Instituto Politécnico de Beja — Escola Superior de Tecnologia e Gestão

Guia Prático 6
Herança, Interface, Comparação de Objetos e Ordenação

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

6 de maio de 2019, revisto em 21 de maio

Objectivos: Definir classes que herdam de uma classe comum; diferenciar entre classes abstractas e concretas; implementar interfaces; aplicar o conceito de classe canónica; comparar objetos.

Conteúdo

Neste guia deve partir da resolução do Guia Prático 5 (máquina de vendas). A resolução deve ser a sua. Se não tem nenhuma deve primeiro fazer uma. Para tal pode partir da que está disponível na página da unidade curricular.

1 Mais de um tipo de produto

O objectivo é o de permitir o tratamento de vários tipos de produtos, cada um com características diferentes. Tal implica a necessidade de especificar diferentes dados para cada tipo de produto. Tal também pode acarretar a necessidade de cada tipo de produto efectuar a mesma operação de forma diferente consoante o tipo de dados que guarda. Para já vamos necessitar de considerar dois tipos de produtos: discos e livros. Naturalmente, a cada um destes deve corresponder uma classe. Teremos então a classe `Disc` e a classe `Book`.

O programa dado pode evoluir para a substituição da classe `Product` pelas classes `Disc` e `Book`. Note que tal implica criar duas novas listas em cada objeto da classe `Store` as quais irão substituir a lista de produto o que origina muito código repetido. Essa modificação fica como exercício opcional. Para já vamos avançar para a secção seguinte.

2 Utilização de herança — a relação *é-um (is-a)*

A alteração referida na secção anterior origina uma grande duplicação de código. Ainda por cima, essa repetição de código aumenta sempre que adicionamos um novo tipo de produtos. O problema está no facto de todos os tipos de produtos discos, livros ou outros terem muito em comum porque embora sendo de tipos diferentes são também de um só tipo mais genérico: o tipo *produto*. Por exemplo, um livro é do tipo livro mas este tipo é um produto e por isso um livro é também um produto. Este tipo de relação designa-se por "é-um" ou, em inglês, *is-a*. É especificada por uma relação de **herança**. Em Java™ utiliza-se a palavra reservada extends para indicar a herança entre classes:

```
1  public class Book extends Product    // each Book is a Product
   {
3    private String publisher;
     private List<Author> authors;
5
     public Book(String name, int price,
7                String publisher, List<Author> authors)
     {
9              super(name, price); // call construtor in superclasse (Product)

11             this.publisher = publisher;
```

```
            this.authors = authors;
13          ...
    }
15      ...
}
17
public class Disc extends Product
19 {
        private int seconds;
21      private List<Author> authors;

23      ...
}
```

1. Defina a classe `Author`. Esta classe destina-se apenas a guardar informação sobre cada autor.
2. Partindo da resolução fornecida, adicione as classe `Book` e `Disc` ao programa *DispenserMachine*. Note que a classe `Product` NÃO tem de ser alterada.
3. Verifique que os testes anteriores continuam a funcionar.
4. Adicione o código de teste para o método `toString` em todas as classes que são produtos. Estes métodos devem mostrar o conteúdo do respectivo objeto.
5. Adicione métodos de teste na classe `DispenserTest`, que efectuam testes idênticos aos já realizados para objetos da classe `Product`, mas agora a objetos das classes `Book` e `Disc`. Note que irá necessitar de cria uma lista de autores.

Deverá obter um estrutura idêntica à do diagrama na Fig. 1.

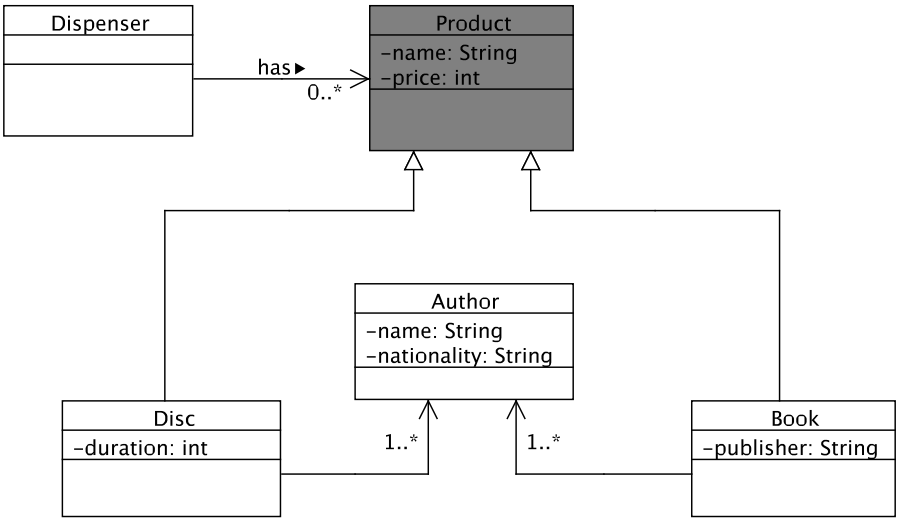

Figura 1: Diagrama de classes após adição das classes `Book` e `Disc`

3 Mais classes

Note que ambas as classes utilizam uma lista de autores. Tal acarreta um conjunto de repetições no código. Um forma de evitar essas repetições é considerar que existe um tipo de produtos que têm todas as características dos produtos mais uma lista de autores. Vamos chamar-lhe: `ProductWithAuthors`.

As classes `Book` e `Disc` passam a herdar desta nova classe. Esta nova classe herda da classe `Product`.

1. Adicione a classe `ProductWithAuthors` e modifique as classes `Book` e `Disc` de forma a que estas herdem daquela.
2. Verifique que os métodos de teste definidos na secção anterior continuam a funcionar.

4 Mais produtos

Considere agora os seguintes novos tipos de produtos: câmara (*Camera*), jogo (*Game*), DVD e CD. Um CD e DVD são discos (*Disc*). E um disco é um produto com autor. Por outro lado, um jogo e uma câmara são produtos. A Fig. 2 ilustra estas relações entre os vários tipos de produto.

1. Adicione métodos de teste na classe `StoreTest`, que efectuam testes idênticos aos já realizados, mas agora a objetos das classes `Camera`, `Game`, `Video` e `CD`.
2. Adicione o código de teste para o método `toString` em todas as classes que são produtos. Estes métodos devem mostrar o conteúdo do respectivo objeto.
3. Note que as classes a cinzento não são verdadeiros produtos. Altere os teste de forma a utilizar apenas os tipos de produtos das classes a amarelo, as que correspondem a "verdadeiros" produtos.

Deverá obter o diagrama de classes na Fig. 2.

Definir classes abstractas e concretas. Definição métodos abstractos. Definição e implementação de interfaces.

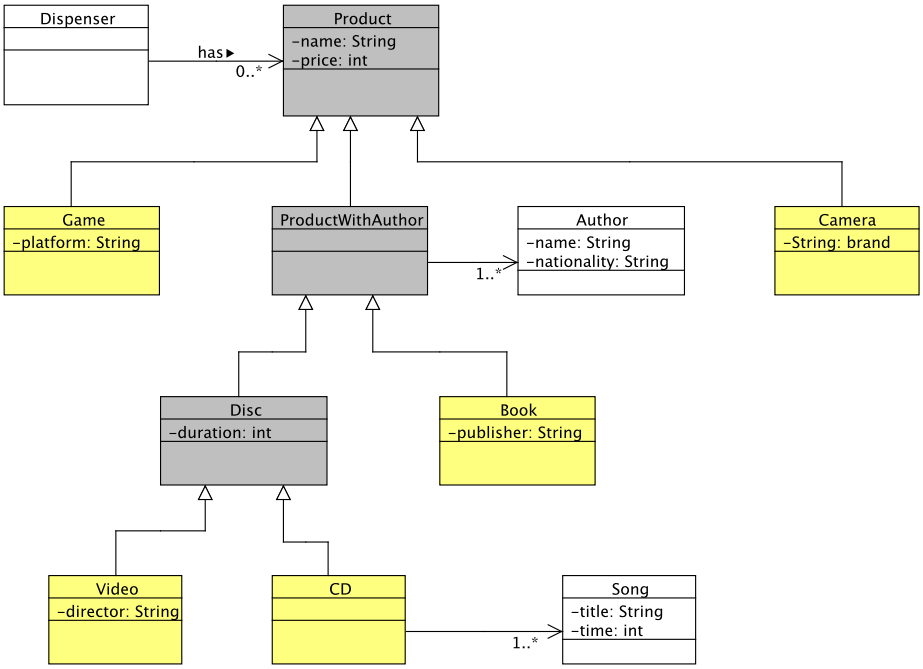

Figura 2: Diagrama de classes a obter

5 Classes abstractas

It makes sense to create a Wolf object or a Hippo object or a Tiger object, but what exactly is an Animal object? What shape is it? What color, size, number of legs...
Trying to create an object of type Animal is like a ***nightmare Star Trek™ transporter accident****. The one where somewhere in the beam-me-up process something bad happened to the buffer.*

– *in* Head First Java™, 2nd edition – By Kathy Sierra and Bert Bates

Algumas classes não correspondem a entidades para as quais necessitemos de criar objetos. Por exemplo, a partir do momento que definimos classes distintas para diferentes tipos de produto, a classe `Product` continua a ser útil como super-classe, pois serve para identificar todas as classes derivadas como sendo do tipo `Product` e para lá colocar atributos e métodos que devem ser comuns a todos os tipos de produto. No entanto, não faz já sentido criar objetos da classe `Product` em lugar de um `Book`, `CD`, `Camera` etc.. O que seria um `Product`? Na verdade, seria algo abstracto que não corresponde a nenhum objeto "concreto", tal como o "Animal" da citação: agora já não há produtos, mas sim livros, cds, câmaras, etc.. Assim, a classe `Product` deve passar a ser uma **classe abstracta**, ou seja, uma classe a partir da qual não podemos criar objetos.

Eis tudo o que nos interessa sobre classe abstractas (as que têm a palavra **`abstract`** na sua declaração):

1. Não podemos criar objetos de uma classe abstracta.
2. Uma classe que não é abstracta, diz-se **classe concreta**, mas tal não implica

qualquer especificação: por omissão, uma classe é concreta. Ou seja, se nada dissermos, uma classe é concreta podemos criar objetos dela.

3. As classes abstractas podem ou não ter métodos sem corpo (apenas com o cabeçalho). Estes métodos denominam-se **métodos abstractos** e são declarados acrescentando o modificador **`abstract`**.

4. Uma classe tem um método **`abstract`** se o declara ela própria ou se herda uma declaração da sua superclasse. Se uma classe tem um método **`abstract`** então a classe tem de ser definida como **`abstract`**.

5. Uma classe concreta tem de definir cada um dos métodos **`abstract`** que herda das suas superclasses.

6. Uma classe que não defina algum método **`abstract`** herdado, tem obrigatoriamente de ser abstracta.

7. Os métodos **`abstract`** não podem ser **`static`** (pois esses são da classe e não dos objetos). Também não podem ser private, por serem invísiveis também na subclasse, nem **`final`**, pois não podem ser redefinidos na subclasse. Qualquer destes casos impediria a redefinição do método numa subclasse.

Agora já pode melhorar o programa do guia prático anterior. Mais especificamente, modifique-o de acordo com os seguinte pontos:

1. As classes `Product`, `ProductWithAuthors` e `Disc` devem passar a ser classes abstractas. Depois, verifique se os testes ainda funcionam. Se não funcionarem será provavelmente porque estava a definir objetos dessas classes. Nesse caso, deverá definir apenas objetos das classes concretas.

2. Vamos admitir que a fórmula de cálculo do imposto pode ser diferente em produtos diferentes mas que todos (os produtos concretos) devem definir a sua fórmula de cálculo. Para tal, começamos por adicionar um método **`abstract`** na classe `Product`, para cálculo do imposto a pagar em cada produto:

```
2 class Product
  {
4     ...
      public abstract int computeTax();
6     ...
  }
```

3. Adicione a definição do método `computeTax` nas respectivas classes, de acordo com as seguintes informações:

   (a) O imposto para câmaras é 20% do respectivo preço.

   (b) O imposto para livros é 10% do respectivo preço.

   (c) O imposto para discos dependem da duração destes! Um disco paga 10% de imposto por cada hora de duração. Por exemplo, se um disco tem menos de uma hora (`duration` < 3600 segundos), o imposto é zero; se o disco tem entre 3600 segundos e 7199 segundos o imposto é de 10% do preço; se o disco tem entre 7200 segundos e 10799 segundos o imposto é de 20% do preço. No cálculo, deve utilizar uma fórmula geral que permita obter o imposto para qualquer duração. Em particular, a resolução não deve utilizar `ifs` nem `switch` nem o operador `?:`.

4. Defina um método `public abstract int priceWithTax()` que permite saber o preço de um dado produto já com o imposto incluído.

5. Modifique a classe `DispenserTest` de acordo com os seguintes pontos:

   (a) A classe `DispenserTest` deve passar a ter um método `public void setUp()` onde é criado um objeto `Dispenser` contendo uma lista de produtos com um produto de cada tipo. Para tal deve definir como atributos da classe `DispenserTest`, um atributo da classe `Dispenser` e os restantes que considere necessários:

```
1  public class DispenserTest extends TestCase
   {
3      private Dispenser d;
       private Product p1, p2, p3;
5      private List<Author> authors;

7      public void setUp()
       {
9          this.s = new Dispenser();

11         this.authors = Arrays.asList(new Author(...),
                                        new Author(...),
13                                      ...);
           this.p1 = new Book("chocolate", ...);
15         ...
       }
17
       ...
19
   }
```

   (b) Defina mais um teste na classe `DispenserTest` para verificar que o imposto de cada um dos produtos é calculado correctamente. Para tal deve percorrer a lista de produtos no objeto da classe `Dispenser`, cujo nome se encontra no atributo `d`, e verificar se o valor devolvido pelo método `priceWithTax` está sempre correcto.

6 Interfaces

> *The fundamental unit of programming in the Java programming language is the class, but the fundamental unit of object-oriented design is the type. While classes define types, it is very useful and powerful to be able to define a type without defining a class. Interfaces define types in an abstract form as a collection of methods or other types that form the contract for that type. Interfaces contain no implementation and you cannot create instances of an interface. Rather, classes can expand their own types by implementing one or more interfaces. An interface is an expression of pure design, whereas a class is a mix of design and implementation.*
>
> – *in* THE Java™ Programming Language, Fourth Edition –
>
> By Ken Arnold, James Gosling, and David Holmes

Em Informática, a palavra interface tem múltiplos significados. Aquele que nos interessa agora é o seguinte: uma **interface** define um tipo, mas de um modo abstracto, sem definir uma classe. Quer as classes quer as interfaces definem tipos, mas as interfaces limitam-se a especificar quais as operações que esse tipo deve

suportar. Não incluem dados nem especificam como é que as operações são efectuadas. Tal corresponde a declarar apenas os cabeçalhos dos métodos que têm de ser implementados.

Na definição de interface que iremos considerar, uma interface pode ser vista como uma classe abstracta com três características:

1. Não tem atributos;
2. Não tem construtores;
3. Todos os métodos são abstractos, e portanto não têm corpo.

Uma interface apenas especifica uma lista de métodos que as classes "implementadoras" (concretas porque não podem ser abstractas) terão de definir. Tal como referido na citação, uma interface é *design* puro, enquanto que uma classe é *design* e implementação.

Cada interface permite garantir que todas as classes que a implementam definem de certeza os métodos declarados na interface.

Uma classe pode "herdar" de uma ou mais interfaces. A este tipo de herança chamamos **implementação (da interface)**. A implementação corresponde à definição de todos os métodos declarados na interface. Ou seja, uma classe que implementa uma interface tem de incluir as suas definições para todos os métodos declarados na interface. Uma classe pode implementar zero ou mais interfaces. Tal significa que uma interface pode ser implementada juntamente com outras interfaces por uma mesma classe.

Vejamos agora um exemplo de uma interface e de uma classe que a implementa:

```
public interface DeliverableByMail
{
        int deliveryCost(); // sempre public
}

public class Camera extends Product implements DeliverableByMail
{
    ...........
    public int deliveryCost() // implementation of method in interface
    {
        return this.getPrice() * Camera.DELIVER_PERCENT;
    }
}
```

7 Um pouco sobre a classe *Object*

Uma classe que não herda de uma outra classe explicitamente, com a palavra extends, herda implicitamente da classe `Object` (`https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`).

A classe `Object` contém vários métodos e respectivas implementações. Desses interessam-nos especialmente três — `equals`, `hashCode` e `toString`:

`boolean equals(Object obj)` O método `contains` dos objetos da classe `ArrayList` deve devolver `true` quando lhe passamos um objeto igual a um que já lá está:

```
List<Position> lst = new ArrayList<Position>();
lst.add(new Position(2, 234));
assert(lst.contains(new Position(2, 234));
```

No entanto, para que a condição no `assert` seja verdadeira, a classe `Position` tem de definir um método `boolean equals(Object obj)` que efectivamente defina quando é que dois objetos da classe `Position` devem ser considerados iguais. Caso contrário, a comparação que o método `contains` efectua será feita utilizando um método `equals` que se limita a comparar os valores das referências de ambos os objetos. Como no nosso exemplo tempos dois objetos distintos (embora de igual conteúdo) o resultado seria `false`, pois dois objetos têm sempre um nome (ou referência) diferente. Sem a definição do método `equals`, aconteceria o seguinte:

```
List<Position> lst = new ArrayList<Position>();
Position p = new Position(2, 234);
lst.add(p);
assert(lst.contains(new Position(2, 234)) == false);
assert(lst.contains(p));
```

Ou seja, apenas quando perguntássemos ao objeto `lst` se contém o **mesmo** objeto que lá foi inserido, obteríamos uma resposta afirmativa. Um objeto, ainda que igual não teria resposta positiva e isto porque não tínhamos definido quando é que dois objetos da classe `Position` são iguais. Por outras palavras, não tínhamos definido o método `equals`.

`int hashCode()` Voltaremos a este tema mais tarde. Para já importa apenas saber que o método `int hashCode()` deve estar definido para que os objetos da nossa classe possam ser utilizados como chaves de `Maps` ou como elementos de
`Sets`. Os valores devolvidos pelos `hashCode` de cada um dos objetos para os quais o "equals"é **`true`** devem ser iguais.

`String toString()` É sempre boa ideia definir o método `String toString()`. Com este método não são necessários cuidados especiais. Como regra, devemos apenas garantir que devolve uma `String` que contém o valor de todos os atributos.

8 *Comparable* e classes canónicas

classe canónica

Com base na nomenclatura utilizada por Bergin (`http://csis.pace.edu/~bergin/patterns/CanonicalJava.html`), se uma classe redefinir os três métodos referidos herdados da classe `Object`, e implementar a interface `Comparable<T>` dizemos que a mesma é uma **classe canónica** . Ou seja, uma classe canónica é uma classe com todos os "elementos base": um conjunto de métodos que podem fazer falta, quer directamente quer para utilizar código que já está feito (por exemplo, o método `contains` na classe `ArrayList`). Implementar a interface `Comparable<T>` (`https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`) implica definir o método `int compareTo(T o)`.

O método `int compareTo(T o)` é necessário para que os objetos (da classe `T`) possam ser ordenados. Em particular, poderão ser elementos de um objeto da classe `TreeSet<T>`, uma *Collection* que mantém os seus objetos ordenado.

A interface `Comparable` é uma interface genérica pelo que a sintaxe deve ser a do seguinte exemplo:

Listagem 1: Classe Product comparável

```
1  public class Product implements Comparable<Product>
   {
3      ...
       /**
5      * Orders by name and then by price
       *
7      * @param otherProduct product to order
       * @return -1, 0 or 1
9      */
       public int compareTo(Product p)
11     {
           int cn = this.getName().compareTo(p.getName());
13         if (cn == 0) // names are equal
           {
15             return new Integer(this.getPrice()).compareTo(p.getPrice());
           }
17         else
           {
19             return cn;
           }
21     }
   }
```

9 Ordenação

Como vimos, um método `compareTo` define uma *relação de ordem* entre objetos.

Resumida e informalmente, tal significa que temos de definir (no método `compareTo` numa dada classe) quando é que queremos que um objecto seja considerado menor do que outro dessa mesma classe.

Segue-se uma explicação mais formal.

Tipicamente queremos que seja uma **relação de ordem parcial** . Formalmente, dado um conjunto A e uma relação binária[1] R sobre A ($R \subseteq A \times A$), dizemos que a mesma é uma relação de ordem parcial quando é reflexiva, anti-simétrica e transitiva:

reflexiva Todo o objeto está relacionado consigo mesmo $\forall x : R(x,x)$

anti-simétrica Se um objeto é "maior" e também "menor" do que outro então é porque é o mesmo objeto: $\forall x, y : (xRY) \wedge (yRx) \implies x = y$

transitiva Se y é "maior"do que x e z é maior do que y então z é maior do que x: $\forall x, y, z : xRy \wedge yRz \implies xRz$

Existindo um método que permite comparar objetos, torna-se possível ordená-los pois uma ordenação resulta na verdade da aplicação sucessiva de comparações entre objetos até se atingir uma sequência ordenada, ou seja uma sequência em que todos os objetos (elementos) adjacentes respeitam a relação binária R. Formalmente, uma sequência s de comprimento n, $s = (a_1, a_2, \ldots, a_n)$, está ordenada segundo uma relação de ordem R quando $\forall 1 \leq i \leq n : a_i R a_{i+1}$

Em Java, para ordenar objetos devemos definir o método `compareTo` mas depois não o devemos utilizar de forma direta. Ou seja, embora o possamos fazer, tipicamente não queremos fazer algo como `product1.compareTo(product2)` para ordenar objetos. Isto porque já existe código para ordenar e esse código é que utiliza o método `compareTo`. Nós vamos utilizar esse código que já existe feito. Vamos então ver alguns exemplos de como podemos ordenar produtos. A ordenação definida pelo método `compareTo`, ou sejam por implementação da interface `Comparable`, chama-se **ordem natural**.

9.1 Inserção num TreeSet

Os objetos inseridos num `Set` do tipo `TreeSet` ficam automaticamente ordenados pois essa é uma caractarística do `TreeSet`. Mas para tal o `TreeSet` tem de saber a relação de ordem, ou seja qual a definição do método `compareTo` que ele utiliza para saber como ordenar. E para que o objeto `TreeSet` tenha a certeza que os objetos que vai ordenar têm realmente esse método definido, esses objetos têm de ser também do tipo `Comparable<T>`.

O exemplo na Listagem 2 mostra criação de uma lista (imutável) com cinco produtos e a adição dos mesmos a um `TreeSet`. Podemo verificar que o mesmos estão ordenados porque quando iteramos sobre o objeto `orderedSet` os objetos surgem ordenados pela sua ordem natural.

[1] entre dois elementos do conjunto.

Listagem 2: Obtenção de objetos ordenados dentro de um TreeSet

```
Product p1 = new Product("chocolat", 100);
Product p2 = new Product("cookies", 200);
Product p3 = new Product("coffe", 50);
Product p4 = new Product("drink", 100);
Product p5 = new Product("drink", 70);

List<Product> products = Arrays.asList(p1, p2, p3, p4, p5);

Set<Product> orderedSet = new TreeSet<>(products);

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

Ao executar este código obtemos o seguinte resultado, ordenado por nome e depois (quando o nome é igual) por preço:

```
Product [name=chocolat, price=100]
Product [name=coffe, price=50]
Product [name=cookies, price=200]
Product [name=drink, price=70]
Product [name=drink, price=100]
```

Note-se que como a classe `Product` implementa a interface `Comparable<Product>` eles são também desse tipo pelo que o exemplo da Listagem 2 pode ser modificado para o que consta da Listagem 3.

Listagem 3: Obtenção de objetos ordenados dentro de um TreeSet

```
Comparable<Product> p1 = new Product("chocolat", 100);
Comparable<Product> p2 = new Product("cookies", 200);
Comparable<Product> p3 = new Product("coffe", 50);
Comparable<Product> p4 = new Product("drink", 100);
Comparable<Product> p5 = new Product("drink", 70);

List<Comparable<Product>> products = Arrays.asList(p1, p2, p3, p4, p5);

Set<Comparable<Product>> orderedSet = new TreeSet<>(products);

for(Comparable<Product> p : orderedSet)
{
  System.out.println(p);
}
```

O resultado é naturalmente igual ao anterior:

```
Product [name=chocolat, price=100]
Product [name=coffe, price=50]
Product [name=cookies, price=200]
Product [name=drink, price=70]
Product [name=drink, price=100]
```

9.1.1 Indicação de um objeto comparador

Também é possível criar o `TreeSet` vazio indicando um `Comparador`.

Listagem 4: Obtenção de objetos ordenados dentro de um TreeSet

```
Product p1 = new Product("chocolat", 100);
Product p2 = new Product("cookies", 200);
Product p3 = new Product("coffe", 50);
Product p4 = new Product("drink", 100);
```

```
Product p5 = new Product("drink", 70);

List<Product> products = Arrays.asList(p1, p2, p3, p4, p5);

// Cria um TreeSet vazio, mas indicando um Comparator
Set<Product> orderedPriceSet = new TreeSet<>(new ComparatorByPriceName());
orderedPriceSet.addAll(products);
for(Product p : orderedPriceSet)
{
  System.out.println(p);
}
```

Ao executar este código obtemos o seguinte resultado, ordenado por preço e depois (quando o preço é igual) por nome:

```
Product [name=coffe, price=50]
Product [name=drink, price=70]
Product [name=chocolat, price=100]
Product [name=drink, price=100]
Product [name=cookies, price=200]
```

No entanto, ainda não definimos a classe `ComparatorByPriceName`. A Listagem 5 mostra a definição dessa classe cujos objetos servem apenas para "incluir" o método onde definimos como ordenar objetos `Product` por preço e depois por nome.

Listagem 5: Classe `ComparatorByPriceName`

```
public class ComparatorByPriceName implements Comparator<Product>
{
  /**
   * Orders by name and then by price
   *
   * @param otherProduct
   *            product to order
   * @return -1, 0 or 1
   */
  @Override
  public int compare(Product p1, Product p2)
  {
    int cn = new Integer(p1.getPrice()).compareTo(p2.getPrice());
    if (cn == 0) // names are equal
    {
      return p1.getName().compareTo(p2.getName());
    }
    else
    {
      return cn;
    }
  }
}
```

Na verdade ainda temos mais duas formas de passar a função que faz a comparação (o compareTo):

1. Utilizando uma classe anónima onde se define o método;
2. Utilizando um método anónimo (um **lambda**).

9.1.2 Utilizando uma classe anónima (e respetivo objeto)

As Listagens 6 e 7 exemplificam como o fazer utilizando uma classe anónima.

Listagem 6: Utilizando uma classe anónima onde se define o método

```
Product p1 = new Product("chocolat", 100);
Product p2 = new Product("cookies", 200);
Product p3 = new Product("coffe", 50);
Product p4 = new Product("drink", 100);
Product p5 = new Product("drink", 70);
List<Product> products = Arrays.asList(p1, p2, p3, p4, p5);

Comparator<Product> compPriceName = new Comparator<Product>()
{
  @Override
  public int compare(Product p1, Product p2)
  {
    int cn = new Integer(p1.getPrice()).compareTo(p2.getPrice());
    if (cn == 0) // names are equal
    {
      return p1.getName().compareTo(p2.getName());
    }
    else
    {
      return cn;
    }
  }
};

Set<Product> orderedPriceSet = new TreeSet<>(compPriceName);
orderedPriceSet.addAll(products);
for(Product p : orderedPriceSet)
{
  System.out.println(p);
}
```

Listagem 7: Utilizando uma classe anónima onde se define o método passada logo por parâmetro

```
Product p1 = new Product("chocolat", 100);
Product p2 = new Product("cookies", 200);
Product p3 = new Product("coffe", 50);
Product p4 = new Product("drink", 100);
Product p5 = new Product("drink", 70);
List<Product> products = Arrays.asList(p1, p2, p3, p4, p5);

Set<Product> orderedPriceSet = new TreeSet<>(new Comparator<Product>()
{
  @Override
  public int compare(Product p1, Product p2)
  {
    int cn = new Integer(p1.getPrice()).compareTo(p2.getPrice());
    if (cn == 0) // names are equal
    {
      return p1.getName().compareTo(p2.getName());
    }
    else
    {
      return cn;
    }
  }
});

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

9.1.3 Utilizando um método anónimo (lambda)

Como o objeto do tipo `Comparator` apenas é utlizado para "transportar" o método `compareTo`, podemos (a partir da versão 8 da linguagem Java) utilizar um método anónimo (um lambda). Desta forma, o código fica mais simples. A Listagem 8 ilustra esta possibilidade.

Listagem 8: Utilizando um método anónimo passado por parâmetro

```
Product p1 = new Product("chocolat", 100);
Product p2 = new Product("cookies", 200);
Product p3 = new Product("coffe", 50);
Product p4 = new Product("drink", 100);
Product p5 = new Product("drink", 70);
List<Product> products = Arrays.asList(p1, p2, p3, p4, p5);

Set<Product> orderedPriceSet = new TreeSet<>(
    (Product pr1, Product pr2) ->
  {
    int cn = new Integer(pr1.getPrice()).compareTo(pr2.getPrice());
    if (cn == 0) // names are equal
    {
      return pr1.getName().compareTo(pr2.getName());
    }
    else
    {
      return cn;
    }
  });

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

A Listagem 9 mostra uma versão ainda mais simplificada e curta:

Listagem 9: Utilizando um método anónimo passado por parâmetro

```

Product p1 = new Product("chocolat", 100);
Product p2 = new Product("cookies", 200);
Product p3 = new Product("coffe", 50);
Product p4 = new Product("drink", 100);
Product p5 = new Product("drink", 70);
List<Product> products = Arrays.asList(p1, p2, p3, p4, p5);

Set<Product> orderedPriceSet = new TreeSet<>(
  (pr1, pr2) ->
  {
    int cn = new Integer(pr1.getPrice()).compareTo(pr2.getPrice());
    return (cn == 0) ? pr1.getName().compareTo(pr2.getName()) : cn;
  }
);

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

9.2 Ordenação de uma lista

Utilizando exactamente os mesmos conceitos já apresentados para inserção num objeto `TreeSet`, nomeadamente as interfaces Comparable<T>, Comparator e lambdas, podemos ordenar diretamente uma lista. A Listagem 10 exemplifica como o fazer utilizando um lambda.

Listagem 10: Ordenação de uma `List`

```
Product p1 = new Product("chocolat", 100);
Product p2 = new Product("cookies", 200);
Product p3 = new Product("coffe", 50);
Product p4 = new Product("drink", 100);
Product p5 = new Product("drink", 70);
List<Product> products = Arrays.asList(p1, p2, p3, p4, p5);

List<Product> unorderedList = new ArrayList<>(products);

unorderedList.sort(
  (pr1, pr2) ->
  {
    int cn = new Integer(pr1.getPrice()).compareTo(pr2.getPrice());
    return (cn == 0) ? pr1.getName().compareTo(pr2.getName()) : cn;
  });

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

10 Exercicios

1. Defina a classe `Product` como classe canónica. Para tal pode utilizar a ajuda do IntelliJ: Clique com o botão direito no editor de texto e escolha `Generate->equals() and hashCode()`. Implemente o método `compareTo`, já apresentado acima, e que ordena por nome e depois por preço (quando os nomes são iguais).

2. Defina um teste na classe `Product` que permita testar a ordenação dos objetos. Experimente utilizar um ciclo *for* na lista de objetos do tipo `Product` para comparar um a um com os objectos de uma lista `expected` já ordenada. Experimente também utilizar apenas um `assertEquals(expected, returned)`.

3. Defina um objeto do tipo `TreeSet<Product>`. Crie um `ArrayList<Product>` a partir do `TreeSet<Product>` e verifique que os objetos ficaram ordenados. Para criar o `ArrayList<Product>` a partir do `TreeSet<Product>` utilize o construtor `ArrayList(Collection<? extends E> c)`.

Deve continuar 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.