# 7 - Interfaces

## 7.1 Introdução

Numa linguagem de PPO, as classes definem a estrutura e o comportamento comum das suas instâncias. Por outro lado, as classes fazem parte de uma hierarquia particular que as relaciona, e cujo mecanismo de herança faz com que seja uma hierarquia de **especialização**. Uma classe que declara que *extends* uma outra torna-se subclasse da primeira e a primeira é, em JAVA, a única possível superclasse da segunda, já que a herança é de tipo simples. A subclasse é uma especialização da superclasse, pois herda toda a estrutura e comportamento definidos, podendo acrescentar-lhe mais estrutura e mais comportamento. Assim, no sentido descendente da hierarquia, temos classes cada vez mais específicas, ou seja, mais detalhadas ou refinadas.

Os conceitos de classe e subclasse (associados ao mecanismo de herança) estão assim muito ligados a uma filosofia muito interessante de concepção e construção de *software* que se baseia na procura de reutilização de componentes já existentes, e que fomenta um estilo de programação que, como vimos antes, é **programação incremental**.

Numa linguagem de PPO pura, como Smalltalk, por exemplo, o tipo de uma dada instância associa-se de forma natural à classe que a definiu e criou, sendo as subclasses de tal classe vistas como **subtipos**.

Porém, a noção de **subtipo** tem, em geral, uma conotação mais abstracta que a noção de subclasse. Um subtipo é sempre definido apenas em função do comportamento e não da estrutura. Assim, **B** pode ser considerado um subtipo de **A**, se qualquer entidade do tipo **A** puder ser "substituída" por uma do tipo **B** sem que de tal substituição resulte qualquer mudança no comportamento dos programas que usam **A** (**princípio da substituição**).

A noção de subclasse de uma dada classe assume um relacionamento que inclui comportamento (métodos) mas também estrutura (atributos), e sabemos até que satisfazem ainda assim o **princípio da substituição**. Em termos formais, subclasses e subtipos são conceitos que não têm necessariamente que estar relacionados entre si.

As linguagens de PPO *strongly typed* (fortemente tipadas), ou seja, com um rigoroso mecanismo de verificação de tipos em tempo de compilação, tais como C#, C++ e JAVA, tendem a diminuir a distinção entre **subclasses** e **subtipos** de duas formas:

- Permitem que variáveis possam ser associadas a instâncias de uma classe diferente da classe da sua declaração, desde que a instância a associar à variável seja de uma subclasse da classe de declaração;
- Assumem que subclasses são subtipos, o que formalmente pode não ser verdade já que as subclasses podem não só redefinir como acrescentar métodos.

JAVA assume que as classes servem como tipos e as subclasses servem como subtipos das variáveis referenciadas, mas introduz um mecanismo adicional visando a especificação sintáctica de tipos abstractos de dados (TAD), ou seja, permitindo a **especificação de tipos** através das **interfaces**.

Para além de especificações de tipos, as interfaces de JAVA apresentar-se-ão como um mecanismo extremamente útil sempre que pretendermos especificar e garantir que duas classes posicionadas em quaisquer zonas da hierarquia de classes, logo não tendo qualquer relação hierárquica entre si, possuem comportamento comum.

Consideremos a hierarquia representada na Figura 7.1. Quer sejam concretas ou abstractas, sabemos que as classes **C** e **D** possuem uma linguagem comum, ou API, que é a resultante de herdarem as "linguagens" de `Object` e de **A**.

As classes **B** e **E**, dado o seu posicionamento na hierarquia, possuem "linguagens" distintas de **C** e **D**, tendo apenas em comum a linguagem que é definida nos métodos das superclasses comuns, neste caso, apenas `Object`.

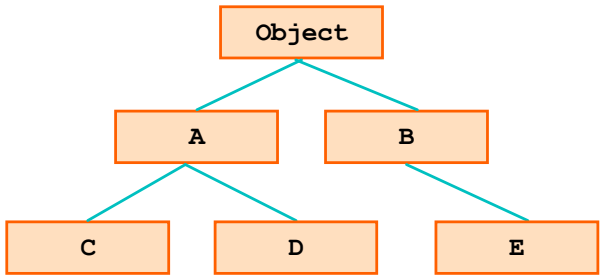


**Figura 7.1 – Hierarquia - exemplo 1**

Consideremos o problema conceptual de se pretender que as classes **C** e **E**, **e apenas estas,** implementem também os métodos `m1()` e `m2()`, ou seja, sejam ambas capazes de, adicionalmente, responder às mensagens `m1()` e `m2()` (conforme Figura 7.2).

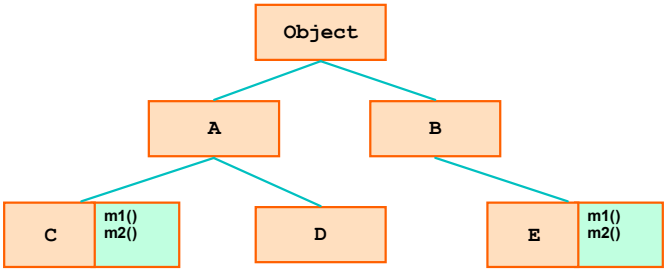


**Figura 7.2. – Interfaces: Hierarquia - exemplo 2**

Perante um requisito deste tipo, se a linguagem de programação usada apenas oferece ao programador o mecanismo de herança, então, o seu cumprimento apenas poderia ser satisfeito se, tomando como exemplo a hierarquia da Figura 7.2, à classe que é a super-classe comum das classes a quem pretendemos ver acrescentado tal comportamento, no exemplo **`Object`**, fossem adicionados os métodos `m1()` e `m2()`. Tal decisão garantiria que ambas as subclasses herdariam os métodos pretendidos, ainda que os tivessem que implementar caso estes fossem abstractos em `Object` (e fosse também possível modificar a classe `Object`).

Esta solução é, no entanto, inaceitável do ponto de vista de uma programação que se pretende que seja estável, mas genérica e extensível. Ter que alterar uma classe sempre que se pretenda que algumas das suas mais distantes, e não relacionadas, subclasses implementem um dado comportamento não é aceitável e não pode ser uma solução para qualquer problema dadas as suas implicações na hierarquia de classes.

Se tal solução fosse implementada na hierarquia exemplo, ou seja, se os métodos `m1()` e `m2()` fossem acrescentados a `Object`, então, não só **C** e **E** iriam passar a responder a tal "linguagem", como também teriam que o fazer todas as outras classes intermédias da hierarquia, que no exemplo são "todas". Ou seja, teríamos que modificar todas as classes desta hierarquia. Tal não poderia ser nunca uma solução.

A linguagem JAVA vai responder a este tipo de requisitos sobre certas classes, aos quais as regras e restrições naturais do mecanismo de herança não permitem responder de forma satisfatória, com as interfaces. Noutras linguagens, a herança múltipla poderia servir.

**Interfaces** são apenas um conjunto de especificações sintácticas (abstractas portanto) de métodos que representam um comportamento particular, que qualquer classe, **em qualquer ponto da hierarquia**, pode implementar.

As interfaces especificam um **tipo de dados** (conjunto de métodos abstractos a implementar), e qualquer classe que os implemente passa a ser **compatível** com tal tipo de dados. Ser compatível com o tipo de dados (que é o nome da interface) significa que as instâncias dessa classe podem ser associadas a variáveis declaradas de tal tipo.

Depois de se apresentarem as formas correctas da sua declaração, estudaremos o mecanismo de *interfaces* de JAVA, seguindo exactamente a sequência dos três problemas que o mesmo visa melhorar relativamente a outras linguagens de PPO, a saber:

- Definição clara de novos tipos de dados;
- Implementação de tais tipos de dados;
- Possibilidade de garantir que classes não relacionáveis na hierarquia possam partilhar certas propriedades comportamentais, ou seja, que implementam certo comportamento especificado.

## 7.2 INTERFACES JAVA: DECLARAÇÃO

A declaração de uma interface JAVA é muito simples, dado que apenas especifica um conjunto de assinaturas de métodos, implicitamente `abstract`, e, opcionalmente, um conjunto de constantes, implicitamente `public static final`.

```
public interface AutoEjectavel {
   int TEMPO_EJECT = 5;
   public void ejectar();
}
```

As *interfaces* JAVA são declarações puramente sintácticas ainda mais restritivas que as classes abstractas. Numa classe abstracta é possível introduzir métodos concretos. **Numa interface, todos os métodos são abstractos por definição**. Numa classe abstracta, é possível introduzir variáveis de instância que todas as subclasses herdam e podem redefinir. **Numa interface, os únicos identificadores de valores são constantes** e **`final`**.

**As interfaces não fazem parte da hierarquia de classes**, como as classes abstractas, antes existindo no seu espaço próprio, onde se vão relacionar hierarquicamente entre si, e tendo com as classes os relacionamentos semânticos que veremos a seguir.

Consideremos mais alguns exemplos de declarações de interfaces:

```
public interface Enumeravel {
   public abstract boolean vazia();
   public abstract Object seguinte();
}

public interface Colorivel {
   public void defineCor(int cor);
}


public interface Ordem {
   public boolean igual(Ordem elem);
```

```
    public boolean maior(Ordem elem);
    public boolean menor(Ordem elem);
}
```

As interfaces anteriores definem os tipos de dados `Enumeravel`, `Colorivel` e `Ordem`.

Em JAVA, as únicas entidades onde se pode definir código concreto são, como já sabemos, as classes. Assim, a sintaxe especificada numa interface deve ser implementada numa qualquer classe de JAVA, devendo ficar claro que, "qualquer" significa mesmo uma qualquer classe de JAVA que declare (e assuma) a implementação de uma ou mais interfaces. Deste modo, uma classe passa a implementar um tipo de dados especificado numa dada interface e também, e em simultâneo, passa a ser uma subclasse "natural" (cf. o mecanismo de herança) de uma classe qualquer.

**Variáveis de tipos referenciados poderão ter tipos associados a identificadores de classes ou de interfaces** a partir de agora. Claro que as interfaces não criam instâncias, mas as classes que as implementam poderão fazê-lo.

Para uma instância `c` de uma classe que declara implementar uma dada interface `I`, a expressão `c instanceof I` dá como resultado `true`.

Qualquer classe que pretenda satisfazer a especificação de um destes tipos apenas terá que fornecer implementação para os métodos abstractos e declarar que implementa as interfaces.

Em JAVA, se uma classe B é subclasse de A, declaramos:

```
public class B extends A {
```

Se, em simultâneo com o facto de ser subclasse de A, a classe **B implementa uma interface I**, isto é, o **tipo especificado em I**, deveremos então declarar:

```
public class B extends A implements I {
```

ou até, caso implemente várias interfaces,

```
public class B extends A implements I1, I2, …, In {
```

## 7.3 HIERARQUIA DE INTERFACES

As interfaces são definidas e criadas como uma classe normal de JAVA, ou seja, num ficheiro fonte de extensão *.java* cujo nome será o nome da própria interface. Após a compilação, as interfaces vão ocupar uma posição na **hierarquia de interfaces** de JAVA, que tem a característica de ser uma hierarquia de herança múltipla e distinta da de classes.

Uma interface pode ter ou não superinterfaces ou subinterfaces, tal como pode ter mais do

que uma superinterface, herdando todos os métodos nestas definidos. Este relacionamento hierárquico é, tal como para as classes, definido usando a palavra reservada ***extends***, tal como se apresenta nos exemplos seguintes:

```
public interface Desenhavel extends Colorivel {
   public void posicao(double x , double y);
   public void desenha() throws OutOfBoundsException;
}
```

A interface `Desenhavel` é subinterface de `Colorivel`. Logo, ser do tipo `Desenhavel` implica implementar os métodos definidos em `Desenhavel` e em `Colorivel`:

```
public interface Amovivel {
   public void movimento(double x, double y);
}

public interface Particula extends Amovivel {
   int LIM_VELOC = 120;
   public void acelera(long dv);
}

public interface Transformavel
                  extends Escalavel, Rodavel, Desenhavel {
}
```

A interface `Transformavel` herda de `Escalavel`, `Rodavel` e de `Desenhavel`, pelo que ser do tipo `Transformavel` é implementar todos os métodos herdados, já que esta interface, por si, apenas dá o nome ao tipo, pois é uma **interface vazia** também designada por ***marker interface*** por essa mesma razão. As *marker interfaces* são como etiquetas, sendo úteis para determinar se os objectos têm certas propriedades usando `instanceof`.

As declarações de interfaces JAVA apresentadas são perfeitamente abstractas, ou seja, não impõem qualquer tipo de restrição sobre as implementações, apenas definindo a sintaxe das operações que se pretendem ver implementadas para que uma classe satisfaça a **propriedade** ou **especificação** de poder ser `Enumeravel`, `Colorivel`, `Ordenavel` ou `Transformavel`.

## 7.4 IMPLEMENTAÇÃO DE INTERFACES

Podemos agora analisar com mais detalhe as declarações das interfaces JAVA apresentadas anteriormente, tendo em especial atenção que as mesmas prescrevem uma obrigatoriedade de implementação de todos os métodos abstractos por parte das classes que declararem serem suas implementações, ou seja, que usem a declaração ***implements***.

Consideremos as interfaces `Apagavel` e `Regravavel` definidas como,

```
public interface Apagavel {
```

```
    public void apaga();
}
public interface Regravavel extends Apagavel {
    // método(s) a decidir
}
```

em que `Regravavel` é subinterface de `Apagavel`, e para a qual veremos que métodos poderia fazer sentido especificar.

Consideremos de novo a hierarquia de `ItemMulti` (Figura 7.3) que usámos anteriormente, à qual foram acrescentadas algumas subclasses. Vamos modificar a classe `Hi8` de modo a que implemente a interface `Apagavel`.

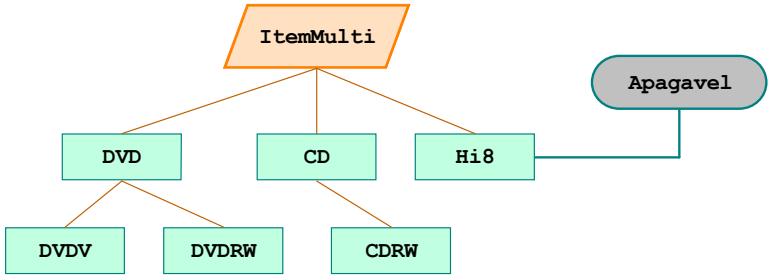


**Figura 7.3 – Hierarquia `ItemMulti`**

Apresenta-se a seguir o essencial do código da classe e as modificações introduzidas:

```
public class Hi8 extends ItemMulti implements Apagavel {
   //
   private int minutos;
   private double ocupacao;
   private int gravacoes;
    . . . .
   // implementação de Apagavel
   public void apaga() { ocupacao = 0.0; gravacoes = 0; }
}
```

No caso da classe `Hi8`, a implementação do método `apaga()` consiste num "*soft clear*", isto é, um *clear* lógico, modificando os valores das variáveis que controlam os conteúdos da cassete.

Se a classe `Hi8` implementa a interface `Apagavel`, então uma qualquer instância de `Hi8` é também do tipo `Apagavel`, pelo que serão válidas as seguintes expressões:

```
Hi8 filme1 = new Hi8("A1", "2005", "obs1", 180, 40.0, 3);
Apagavel apg1 = filme1;
apg1.apaga();
```

```
out.println("Ocupacao = " + filme1.getOcupa());
out.println("Gravações = " + apg1.getGrava()); // ERRO
```

Nesta sequência de instruções, é criada uma instância de `Hi8`, é declarada uma variável do tipo `Apagavel` e é-lhe atribuída a instância de `Hi8` criada. A atribuição é pois correcta porque `Hi8` é um tipo compatível com `Apagavel`, dado que `Hi8` implementa a interface. Podíamos mesmo ter escrito:

```
Apagavel apg1 = new Hi8("A1", "2005", "obs1", 180, 40.0, 3);
```

Porém, neste caso, teríamos criado um `Hi8` para nada, pois de imediato o "truncamos" ao convertê-lo para `Apagavel`, já que perde toda a informação e comportamento, dado que, enquanto objecto do tipo `Apagavel` apenas responde à mensagem `apaga()`.

Depois de escrevermos `apg1.apaga();` pretende-se confirmar que o estado de `filme1` foi mesmo modificado e escrevemos `filme1.getOcupa();` para calcular a ocupação actual. O valor devolvido é 0.0 tal como esperávamos, o que demonstra que `apg1` é uma referência para `filme1` (cf. a atribuição realizada `Apagavel apg1 = filme1;`).

Se ambas as variáveis partilham a mesma informação, então, para verificar se o método `apaga()` também limpou o número de gravações, escrevemos **`apg1.getGrava();`**.

Porém, neste caso, o compilador dá um erro de compilação, pois uma instância do tipo `Apagavel` não possui na sua API o método `getGrava()`.

Note-se que, se cometêssemos o duplo erro de escrever:

```
Apagavel apg2 = new Hi8("A2", "2004", "obs2", 160, 50.0, 6);
apg2.apaga();
```

nunca mais poderíamos confirmar se o filme "A2" havia sido apagado ou não.

Em conclusão, os tipos são compatíveis, e uma instância da classe implementadora da interface pode ser atribuída a uma variável do tipo da interface, mas, através do tipo da interface, apenas os métodos desse tipo estarão disponíveis (o que é lógico).

Naturalmente que a uma instância de `Hi8` se pode enviar a mensagem `apaga()` directamente, não sendo necessário nem atribuí-la a uma variável `Apagavel` nem fazer *casting*.

```
Hi8 filme1 = new Hi8("A1", "2005", "obs1", 180, 40.0, 3);
filme1.apaga();
out.println("Ocupacao = " + filme1.getOcupa());
```

Considere-se a hierarquia apresentada na Figura 7.4, onde as classes `CDRW`, `DVDRW` e `DVDV` declaram que implementam a interface `Regravavel`, interface esta que é uma sub-interface de `Apagavel`:

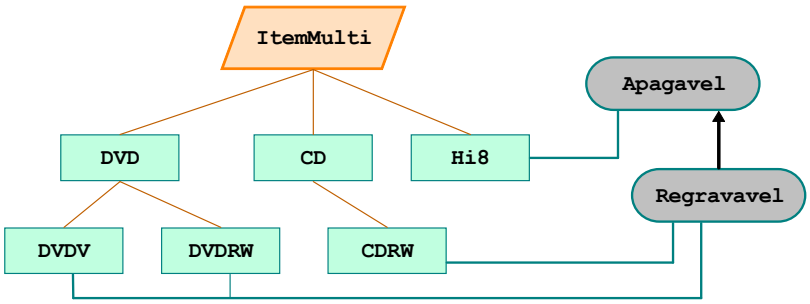


**Figura 7.4 – Nova hierarquia de items multimédia**

Não há muitos métodos que façam sentido especificar nesta interface `Regravavel`. Poderíamos definir um método `grava(int tempo)` correspondente a mais uma gravação de certo tempo, mas este seria o método normal para fazer a gravação. No entanto, como o exemplo mostra, depois de um dispositivo estar gravado, faz todo o sentido saber se ele é ou não `Regravavel`.

Portanto, temos uma situação excepcional em que criámos uma interface `Regravavel` sem quaisquer métodos definidos, ou seja, uma **interface vazia** ou ***marker interface***, segundo a terminologia JAVA. Esta interface serve para notificar os interessados de uma propriedade que uma classe pode possuir ou não, e que pode ser importante para certas operações sobre instâncias da mesma. Não implica codificar qualquer método.

No nosso exemplo, poderíamos criar um *array* de `ItemMulti`, e determinar quantos dos seus elementos poderão ser regravados. Para tal, vamos usar o operador **instanceof**, que testa se o tipo dinâmico do objecto do seu lado esquerdo coincide com o tipo especificado no seu lado direito (ou é subtipo deste):

```
ItemMulti[] filmes = new ItemMulti[500];
// código para inserção de filmes no array....
int contaReg = 0;
for(ItemMulti filme : filmes)
  if (filme instanceof Regravavel) contaReg++
out.printf("Existem %d items regraváveis.", contaReg);
```

As *interfaces* definem portanto um protocolo de comportamento que pode ser implementado por qualquer classe em qualquer ponto da hierarquia. Do ponto de vista da concepção de programas, as interfaces são úteis para:

- Reunirem similaridades comportamentais (métodos) entre classes não relacionadas sem forçar artificialmente o seu relacionamento hierárquico;
- Definirem novos tipos implementáveis por qualquer classe;
- Conterem a API comum de objectos sem indicação da sua verdadeira classe.

Conceptualmente, as interfaces JAVA estão ligadas a certas propriedades operacionais

que se pretendem ver realizadas nas diversas implementações dos respectivos tipos de dados, tais como: o tipo ser enumerável, o tipo ser clonável, ser visualizável, ser colorível, ser ordenável, etc.. Tal é sempre garantido, dado que qualquer classe que implemente uma interface tem obrigatoriamente que fornecer uma implementação concreta para cada um dos métodos abstractos declarados na interface, bem como para todos os que são por esta herdados.

Ou seja, a partir de agora, uma qualquer classe de JAVA poderá, em função da forma como é definida, ser vista de duas diferentes perspectivas ou *views* (Figura 7.5): de quem é subclasse, e, portanto, de quem herda estrutura e comportamento (e o que destes redefine ou acrescenta) e, ainda, que interfaces implementa, ou seja, quais as propriedades adicionais que acrescenta às suas instâncias, tais como, por exemplo, serem Conjuntos, serem Coloríveis, serem Regraváveis, Clonáveis, Persistentes, etc.

Algumas destas propriedades são, como veremos em seguida, fundamentais para uma efectiva utilização das classes desenvolvidas pelo utilizador de JAVA e até das classes já existentes.

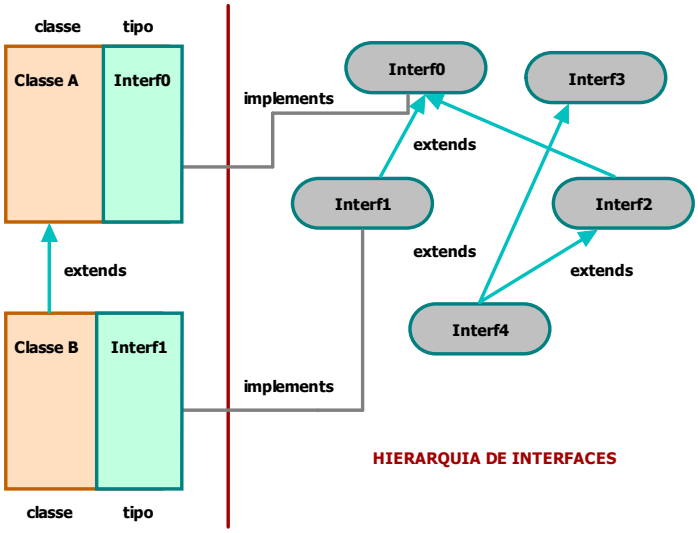


**Figura 7.5 - Classes JAVA vistas como classes e como tipos**

Numa dada classe de JAVA, convergem agora, quando tal se verifica, as noções de **classe** e **tipo**, ainda que através de mecanismos diferentes que foram já devidamente identificados.

Havendo em JAVA uma clara separação entre subtipos e subclasses, deverão ser sempre e apenas as classes de JAVA a fornecer as implementações. Assim, uma classe passará a ter duas *views* (ou interpretações) possíveis:

- É **subclasse** por se encontrar inserida na hierarquia normal de classes, caracterizada por possuir um mecanismo de herança simples de estrutura e comportamento, ainda que com redefinições e extensões;
- É **subtipo** por se encontrar semanticamente "envolvida", ainda que apenas como o único mecanismo possível de implementação, numa hierarquia lógica com herança múltipla de definições de comportamento abstracto, ou seja, de especificações puramente sintácticas.

Quando uma classe implementa uma interface, define um conjunto de métodos que nada têm a ver com a normal hierarquia de classes, mas apenas com o facto de que essa classe pretende ser também compatível com o tipo definido pela interface. No entanto, o que se passa com as suas subclasses? São obrigadas a satisfazer tal tipo? Herdam ou não tais métodos?

Conceptualmente, não se deveria considerar que as subclasses de uma classe que implementa uma ou mais interfaces herdem tais métodos, pois estes métodos não resultam de um processo de subclassificação mas de um processo de implementação de tipos.

De facto, se uma classe define um conjunto de métodos que implementam uma dada interface e se as suas subclasses herdam tais métodos, então, as suas subclasses também implementariam tal interface e, portanto, estariam igualmente associadas a tal tipo. No entanto, se não o declaram expressamente, então, tal deveria ser considerado como se não o implementassem de facto, e assim a invocação de tais métodos deveria provocar erros de compilação.

Por estas razões e em definitivo, a regra que deve ser seguida é a de que os **métodos de implementação de interfaces JAVA definidos numa dada classe não devem, do ponto de vista de concepção, ser considerados como herdados pelas subclasses desta**.

No entanto, e em rigor, este princípio conceptual não é satisfeito em JAVA. Os métodos de uma classe que implementa uma ou mais interfaces são herdados pelas suas subclasses e podem mesmo ser invocados nestas, e as instâncias das subclasses são compatíveis com variáveis dos tipos das interfaces. Porém, na informação interna destas subclasses as interfaces não aparecem como sendo suas interfaces implementadas.

## 7.5 CLASSES ABSTRACTAS *VERSUS* INTERFACES

As interfaces JAVA parecem ser, numa primeira análise, excelentes substitutas para as classes abstractas. Porém, esta ideia aparentemente correcta, dado existir alguma possível semelhança entre uma classe 100% abstracta e uma interface, não tem em consideração um grande número de diferenças (para além das conceptuais que acabámos de apresentar) entre as duas entidades da linguagem JAVA, entre outras:

- Uma classe abstracta pode não ser 100% abstracta; porém, uma interface é sempre 100% abstracta (ou seja, sintáctica e sem especificação de qualquer semântica);
- Uma classe abstracta não impõe às suas subclasses a implementação obrigatória

dos métodos abstractos, ou seja, uma subclasse de uma classe abstracta pode ainda ser uma classe abstracta; uma interface impõe, à classe que declarar que a implementa, uma implementação completa;

- Uma classe abstracta pode ser usada para se escrever *software* genérico, parametrizável e extensível; uma interface não tem, em princípio, tais objectivos, sendo em geral associada à necessidade de se especificar um conjunto adicional de propriedades funcionais, ou seja de tratamento, que sejam garantidas por uma dada implementação, podendo ser tais classes de implementação horizontais na hierarquia, isto é, não existindo entre si qualquer relacionamento na hierarquia simples de classes de JAVA;
- Classes e interfaces coexistem em JAVA em hierarquias com propriedades distintas, resultando o poder expressivo e definicional da linguagem JAVA da correcta simbiose das duas, o que requer, naturalmente, muita prática em desenvolvimento de grandes aplicações.

Esta questão da utilização de classes abstractas *versus* interfaces é um assunto recorrentemente estudado, não havendo regras ou metodologia para enumerar. Há, no entanto, através de exemplos, a possibilidade de se analisarem situações comuns.

Para melhor analisarmos algumas das diferenças entre classes abstractas e interfaces do ponto de vista da sua utilização efectiva, consideremos a seguinte hierarquia (Figura 7.6) onde temos três grandes famílias de classes que se distinguem pelas três classes abstractas: `Veiculo`, `Animal` e `Forma`.

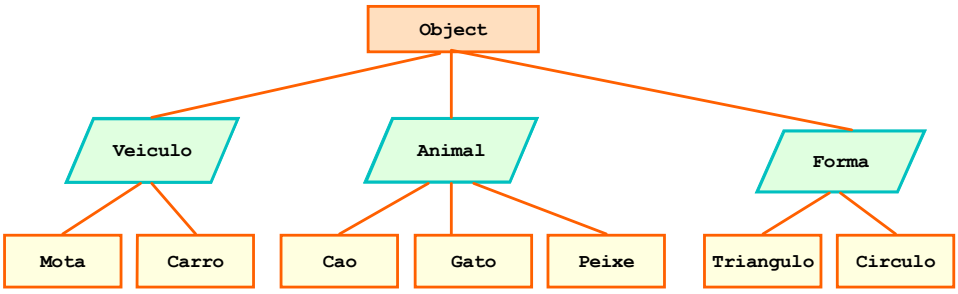


**Figura 7.6 – Hierarquia exemplo**

O exercício que vamos fazer sobre esta hierarquia é estrutural e conceptual, pelo que não tem de momento importância saber precisamente que métodos estão definidos em cada classe.

Pretende-se que todas as subclasses de `Veiculo` implementem um método `aLavar()`, que indica um estado particular do objecto, podendo cada uma delas fazê-lo da forma como entender em função dos seus atributos actuais, ou até acrescentando atributos se tal for necessário para garantir a implementação do método.

Uma primeira decisão, será criar uma classe abstracta designada `Lavavel`, da qual todas as subclasses herdam, classe que define o método como sendo abstracto, o que vai forçar as subclasses a realizarem a sua implementação. A hierarquia ficaria então (Figura 7.7):

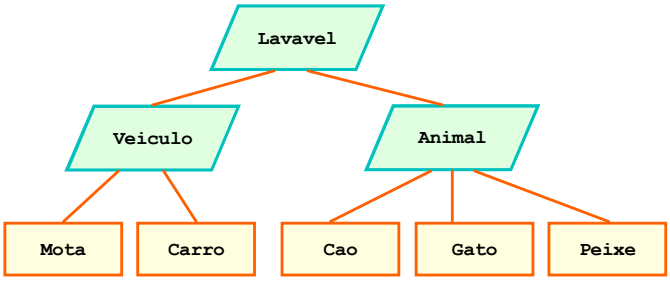


**Figura 7.7 – `Lavavel` como classe abstracta**

A decisão de criar `Lavavel` como uma classe abstracta, a ser posicionada na hierarquia como superclasse de `Veiculo`, impõe à classe abstracta `Animal` a herança do método abstracto de `Lavavel`, impondo às subclasses de `Animal` que implementem o método da classe abstracta. Para além de todas as questões de concepção, designadamente a inusitada reestruturação forçada da hierarquia, as instâncias de `Peixe` não compreenderiam e as de `Gato` não gostariam.

Este é um exemplo claro de uma situação de concepção, em que a possibilidade de utilização de uma **interface** se revela fundamental em termos de liberdade conceptual. Assim, criamos a interface `Lavavel` (Figura 7.8), que não interfere com a hierarquia de classes, e as classes que a desejarem implementar apenas terão que a declarar.

```
public interface Lavavel {
   public void aLavar();
}
```

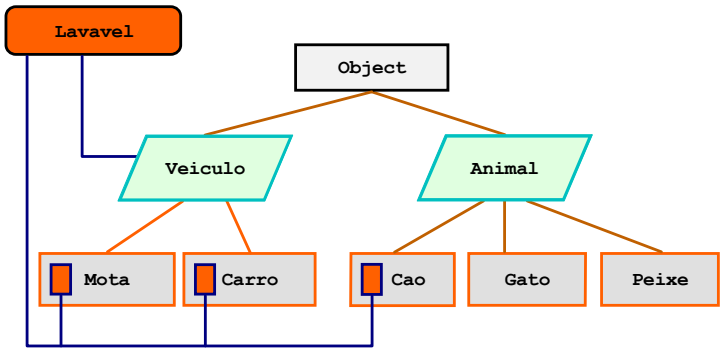


**Figura 7.8 – Interface e classes de implementação**

No exemplo, temos uma classe concreta que declara pretender ser do tipo `Lavavel`, a classe `Cao`, que também é do tipo `Animal` (passa a ser um `Animal` e `Lavavel`), e uma classe abstracta `Veiculo`, que também declara implementar `Lavavel`.

```
public abstract class Veiculo implements Lavavel {
   // Força as subclasses a implementarem a interface
   // Lavavel, logo, todos os veículos são laváveis
```

A classe abstracta `Veiculo` ao realizar esta declaração vai forçar as suas subclasses a serem todas de tipo `Lavavel`. Se a classe abstracta implementar o método `aLavar()`, as subclasses herdam esse código. Se não implementar, então, as subclasses são obrigadas a fornecer implementações para o método.

Fica mais uma vez clara a distinção entre a utilização de classes abstractas e de interfaces, em particular a facilidade com que a utilização destas últimas nos permite acrescentar **identidade de tipo** e **similaridade de tipo** a classes improváveis para tal (cf. `Cao` e `Mota`) sem que tal implique, porque também não o justificaria, realizar modificações na hierarquia de classes.

O tabela 7.1 apresenta mais algumas diferenças entre a utilização de interfaces e de classes abstractas, relativamente a propriedades diversas, não tanto conceptuais mas mais relacionadas como a funcionalidade e restrições na utilização.

| Propriedades | Interfaces | Classes Abstractas |
|---|---|---|
| **Herança Múltipla** | Uma classe pode implementar várias interfaces | Uma classe só pode "estender" uma classe abstracta |
| **Implementação** | Não aceita nenhum código | O código que se pretender, cf. as necessidades da concepção |
| **Constantes** | Apenas `static final`. Podem ser usadas pelas classes implementadoras sem qualificadores | De classe ou de instância e código de inicialização se necessário |
| **Relação *is-a*** | Em geral não descrevem a identidade principal de uma classe, mas capacidades periféricas aplicáveis a outras classes | Descreve uma identidade básica de todos os seus descendentes |
| **Adicionar funcionalidade** | Não deve ser feito pois implicaria refazer todas as implementações da interface. Usar antes subinterfaces | Sem problema para as subclasses desde que o método seja codificado na classe abstracta |
| **Homogeneidade** | Classes homogéneas em API | Classes homogéneas em API e que partilham algum código, por exemplo, classes de dados |
| **Velocidade (não é relevante)** | Requer extra indirecção na procura | Procura directa do método |

**Tabela 7.1 – Interfaces *versus* classes sbstractas**

## 7.6 INTERFACES CONSTANTES E IMPORTAÇÃO ESTÁTICA

**Interfaces constantes** são interfaces que declaram apenas constantes (`public static`). Quando uma classe declara que implementa tal interface, pode ter acesso aos identificadores dessas constantes sem ter que usar o nome da interface como prefixo, tal como se mostra no exemplo seguinte:

```
public interface Taxas {
   double TAXA1 = 5.0;
   double TAXA2 = 21.0;
   double TAXA3 = 30.0;
}

public class Produto implements Taxas {
   . . . .            // variáveis e métodos
   public double preco() {
      . . .
      if(precoBase > limite2)
          precoFinal = precobase * (1.0 + TAXA3);
      . . .
   }
}
```

Esta técnica de **exportação de constantes**, que tem alternativas bastante melhores, é pouco aconselhada por várias razões, das quais apenas referiremos algumas.

A primeira, é quase uma questão de "honra" ou de "identidade". As interfaces foram criadas para especificarem tipos de dados e uma interface constante não é um tipo de dados. A segunda tem a ver com o facto de que uma interface se destina a especificar as instâncias de uma classe, e só excepcionalmente se especificam constantes de classe que sejam importantes para o comportamento de tais instâncias. Fazer o contrário é subverter a noção de interface.

Uma classe que implementa uma interface constante "exporta" tais constantes para todas as suas subclasses também, passando a fazer parte das API de todas elas. A esta subversão foi mesmo dado o nome de *Constant Interface Antipattern.*

Assim, antes de JAVA5, a técnica aconselhada para guardar um conjunto de constantes de forma a que pudessem ser exportadas para as classes que delas necessitassem, seria criar uma classe `final` não instanciável, contendo todas as constantes `static` necessárias, e depois importar a classe e usá-las com o nome da classe como prefixo:

```
public final class Taxas {
   public static final double TAXA1 = 5.0;
   public static final double TAXA2 = 21.0;
   public static final double TAXA3 = 30.0;
   private Taxa() {}
}
```

```
import minhas.java.Taxas;
public class Produto implements Taxas {
. . . .
   public double preco() {
      . . .
      if(precoBase > limite2)
        precoFinal = precobase * (1.0 + Taxas.TAXA3);
      . . .
   }
}
```

A nova instrução de importação estática, vem permitir superar todas estas dificuldades e libertar as interfaces deste propósito, que consistia em realizar a missão de um mecanismo, então inexistente, de agrupamento de valores "globais" para uso em certos contextos.

Com importação estática, criada a classe final `Taxas` basta importá-la para `Produto` como um todo, e usar os identificadores sem prefixo:

```
import static minhas.java.Taxas.*;
public class Produto {
   . . .
   precoFinal = precobase*(1.0 + TAXA3);
   . . .
```

ou realizando as importações individualizadas em função das necessidades, como em:

```
import static minhas.java.Taxas.TAXA1;
import static minhas.java.Taxas.TAXA2;
import static minhas.java.Taxas.TAXA3;
public class Produto {
   . . .
```

A introdução em JAVA5 dos **tipos enumerados**, que estudaremos no capítulo seguinte, em conjunção com as importações estáticas, conduzirão a soluções elegantes e bem claras para este tipo de necessidades de representação de informações constantes.

## 7.7 HIERARQUIA MÚLTIPLA DE INTERFACES

Existindo uma hierarquia de interfaces de herança múltipla, torna-se importante analisar quais as regras quanto à redefinição de constantes e de métodos, e quanto a eventuais colisões de nomes provocadas quer pelos mecanismos de herança de interfaces quer pela via da sua implementação em classes que possuem outros métodos implementados.

As interfaces podem naturalmente ser compiladas sem erros, mas as classes que as implementam podem gerar erros de compilação por incompatibilidades de métodos ou mesmo de constantes de classe. Por exemplo, se uma classe declarar que implementa duas interfaces e se nestas existir um método com a mesma assinatura mas resultados de tipos diferentes, as interfaces compilam sem problemas, mas a classe gerará erros de compilação, e só num caso especial poderá, de facto, implementar tais interfaces.

Vejamos então as regras a satisfazer relativamente a constantes e a métodos:

### CONSTANTES

- Podem ser redefinidas nas subinterfaces, ou seja, dado novo valor ao mesmo identificador;
- As classes que implementam subinterfaces podem ter acesso a constantes das superinterfaces que foram redefinidas prefixando-as com o nome da interface;
- A regra anterior aplica-se em situações de ambiguidade por herança múltipla.

### MÉTODOS

- Toda a análise relativa a métodos herdados e redefinidos tem por base a assinatura dos mesmos, no sentido lato, tendo particular relevância a regra do tipo de resultado;
- Os métodos de uma superclasse apenas podem ser redefinidos por métodos que, tendo a mesma assinatura, tenham um tipo de resultado do mesmo tipo ou de um subtipo do método redefinido para os tipos referenciados;
- Os métodos com o mesmo nome e assinaturas distintas são distintos;
- Uma subinterface não pode herdar das suas superinterfaces métodos de igual nome, igual assinatura e tipos de resultados incompatíveis.

Considere-se que a classe A é superclasse de B no exemplo seguinte.

```
public interface Inter1 {
   public A m1(int x, int y);
   public String m2(String n1, String n2);
}

public interface Inter2 {
   public B m1(int x, int y);
   public double m2(String n1, String n2);
}
```

A compilação da interface `Inter3` a seguir geraria um erro de compatibilidade.

```
public interface Inter3 extends Inter1, Inter2 {
// marker interface
}
```

A interface herda os métodos `m1()` e `m2()` mas apenas `m2()` dá erro porque os métodos `m1()` possuem tipos de resultado que são subtipos um do outro dado que a classe B é subclasse de A. Os métodos `m2()` devolvem, respectivamente, `String` e `double` que, tal como o compilador indica, não são relacionáveis (Figura 7.9).

Note-se que o mesmo problema surgiria se uma classe declarasse implementar as interfaces `Inter1` e `Inter2`, pelo que as regras se aplicam a ambas as situações.

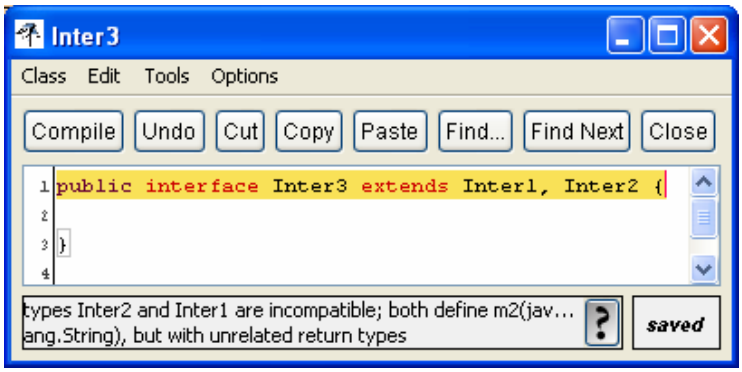


**Figura 7.9 – Colisão de tipos na herança de métodos**

Vamos reescrever o `m2()` de `Inter2` para `String m2(String n1, String n2)` e recompilar. Já sem erros, vamos criar a classe `Imp` que vai implementar `Inter3`.

```
public class Imp implements Inter3 {
   public A m1(int a, int b) { return new A(); }
   public String m2(String a, String b) { return ""; }
}
```

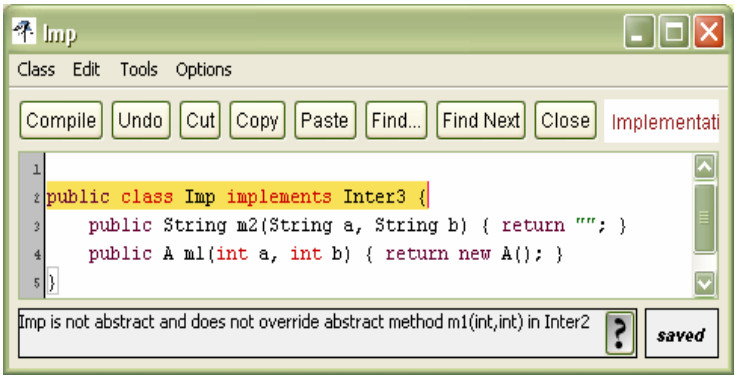


**Figura 7.10– Colisão de tipos na herança de métodos**

A compilação dá erro, indicando que a classe `Imp` é abstracta pois não implementou o método `m1()` de `Inter2`. Porém, ou a classe abdica de implementar a interface `Inter3` ou o método `m1()` de `Inter2` tem mesmo que ser implementado.

Vamos então implementá-lo como sendo:

```
public B m1(int a, int b) { return new B(); }
```

o que não deverá conduzir a bons resultados pois ficaremos com dois métodos de igual assinatura e tipos de retorno diferentes, ou seja, incompatíveis.

A Figura 7.11 apresenta o resultado da compilação e a mensagem apresentada pelo compilador.

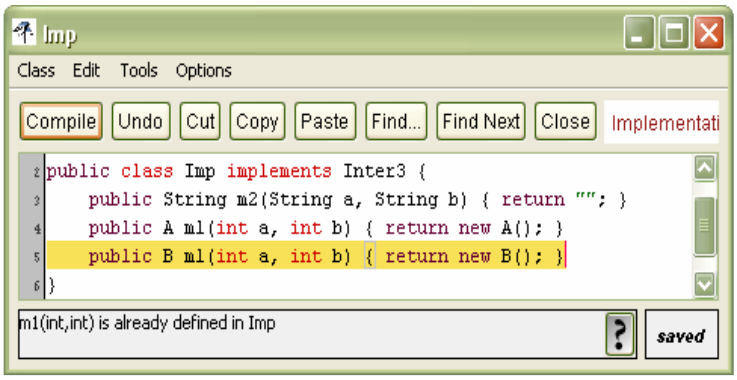


**Figura 7.11 – Dupla implementação**

Claro, o compilador indica que o método já está definido. Note-se que já temos definido um método `A m1(int, int)`, e que `B m1(int, int)` é uma **redefinição por covariância** do anterior. Assim, vamos ter que abdicar de um deles, pois não podemos implementar os dois métodos na mesma classe.

Porém, dado que o método `B m1(int a, int b)` possui um tipo de resultado que é subtipo de A, B <: A, então, este método acaba por satisfazer também a especificação de `A m1(int, int)` de `Inter1`, pelo que será apenas este que iremos implementar na classe `Imp` (ver Figura 7.12).

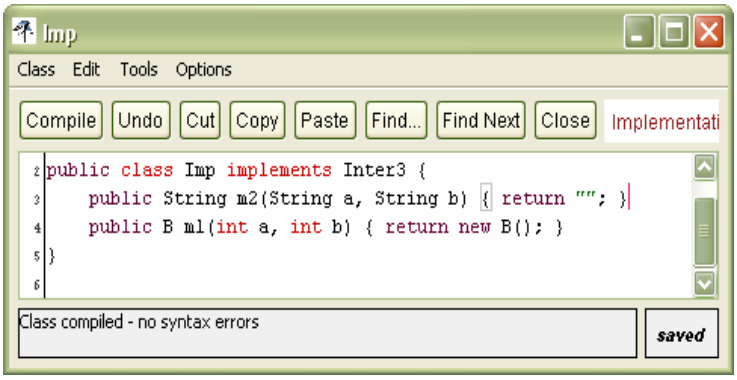


**Figura 7.12 – Construção final**

Assim, e em conclusão, numa situação de colisão para implementação entre dois métodos herdados por uma subclasse que possuem a mesma assinatura e o mesmo tipo de resultado, uma implementação satisfará várias interfaces. No caso de colisões em que os resultados são subtipos uns dos outros, temos que implementar o método que satisfaz as especificações de todos os outros, sendo este o método que devolve o resultado que, na hierarquia desses tipos, for o subtipo de todos os outros.

Resultados de tipos primitivos não estão abrangidos pela regra dos **tipos covariantes**, pelo que têm que ser iguais.

Vejamos um último exemplo. Considerem-se as seguintes interfaces:

```
public interface Colorivel {
   int getCor();
   void setCor(int cor);
}

public interface Pintavel extends Colorivel {
   void setTinta(String tinta);
   String getTinta();
}
```

e as classes das quais se apresentam apenas os métodos de interesse para o exemplo:

```
public class MeuPonto {
   public int getX() { return x; }
   public int getY() { return y; }
}

public class MeuPontoColorido extends MeuPonto
                              implements Colorivel {
   private int cor;
   public int getCor() { return cor; }
   public void setCor(int nvCor) { cor = nvCor; }
}
```

A definição de uma classe `MeuPontoPintado` que implementa a interface `Pintavel` deverá ser coerente com a estrutura das interfaces, tal como estas se encontram definidas na hierarquia.

```
public class MeuPontoPintado extends MeuPontoColorido
                             implements Pintavel {
   private String tinta;
   public void setTinta(String tinta) { this.tinta = tinta; }
   public String getTinta() { return tinta; }
}
```

A implementação da interface `Pintavel`, porque esta é subinterface de `Colorivel`, exige a codificação dos métodos definidos em ambas. A classe `MeuPontoPintado` apenas codifica os dois métodos de `Pintavel`. No entanto, como herda da sua superclasse os métodos `getTinta()` e `setTinta()`, acaba por, desta forma, satisfazer os requisitos de implementação de `Colorivel`. O código seguinte mostra a sua utilização:

```
MeuPontoColorido pc = new MeuPontoColorido();
Colorivel c = pc; c.setCor(1);
MeuPontoPintado pp = new MeuPontoColorido();
Pintavel p = pp;
p.setTinta("Texturada");
String tinta = p.getTinta();
```

As instruções anteriores mostram a compatibilidade entre classes e subclasses, tal como determinada pela hierarquia de classes, bem como a compatibilidade de tipos, tal como determinada pela hierarquia das suas interfaces.

## 7.8 INTERFACES PREDEFINIDAS DE JAVA

A linguagem JAVA usa de forma intensa o mecanismo das **interfaces** como meio de especificação de funcionalidades e propriedades implementáveis por algumas das suas classes predefinidas, bem como pelas classes definidas pelo utilizador. Algumas destas interfaces de JAVA assumem um papel importante no desenvolvimento de aplicações reais, pelo que se torna imperioso analisá-las.

Vamos nesta secção apresentar as interfaces `Serializable` e `Cloneable` que especificam funcionalidades (ou tipos) muito importantes para as aplicações desenvolvidas em JAVA, bem como algumas das outras interfaces de JAVA que, por serem parametrizadas, estudaremos no próximo capítulo onde abordaremos os tipos parametrizados.

### 7.8.1 A INTERFACE `Serializable`

Em geral, os objectos criados ao longo de um programa, e que são responsáveis pela computação desenvolvida, dado existirem apenas em memória central, não são capazes de sobreviver ao fim do próprio programa. Assim, se pretendermos que tais objectos sobrevivam ao final do programa, deveremos torná-los possuidores de **persistência** (ou perenidade), em particular garantindo que são "serializáveis".

Em JAVA, todas as variáveis não `static` e não `transient` são serializáveis. A interface `Serializable` é uma *marker interface*, ou seja, é uma interface vazia que não impõe a implementação de qualquer método (cf. `Regravavel`, no exemplo atrás). Portanto, qualquer objecto é serializável em toda a porção do seu estado que não corresponda a atributos `transient` (em geral, não usados). Atributos `static` não são das instâncias.

As instâncias de classes que declaram que implementam a interface `Serializable` podem ser gravadas em memória secundária sob a forma de objectos, usando *streams* particulares designadas por `ObjectStreams` (ver capítulo das *Streams*) que as guardam em ficheiro de forma a poderem ser mais tarde reconstruídas em memória.

Para que uma qualquer das classes que criámos até ao momento possa ter as suas instâncias gravadas como objectos em memória secundária de forma especial (**serializadas**) e em qualquer momento lidas para memória central de novo (**desserializadas**), usando métodos apropriados das *ObjectStreams* (como `writeObject()` e `readObject()`), com todas as eventuais referências a outros objectos recompostas, basta que nos seus cabeçalhos se escreva:

```
public class C implements Serializable {
```

Deixaremos as questões relacionadas com a efectiva gravação e leitura dos objectos em ficheiro através das *ObjectStreams* para o capítulo específico de *Streams*.

### 7.8.2 A INTERFACE `Cloneable`

O método `java.lang.Object.clone()` é um método `protected`. Nas primeiras versões de JAVA, quando a linguagem ainda era chamada de *Oak* e tinha como objectivo principal a programação de *embedded systems*, o método era público, e todos os objectos poderiam ser "clonados" por uma classe qualquer. Tendo-se JAVA tornado uma linguagem de larga utilização e de propósitos gerais, com preocupações de segurança de código reforçadas, o método `clone()` foi posteriormente tornado `protected`.

A classe `Object`, da qual todas as classes são subclasses, contém a implementação nativa de `clone()`. Este método nativo, portanto escrito na linguagem da máquina que executa JAVA, calcula a memória a alocar para a cópia de um dado objecto, e, em seguida, copia os *bytes* do objecto da antiga localização para uma nova. O novo endereço é devolvido associado a um objecto do tipo `Object`, pois o método não tem acesso a informação de tipos e não sabe qual o tipo do objecto que está a copiar. O programador que invocou o método `clone()` deverá em seguida realizar o *casting* de `Object` para o tipo do objecto que acabou de ser copiado (ex.: `Tipo inst1 = (Tipo) inst.clone();`).

Vimos também, anteriormente, que este método `clone()`, pelas razões acima indicadas, não faz *deep copy*, apenas *shallow copy*, porque se no objecto a copiar (e nas suas superclasses) existirem referências para outros objectos, o método copia esses apontadores para a cópia mas não os objectos que são os conteúdos de tais endereços. Portanto, cópias de instâncias que possuam atributos que sejam objectos passam a partilhar tais objectos, pois recebem nos respectivos campos a cópia dos endereços destes, ficando, assim, a referenciá-los também. Os objectos que compõem o objecto copiado ficam a ser partilhados pela cópia e pelo original.

Apesar desta ineficácia, a utilização do método `clone()` numa dada classe está ainda sujeita a certas regras. A primeira regra é a de que apenas poderão invocar o método as classes que declararem que implementam a interface `Cloneable` (não se escreve *Clonable*). A própria classe `Object` não implementa a interface pelo que não se podem realizar cópias de instâncias de `Object`. Se uma classe tentar invocar o método sem ter declarado que implementa a interface, um erro de execução `CloneNotSupportedException` será gerado pelo código de `Object.clone()`.

A interface `Cloneable` é outra *marker* interface que tem a seguinte forma:

```
public interface Cloneable {
```

servindo apenas para indicar ao método `Object.clone()` que o programador da classe que está a invocar o método autoriza que as instâncias da classe possam ser clonadas desta forma.

A segunda regra para uma correcta implementação de `Object.clone()`, consiste em fazer a sua reescrita na classe que pretende ver as suas instâncias clonáveis, redefinindo-o como sendo `public`, pois caso a classe mantivesse o modificador `protected` original, classes de outro *package* não poderiam invocar o método `clone()` nela redefinido. No entanto, o método redefinidor invoca `Object.clone()`.

Assim, a forma geral para a escrita do método `clone()` numa classe que pretenda usar o método `clone()` de `Object` como método auxiliar de cópia, será a seguinte:

```
public class X implements Cloneable {
   // Variáveis de Instância de X sem objectos ou
   // apenas com objectos imutáveis !!
   . . .
   public X clone() { // redefine Object.clone()
      X copiaDeMim = null;
      try {
           copiaDeMim = (X) super.clone();
      }
      catch(CloneNotSupportedException e) { }
      return copiaDeMim;
   }
}
```

A excepção `CloneNotSupportedException` nunca deverá ocorrer porque a classe declara implementar `Cloneable`. Mas como o método `Object.clone()` pode lançar tal excepção, o método invocador tem que usar a construção *try/catch* (ver capítulo sobre *Excepções* de JAVA), caso contrário não compila.

Depois de todo este processo, repare-se que o código acima apresentado apenas é aplicável a classes cujas instâncias sejam de tipos primitivos, de tipos referenciados imutáveis ou elas próprias imutáveis. Não havendo em JAVA uma construção do tipo `const` que nos garanta tal semântica para um dado símbolo, terá sempre que ser por análise do problema, que se reflectirá no código, que poderemos concluir sobre a imutabilidade de um dado objecto.

Assim, se a classe contiver variáveis de instância de tipos referenciados mutáveis, o que é muito comum, então, depois de estar clonada correctamente a porção imutável do objecto usando o método `clone()` de `Object`, teremos que, "à mão", modificar o valor dos campos que estão mal copiados, para tal escrevendo código que permita realizar a clonagem correcta das variáveis de instância "mal" clonadas (serão sempre as variáveis de tipos referenciados que podem mudar de valor). Quando as classes destas variáveis têm o método `clone()` definido, de novo ele poderá ser usado, ficando novamente por corrigir os campos de tipos referenciados, e assim recorrentemente.

Como a maioria dos métodos `clone()` das classes de JAVA usa `Object.clone()`, tais

métodos fazem apenas *shallow copies*. Quando assim é, e é quase sempre, temos que acabar por escrever código que copie o objecto correctamente campo a campo.

Em conclusão, nada lucramos com a utilização de `clone()` da classe `Object`, a não ser adiar o mais possível o inexorável algoritmo de criarmos uma instância do mesmo tipo da que pretendemos copiar (alocação de memória a alto nível) e clonarmos os campos referenciados mutáveis usando o método `clone()` das respectivas classes, caso estes façam *deep copy*, ou copiando-os "manualmente" sem usar `clone()` em última instância (ver no capítulo 8 a clonagem de **colecções**). Nem sequer a justificação de as classes poderem ter a etiqueta `Cloneable` se justifica, pois esta etiqueta apenas serve de informação para o método `Object.clone()`, podendo ser usada mesmo não invocando este método.

Portanto, embora saibamos que não utilizamos o `clone()` de `Object`, nada nos impede de declararmos que as nossas classes implementam `Cloneable`. Tal é até positivo para quem as possa usar, pois sabe de imediato que tem um método de nome `clone()` à sua disposição, sendo-lhe irrelevante (por abstracção) saber qual a implementação do mesmo.

Note-se que, ao tomarmos a decisão de designarmos por `clone()` o nosso método que realiza a *deep copy* de objectos sem invocar `Object.clone()`, sabemos que estamos a redefinir o método herdado e que as subclasses das nossas classes irão herdar o nosso método `clone()`. Assim, para estas, a invocação de `super.clone()` nunca corresponderá à utilização do método `Object.clone()`.

Ainda que não seja por nós utilizado, apresenta-se de seguida o padrão a seguir para uma utilização correcta do método `Object.clone()` numa situação que reproduz todas as possibilidades. A classe tem variáveis de instância primitivas, de tipo referenciado mas imutáveis, e mutáveis de tipo referenciado. Os identificadores das variáveis procuram ser elucidativos do tipo de situação que representam, e consideram-se disponíveis os métodos `get()` e `set()` para cada variável de instância mutável:

```
public class CloneEx implements Cloneable {
   // Variáveis de Instância
   int primt1;
   double primt2;
   Integer refim1;           // String e Integer são imutáveis
   String refim2;            // por definição
   String refString;         //
   TipoRef1 refmut1;         // Possui método clone()
   TipoRef2 refmut2;         // Este campo tem que ser copiado
   // Método
   public CloneEx clone() {  // redefine Object.clone()
      CloneEx copia = null;
      try {
          copia = (CloneEx) super.clone();
      }
      catch(CloneNotSupportedException e) {}
      // primt1, primt2, refim1, refim2, refString ok!!
```

```
      // Classe TipoRef1 tem clone()
      copia.setRefMut1( refmut1.clone() );
      // Classe TipoRef2 não tem clone(), tem que se usar
      // construtor com cada campo de refmut2 copiado.
      copia.setRefMut2( new TipoRef2( refmut2.getC1().clone(),
                        refmut2.getC2().clone(), ... ));
      return copia;
   }
}
```

Depois de feita a *shallow copy*, os campos `refmut1` e `refmut2` da variável `copia` têm que ser, portanto, modificados. Como a classe `TipoRef1` tem um método `clone()` supostamente *deep*, este é usado para copiar `refmut1` para o respectivo campo de `copia`. A classe `TipoRef2` não tem o método `clone()` implementado, pelo que teremos que usar um construtor completo para criar uma instância de `TipoRef2`, passando como parâmetros as cópias dos valores de cada um dos campos de `refmut2`, e assim sucessivamente.

Na nossa abordagem, o **esquema de código** do método `clone()` será:

```
public CloneEx clone() { // redefine Object.clone()
   // Classe TipoRef1 tem clone()
   TipoRef1 vref1 = refmut1.clone();
   // Classe TipoRef2 não tem clone(), tem que se usar
   // construtor com cada campo de refmut2 copiado.
   TipoRef2 vref2 = new TipoRef2( refmut2.getC1().clone(),
              refmut2.getC2().clone(), ... );
   return new CloneEx(primt1, primt2, refim1, refim2,
                refString, vref1, vref2);
}
```

Analisaremos a questão da realização de cópias de objectos inúmeras vezes ao longo dos restantes capítulos e durante a análise dos vários exemplos apresentados.

A clonagem de colecções de objectos será abordada particularmente no capítulo seguinte, sendo no entanto desde já de referir que a maioria das operações de `clone()` sobre colecções é de tipo *shallow*.

JAVA tem inúmeras implementações de classes que se comportam como *containers* de objectos, e oferecem os métodos necessários para inserir, ler, procurar, remover e iterar objectos dentro da colecção. Todas as colecções possuem um método `clone()` que pode ser invocado, e cuja forma de funcionamento *shallow* se apresenta a seguir.

Por exemplo, se tivermos uma colecção de objectos de um tipo predefinido de JAVA associada à variável `coll1`, tal como ilustrado na Figura 7.13, o resultado da execução de `coll1.clone()` é a criação de uma estrutura de apontadores igual à de `coll1`, numa outra zona de memória, para onde são copiados os conteúdos de `coll1` (que são endereços) mas não os objectos referenciados por tais endereços, conforme se procura ilustrar com a Figura 7.14.

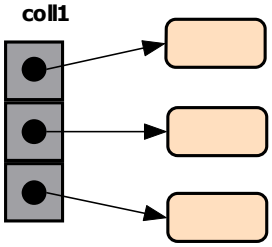


**Figura 7.13 – Representação de uma colecção de objectos**

Ora sendo os conteúdos de uma qualquer colecção os endereços dos objectos que a colecção contém, uma *shallow copy* vai colocar na nova zona de memória alocada apenas os endereços dos objectos referenciados por `coll1`.

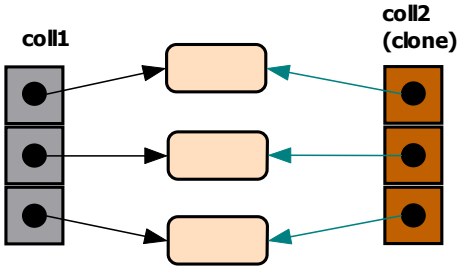


**Figura 7.14 – clone() de uma colecção de objectos**

Assim, o resultado de `coll2 = coll1.clone()` será uma nova colecção, `coll2`, do mesmo tipo da primeira, num novo espaço de memória, mas que partilha todos os objectos de `coll1`.

Inserir elementos em `coll1` não modifica `coll2`. Inserir elementos em `coll2` não altera `coll1`, mas outras operações (ex.: modificadores dos objectos) podem provocar alterações numa das colecções através da utilização da outra, o que é muito desaconselhável. As duas colecções estão "demasiado" interligadas entre si, ou seja, não são entidades verdadeiramente independentes, como seria desejável (Figura 7.15).

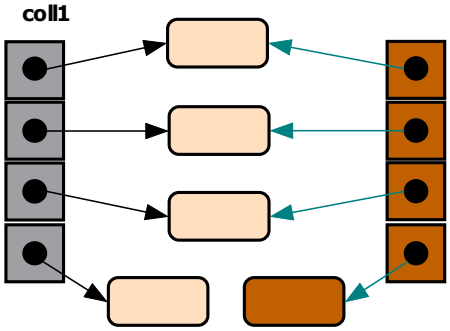


**Figura 7.15 – Colecções "demasiado" interligadas**

Os erros resultantes desta forte interligação entre as estruturas são de muito difícil detecção, tanto mais que cada `clone()` produz uma estrutura semelhante, sendo incontrolável saber qual destas provoca modificações não esperadas no original.

No caso de estarmos a falar de variáveis de instância, estas situações são absolutamente proibidas, pois são a antítese do encapsulamento e da protecção desejados.

Em conclusão, também para colecções, o `clone()` predefinido, sendo *shallow*, não nos garante segurança quanto ao princípio do encapsulamento e protecção de acesso aos nossos objectos, pelo que acabaremos sempre por ter que programar a sua *deep copy*.

## 7.9 INTERFACES PARAMETRIZADAS

As interfaces e as colecções de JAVA5 são genéricas, ou seja, são parametrizadas, tendo uma **variável de tipo** como parâmetro, logo, uma variável que será instanciada com um tipo de dados concreto.

A **parametrização** será estudada com detalhe no próximo capítulo, mas, porque no caso das interfaces a sua compreensão em termos gerais não é complexa, vamos nesta secção referir algumas importantes interfaces parametrizadas de JAVA.

A interface **`Comparable<T>`** está definida da seguinte forma genérica:

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

Claro que nenhuma classe implementa a interface `Comparable<T>` mas sim instanciações desta interface para valores concretos do tipo formal `T`. Assim, para `T = String`, por exemplo, teremos a seguinte interface concreta:

```
public interface Comparable<String> {
   int compareTo (String obj);
}
```

O método `compareTo(T obj)` tem como requisitos que o objecto parâmetro seja comparado com o receptor, devendo devolver como resultado um valor positivo se o receptor for maior (tipicamente 1), um valor negativo se for menor (tipicamente -1) e 0 se for igual ao objecto parâmetro. Claro que a igualdade deverá ser consistente com a implementação de `equals()` para o tipo `T`, ou seja, se `equals()` é `true` então `compareTo()` deve dar como resultado 0.

Quanto à comparação, sem dúvida que um objecto de tipo `T`, no mínimo, deverá ser comparável com outro objecto de tipo `T`, pelo que, para `T = String`, então o receptor da mensagem `compareTo(s)` deverá ser outra `String`. Para que tal seja possível, a classe `String` deverá implementar a interface `Comparable<String>`.

As classes *wrapper*, `Integer`, `Double`, `Boolean`, etc., bem como as classes nucleares

de JAVA que necessitam de ter uma ordem nos seus elementos (cf. maior, menor e igual), todas implementam as "respectivas" instanciações da interface `Comparable<T>`, que são `Comparable<Integer>`, `Comparable<Date>`, `Comparable<Time>`, etc.

Esta interface, quando implementada por uma dada classe, garante a quem a usa que as instâncias possuem uma **ordem total** estabelecida nos seus elementos, designada a sua **ordem natural**, segundo o método de comparação natural `compareTo()`. A existência desta ordem é muito importante quando temos que ordenar objectos ou utilizá-los em certas estruturas de dados que necessitam de os comparar com elementos já inseridos para determinar a sua posição de inserção, etc.

Algumas das classes anteriormente desenvolvidas "mereceriam" possuir tal ordem definida nos seus elementos, por exemplo, para que os mesmos pudessem ser ordenados por ordem crescente ou decrescente, como, por exemplo, as classes `Ponto2D` e `Pixel`.

Para tal, teríamos que acrescentar ao cabeçalho de tais classes as declarações seguintes:

```
public class Ponto2D implements Serializable, Cloneable,
                                Comparable<Ponto2D> {

public class Pixel implements Comparable<Pixel> {
```

e acrescentar às classes a implementação dos métodos `int compareTo(Ponto2D p);` e `int compareTo(Pixel px);`, respectivamente.

A interface **`Comparator<T>`** é semelhante à anterior, porém, difere na forma de especificação do método de comparação, conforme se pode verificar pela sua definição:

```
public interface Comparator<T> {
   int compare(T obj1, T obj2);
}
```

Enquanto que `Comparable<T>` define uma propriedade que a classe declara possuir, daí o seu identificador ser um adjectivo, a interface `Comparator<T>` define um mecanismo que a classe implementa e disponibiliza, daí o seu identificador ser um verbo e o método possuir dois parâmetros (não existindo receptor!). Trata-se de um serviço que a classe tem implementado para fornecer ao exterior para comparar dois objectos do seu tipo.

O método `compare(T o1, T o2)` tem a mesma semântica do método `compareTo()` tomando agora os dois parâmetros como elementos da comparação.

Se uma classe pretende implementar e passar a fornecer um **comparador** de um dado tipo, então deverá declarar que o implementa e codificá-lo. Por exemplo, a classe:

```
public class StringComp implements Comparator<String> {
   public int compare(String s1, String s2) {
      return (s1.toUpperCase()).compareTo(s2.toUpperCase());
}
```

implementa o método `compare()` sobre *strings*, convertendo os caracteres de cada uma em maiúsculas e invocando em seguida o método `compareTo()` da classe `String`.

Uma classe que necessite de usar um `Comparator<String>` apenas terá que escrever:

```
Comparator<String> cpString = new StringComp();
```

ou seja, criar uma instância do comparador, e usar `cpString` (que é o comparador) onde quer que este seja necessário, por exemplo, como parâmetro de um método de classe (da classe `ProcStrings`) que aceita um *array* `String[]` e um `Comparator<String>`, e devolve um *array* ordenado de *strings*:

```
String[] nmsOrd = ProcStrings.sortArrStr(nomes, cpString);
```

A criação destes comparadores será de novo abordada no capítulo das colecções, dado serem imprescindíveis na utilização de colecções que mantêm os seus elementos organizados através da sua ordem natural definida em tais algoritmos de comparação.

## 7.10 INTERFACES PARAMETRIZADAS DE COLECÇÕES

As interfaces parametrizadas usadas em JAVA5 para representar os vários **tipos** das colecções de objectos da linguagem são apresentadas na Figura 7.16.

Se repararmos nos nomes das interfaces e na sua hierarquia, verificamos de imediato que esta hierarquia representa, de forma genérica porque `E` é um tipo parâmetro que pode ter inúmeras instanciações concretas, as grandes famílias de colecções que estão previamente catalogadas em JAVA.

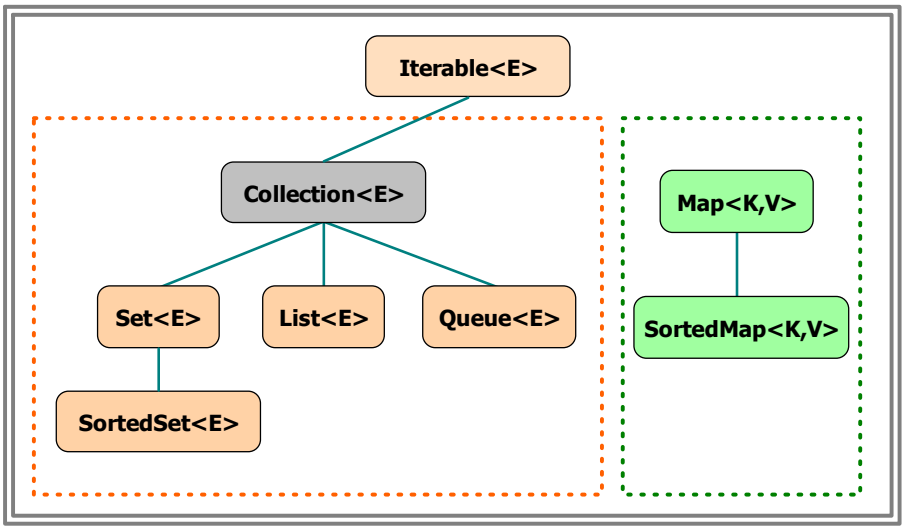


**Figura 7.16 – Interfaces e tipos das colecções de JAVA**

Esta hierarquia genérica, em que cada nodo representa uma família infinita de possíveis interfaces, para um dado valor de `E`, por exemplo `E = String`, transforma-se numa hierarquia de interfaces concretas que representam os vários tipos de colecções que podemos

ter com elementos do tipo `String`.

Teríamos classes do tipo `Collection<String>` (de momento não interessa a API), que são o supertipo de subtipos como `Set<String>`, `List<String>` e `Queue<String>`, que pelos nomes sabemos serem conjuntos, listas e filas de *strings*.

Ora, como todas estas interfaces são **tipos**, existirão então classes que irão implementar cada um destes tipos. Teremos então certamente várias classes do tipo `Set<String>`, ou seja, várias implementações de conjuntos de *strings*, e várias implementações do tipo `List<String>` e do tipo `Queue<String>`, etc. Tudo isto para `E = String`, tal como ocorrerá para outro qualquer valor concreto de `E`. Daí a razão para a sua **parametrização**.

Qualquer que seja a instanciação efectiva dos seus parâmetros, a hierarquia de tipos definida por esta hierarquia de interfaces mantém-se. Veremos como a hierarquia de tipos se relaciona depois com a hierarquia das suas implementações (classes).

As colecções parametrizadas de JAVA5 são o alvo do nosso estudo no capítulo seguinte.

## 7.11 SÍNTESE DO CAPÍTULO

As **interfaces** de JAVA são **especificações de tipos de dados**. Especificam, fundamentalmente o conjunto de operações que podem ser realizadas com as instâncias desse tipo. As classes são as entidades que implementam as interfaces. As classes que implementam interfaces, passam por isso a ter uma condição especial: são tipos por serem classes e são tipos por implementarem as interfaces.

Uma instância de uma classe que implemente uma interface, é de imediato compatível com dois tipos distintos: o da classe e o da interface. Porém, quando uma instância de uma classe é associada a uma variável do tipo da interface, apenas os métodos definidos para tal tipo podem ser empregues.

Sendo interfaces tipos, definem uma hierarquia que é claramente uma hierarquia de tipos, e as classes que satisfazem tais interfaces passam a fazer parte dessa hierarquia de tipos para além de fazerem parte da sua natural hierarquia de classes. Uma simples declaração ***implements***, e uma implementação de métodos, torna a classe membro dessa hierarquia de tipos. No entanto, uma interface pode não fazer parte de qualquer hierarquia, sendo livre no espaço das interfaces. Porém, tal como qualquer classe, uma interface faz sempre parte de um *package*, e pode fazer parte da hierarquia de interfaces que tem a propriedade de permitir herança múltipla.

Vimos ainda neste capítulo que certas interfaces de JAVA5 são parametrizadas, o que significa que cada interface parametrizada representa uma família de possíveis interfaces concretas, uma por cada instanciação do seu parâmetro. As interfaces são um mecanismo extremamente relevante em JAVA, tal como veremos no capítulo seguinte.