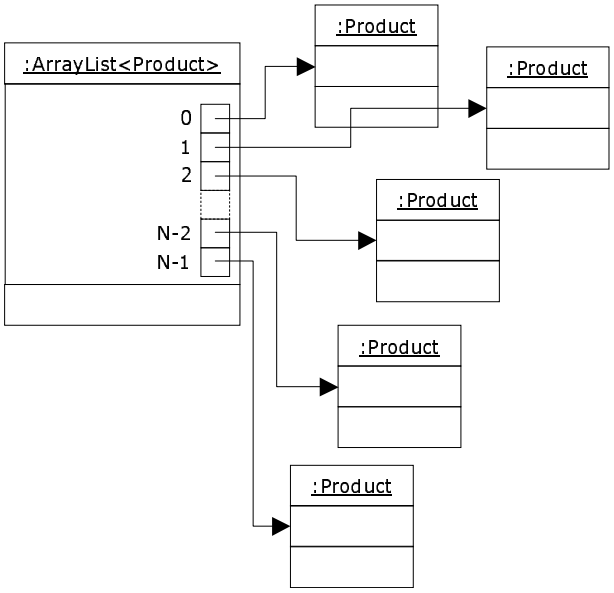


Figura 6: Representação de um `ArrayList<Product>`.

A Figura 7 ilustra a lista definida no método `testProductList` para a classe `Product`.

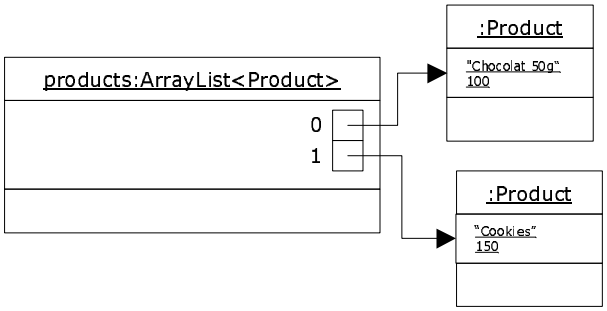


Figura 7: Representação da lista `products` definida no método `testProductList`.

Também é utilizado o método `get`, que permite obter a referência (para um objecto) guardada na posição dada pelo respectivo parâmetro. Assim, a instrução `products.get(1)` devolve a referência para o objecto da classe `Product` guardada na posição 1 da lista e que corresponde a um pacote de "Cookies"que custa 150 cêntimos.

5 Método `equals`

Para que o código apresentado funcione correctamente, nomeadamente o último `assertEquals` no método `testProductList`, necessitamos ainda de definir o método `equals` na classe `Product`. O método define quando é que dois objectos da classe `Product` devem ser considerados iguais. Tipicamente, tal corresponde a comparar os atributos, ou seja, dois produtos são iguais quando têm o mesmo nome e o mesmo preço. No entanto, antes de compararmos os valores dos atributos, vamos verificar se não estamos por acaso a comparar o objecto com ele próprio e se o objecto com que queremos comparar é realmente da classe `Product`. O método é então o seguinte (note que o cabeçalho deve ser o indicado, com um parâmetro da classe `Object`). Leia a explicação após o código juntamente com este.

```
public boolean equals(Object obj)
{
   if (obj == this) // comparing the object with itself!
   {
     return true;
   }
   else if (!(obj instanceof Product)) // not a Product!
   {
      return false;
   }
   else
   {
      Product p = (Product)obj;
      return this.getPrice() == p.getPrice() &&
             this.getName().equals(p.getName());
   }
}
```

O método devolve um valor booleano que indica se os dois objectos da classe `Product` são iguais. Para tal perguntamos a um objecto (e.g. `obj1`) se ele é igual a outro (e.g. `obj2`). Assim, a sintaxe é a seguinte: `obj1.equals(obj2)`. Claro que também podemos perguntar ao objecto `obj2` se ele é igual ao objecto `obj1`, caso em que escreveremos `obj2.equals(obj1)`.

O método apresentado corresponde a um formato recomendado e que pode ser visto como uma receita para fazer métodos `equals`. Funciona em três fases e vamos explicá-las utilizando o pedido a um objecto `obj1`, mais concretamente `obj1.equals(obj2)`:

1. O método testa se as referências `obj2` e `obj1` referenciam um mesmo objecto, ou seja se são duas variáveis referência que contêm o mesmo nome (referência) de objecto. Se assim for, podemos devolver **`true`** terminando o método, pois um objecto é sempre igual a ele próprio.

2. Se as referências são diferentes, vamos agora verificar se o `obj2` é realmente um `Product`. Tal é feito utilizando o operador **`instanceof`**. Se não é um `Product` devolvemos **`false`**.

3. Finalmente, se as referências são diferentes e o `obj2` é um `Product` podemos então comparar ambos os objectos. Nesse caso, começamos por criar uma referência para `Product` a partir da referência `obj2`; na verdade é uma referência para o mesmo objecto mas agora do tipo `Product` o que nos permite utilizá-la para efectuar pedidos a objectos da classe `Product`. Com ela comparamos os vários atributos de ambos os objectos, que para objectos da classe `Product` são `name` e `price`.

Duas regras:

1. Utilizamos o operador == para comparar valores de tipos primitivos (que não são objectos): **`long`**, **`int`**, **`short`**, **`byte`**, **`double`**, **`float`**, **`char`**, **`boolean`**.
2. Utilizamos o método `equals` para comparar objectos. Nos objectos da classe `Product` tal aplica-se aos atributos `name` que são objectos da classe `String`.

Note que o método `equals` também está definido na classe `String` e é a forma correcta de comparar `Strings`. É esse método `equals` para objectos da classe `String` que é utilizado para comparar os nomes (atributos `name` devolvidos pelo método `getName`) na expressão lógica cujo valor é devolvido: **`this`**`.getName().equals(p.getName())`.

As boas notícias estão no facto do IntelliJ ser capaz de gerar o método `equals`. Este é gerado simultaneamente com o método hashCode pois ambos são necessários em diversos contextos.

Aqui vamos fazer uma pausa para apresentar o ciclo `foreach` e rever os restantes ciclos disponíveis na linguagem Java™.

6 Utilização da classe `ArrayList`

Adapte todos os testes que definiu no guia prático anterior de forma que ainda funcionem com a classe `Dispenser` com um `ArrayList` de productos (objectos da classe `Product`.

Para definir os métodos que suportam estas novas funcionalidades pode consultar o código sobre operações em `ArrayList`.

Para começar, criamos uma máquina sem produtos. Ou seja, com uma lista vazia de produtos:

```
class Dispenser
{
    private Register register;
    private int moneyReceived;
    private List<Product> products;
    ...

    /**
     * Creates an empty dispenser with
     * no received money and no selected location
     */
    public Dispenser(Register r)
    {
        this.register = r;
        this.moneyReceived = 0;
        this.products = new ArrayList<Product>();
    }

    ...

}
```

Este construtor cria uma lista vazia para colocar `Products`.

Vamos agora definir o método `addProduct` que deve permitir colocar produtos na máquina, ou seja, objectos na lista `products`:

```
/** Adds a new product to be sold by the dispenser
 *  @param product product to be inserted
 */
public void addProduct(Product product)
{
    this.products.add(product);
}
```

7 Ciclos `while`, do `while`, `for` e *foreach*

A repetição de instruções é extremamente comum. Especialmente a repetição de instruções semelhantes mas não iguais está na base de muitíssimos algoritmos. Basta pensar, por exemplo no algoritmo de soma que todos aprendemos nos primeiros anos de escola: somar as unidades, guardar o transporte, somar as dezenas mais o transporte e guardar o transporte, somar as centenas mais o transporte e guardar o transporte, etc..

É usual considerar que as repetições de instruções se dividem em três grupos:

0 ou mais vezes As instruções podem não ser executadas ou ser executadas uma quantidade de vezes que não podemos prever quando escrevemos o código. Em Java™, tal é especificado com um ciclo **`while`** o qual tem a seguinte sintaxe:

```
while (\emph{condition})
{
        ...
}
```

1 ou mais vezes As instruções são sempre executadas uma vez. Depois são executadas uma quantidade de vezes adicional que não podemos prever quando escrevemos o código. Em Java™, tal é especificado com um ciclo **`do while`** que tem a seguinte sintaxe:

```
do
{
        ...
} while (condition);
```

***n* vezes** As instruções são sempre executadas *n vezes*. Deve escolher este tipo de ciclo quando pretende que as instruções sejam executadas uma quantidade de vezes predeterminada. Em Java™, tal pode ser especificado de duas formas:

for é uma abreviatura do ciclo **`while`**. Normalmente é utilizado para processar listas indexadas dado que permite indicar quais os índices e portanto permite escolher quais os elementos que queremos processar. Tem a seguinte sintaxe:

```
for(inicializacao; condicao; incremento)
{
        ...
}
```

foreach Permite repetir uma acção para cada elemento de uma lista. Por exemplo, podemos escrever no ecrã o nome de cada produto numa lista de produtos (`products`), utilizando o seguinte código:

```
for(Product p : products)
{
    System.out.println(p.getName());
}
```

Note que a palavra reservada utilizada é a palavra `for`. Não existe nenhuma palavra *for each* mas é esse o nome dado a este tipo de ciclo dado que corresponde ao conceito: *para cada elemento da lista fazer qualquer coisa.*

O diagrama na Fig. 8 exemplifica o funcionamento do ciclo **`while`** e **`for`**. Ambos os ciclos escrevem o dobro de cada um dos números entre `0` e `n - 1`. Note que são apenas sintaxes diferentes para uma mesma sequência de instruções.

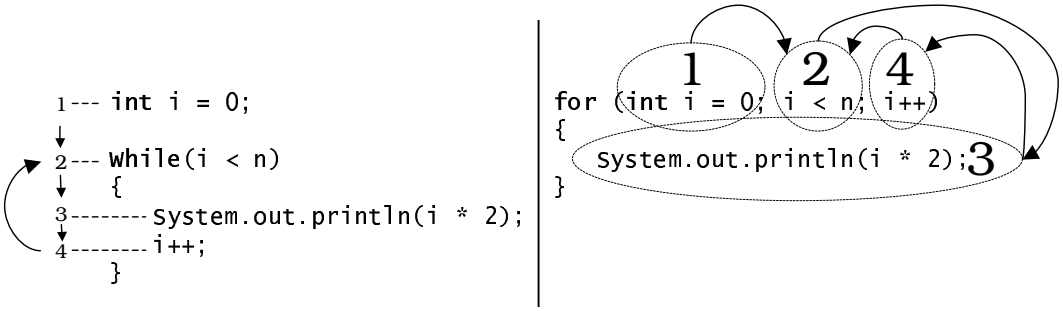


Figura 8: Funcionamento dos ciclos **`while`** e **`for`**.

A secção seguinte apresenta uma lista de novos requisitos.

8 Requisitos adicionais

Considere o suporte para cada um dos seguintes novos requisitos. Para cada um deve definir um teste e depois o código que permite que o teste passe.

1. O administrador obtém a informação de qual o produto mais caro na máquina.
2. O administrador obtém a informação de qual o produto mais barato na máquina. Para tal deve percorrer todo os elementos da lista de forma a escolher o que tem um preço mais baixo. Para tal, o método que implementa este serviço deve utilizar um ciclo *foreach*.
3. O administrador obtém a informação de qual a média de preços dos produtos na máquina. A média deve ser um número real pelo que o respectivo método deve também devolver um número real.
4. O administrador obtém a lista dos nomes dos produtos na máquina. Esta lista deve ter um nome por linha e deve ser devolvida como um objecto da classe `String`:

(a) Definir uma nova *string*, ainda vazia.

(b) Percorrer a lista existente. Para cada elemento, adicionar o seu nome à nova *string*.

(c) Depois de terminado o percurso na lista existente, o método deve devolver a nova *string*.

5. O administrador obtém a lista dos produtos que custam menos do que um dado preço. Para realizar este método necessita de criar uma nova lista. Deve seguir os seguintes passos:

   (a) Definir uma nova lista de produtos, ainda vazia.

   (b) Percorrer a lista existente. Para cada elemento, se o elemento tem um preço menor do que o preço recebido como parâmetro então esse produto deve ser adicionado à nova lista.

   (c) Depois de terminado o percurso na lista existente, o método deve devolver a nova lista.

Deve terminar a resolução deste guia fora das aulas. Traga as dúvidas para a próxima aulas ou coloque-as no fórum de dúvidas da disciplina. As sugestões para melhorar este texto também são bem-vindas.