Fundamentos de linguagem Java

Sérgio Lopes

Fundamentos de linguagem Java

Sérgio Lopes

Índice

Sobre o Manual

Público Alvo

O presente manual destina-se ao módulo *0789 - Fundamentos de linguagem Java* do catálogo da ANQ, www.catalogo.anq.gov.pt.

Estrutura e Conteúdo

É explicada a tecnologica Java e os seus diferentes componentes, são depois apresentados os conceitos teóricos de programação orientada a objectos e a forma como a linguagem de programação Java faz uso e implementa esses conceitos.

O conteúdo do manual serve como base introdutória à linguagem de programação Java e fornece todos os conceitos teóricos necessários para se aprofundar a programação com objectos em linguagem Java. No entanto, nem o módulo nem o manual, contêm uma exposição completa de toda a tecnologia Java existindo muitos componentes que não são abordados.

Licença

Capítulo 1. Definir "Java"

O termo Java é acompanhado de alguma ambiguidade e desinformação, quer por culpa da publicidade e marketing de que foi alvo, quer pelas confusões naturais da *Internet* e dos seus utilizadores. Neste ponto pretende-se explicar o que é, e o que não é o Java, e definir a que nos referimos quando dizemos *"Java"*.

Java é um termo que designa uma **linguagem de programação**, uma **plataforma** onde o código dessa linguagem é executado, e um **conjunto de bibliotecas** que permitem o desenvolvimento de aplicações usando a linguagem. Para se programar em linguagem Java todos estes três pontos são usados, e o seu significado está intimamente ligado.

Java, como plataforma é o sistema responsável pela execução, muitas vezes compilação, do código que compõe a aplicação. É um software desenvolvido noutra linguagem, a mais comum sendo C++, que é responsável pela interpretação do código escrito na linguagem Java, e que fornece o ambiente necessário à execução dos programas feitos na linguagem Java. Este software, conhecido como **JVM**, *Java Virtual Machine*, é, para todos os efeitos, uma máquina virtual dentro do sistema operativo que faz a gestão de memória, iniciação e término das aplicações, e todas as tarefas necessárias para que uma aplicação feita na linguagem Java possa correr.

Java, como linguagem de programação assenta nas tecnologias acima mencionadas e oferece uma sintaxe similar a C e C++, com um modelo de objectos mais simples que C++ e com menos funções de baixo nível que C ou C++. Possui uma biblioteca de funções extremamente grande, composta por centenas de classes, que permitem ao programador ter acesso a um vasto conjunto de funcionalidades que pode usar na criação de software.

Existem várias versões da plataforma Java, da linguagem Java e da JVM. Versões para software empresarial, *Java EE*, que fornecem um conjunto extra de bibliotecas e tecnologias, para dispositivos móveis, *Java ME*, com conjunto de funcionalidades e consumo de recursos reduzido. Neste manual iremos focar apenas o *Java SE*, actualmente o *Java SE 6*.

Java Como Plataforma

Foram criadas várias implementações da plataforma Java, a mais comum em computadores pessoais com sistema operativo Windows será certamente a implementação desenvolvida pela Sun footnote: [A empresa Sun Microsystems foi adquirida em 2010 pela empresa Oracle. Ao longo do manual poderá surgir tanto o nome Sun como o nome Oracle como empresa detentora/produtora do Java.]. Mas existem também implementações criadas pela IBM, pela Oracle e outros projectos que surgiram com a passagem da tecnologia para software livre, sob uma modificação da GNU GPL [1].

Todas estas plataformas possuem diferenças, mas todas elas possuem um conjunto de funcionalidades base que permitem que uma aplicação, desenvolvida usando uma implementação, possa ser executada noutras implementações com o mínimo de alterações. Se uma implementação é identificada como sendo 100% compatível, então essa implementação foi sujeita um programa de testes que garantem que segue os padrões definidos pela Sun e que uma aplicação Java poderá ser executada sem qualquer tipo de alteração ou limitação das suas funcionalidades.

Consoante o objectivo, a plataforma está dividida em:

[1] GNU General Public License, licença de software livre gerida pela Fundação de Sofware Livre, www.fsf.org

- **Java Standard Edition** - destinada a computadores pessoais, sendo a plataforma mais comum;
- **Java Enterprise Edition** - destina a aplicações empresariais [2], e que fornece um conjunto mais alargado de bibliotecas, nomeadamente no que toca a acesso a rede, comunicação com *Web Services*, etc.;
- **Java Micro Edition** - usada por dispositivos com recursos limitados, como PDAs, telemóveis ou até usada em equipamentos de domótica, e é composta por um conjunto reduzido de bibliotecas e por optimizações no que toca a consumo de recurso;

Java Como Linguagem

Ao nível da linguagem, podemos considerar o Java como mais uma das linguagens de programação que usa uma sintaxe derivada do C, onde as diferentes versões introduzem apenas novas funcionalidades.

Até à versão 1.4, a sintaxe da linguagem manteve-se praticamente constante. Sendo as alterações pouco significativas para quem aprende a linguagem neste momento. Mas da versão 1.4 para a versão 5 [3], foi adicionado o suporte para **Genéricos**, uma funcionalidade importante da linguagem que quebra alguma da compatibilidade gozada nas versões anteriores.

Versões

Uma versão de Java indica sempre características alteradas em toda a tecnologia, mesmo que apenas alguns componentes tenham sofrido modificações. Assim, ao referirmos a versão 6 do Java, estamos a indicar que a versão se aplica à JVM, à linguagem, ou à plataforma.

A nomenclatura das versões Java sofreu várias alterações durante os anos, reflectindo ideias diferentes ao longo de desenvolvimento, e causando alguma confusão na forma correcta de nos referirmos a uma determinada versão. Isto porque cada versão de Java é composta por, pelo menos, três formas diferentes de identificação: o *número de desenvolvimento*, *o número da versão* e o *nome de código*.

As versões cujo número de desenvolvimento é inferior a 1.4 eram referidas apenas por esse número de desenvolvimento, ex: Java 1.3. A partir da versão com o número de desenvolvimento 1.4, foi adicionado "Java 2" ao nome, ex: Java 2 SE 1.4. A partir da versão com número de desenvolvimento 1.5, a Sun decidiu retirar o nome "Java 2" e deixar de usar o número de desenvolvimento, passando a existir o Java 5, cujo número interno era 1.5.

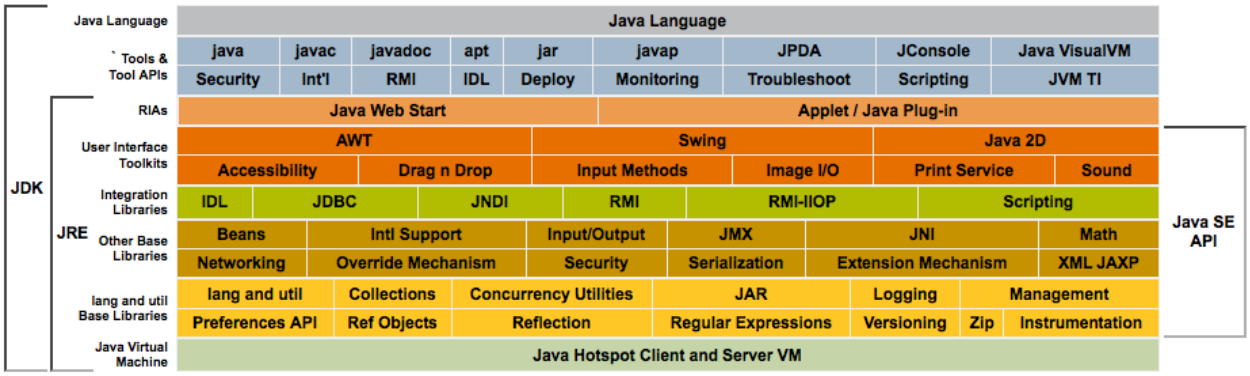
Com o Java 6, foram adicionados novos elementos à nomenclatura, os números de actualizações. Estes indicam actualizações importantes mas que não alteram nenhuma das APIs [4] públicas da plataforma. Um exemplo é a actualização 10 (Java SE 6 update 10), que introduziu profundas melhorias a nível de usabilidade e performance.

Assim, na altura em que este texto é escrito, a versão do Java possui um número de desenvolvimento **1.6**, número de versão **6**, nome de código **Mustang** e número de actualização **22**, sendo comummente designada por **Java 6**.

[2] Entende-se por aplicações empresariais, sistemas que lidam com grandes quantidades de dados, criadas com o objectivo de resolver problemas ao nível da empresa, cujas funcionalidades estão orientadas para determinado negócio
[3] Na secção seguinte é explicado o significado das diferentes versões mencionadas
[4] Application Programming Interface.

Visão Geral da API

A imagem seguinte permite ter uma visão geral dos *packages* e componentes da plataforma Java



Como se pode ver na imagem, a plataforma Java oferece várias classes organizadas por funcionalidade, e que permitem desenvolver aplicações com bastante rapidez.

Capítulo 2. Ferramentas e Ficheiros Necessários

Como em qualquer outra tecnologia, também para se usar Java serão necessárias algumas ferramentas de desenvolvimento. Nomeadamente o Java JDK, o pacote de bibliotecas que constitui a plataforma Java e sem o qual não é possível criar aplicações Java.

Java SDK ou JDK

Como já foi mencionado, existem várias implementações e versões da plataforma Java. Para executar aplicações Java, é necessário ter instalado e correctamente configurado, o Java Runtime Engine, ou JRE.

O JRE é a plataforma Java que as aplicações usam para serem executadas e incluí, entre outras coisas, a JVM que irá executar as nossas aplicações, as classes que as nossas aplicações precisam para funcionar, e todo um conjunto de ferramentas destinadas à correcta execução das aplicações.

Mas, e para criar aplicações? Bem, nesse caso precisamos do Java Development Kit, que, segundo a Oracle, é um subconjunto do Software Developement Kit usado para criar aplicações em Java. Tipicamente, é comum referirmos apenas o JDK, sendo que o SDK incluí coisas como servidor de aplicações, que não são obrigatoriamente necessárias durante o desenvolvimento.

Para o nosso tutorial, vamos usar o JDK criado pela Oracle, mas qualquer JDK que seja 100% compatível pode ser usado. O JDK da Oracle pode ser obtido em http://www.oracle.com/technetwork/java/javase/downloads/index.html

IDEs e outros Editores

Além do JDK, precisamos de uma forma de criar e editar os ficheiros de código fonte, em Java, qualquer editor de texto será suficiente, por mais simples que seja. Todas as restantes ferramentas necessárias fazem parte do pacote de download do JDK footnote: [Isto é verdade para o JDK da Oracle, outras implementações podem exigir mais configurações ou downloads separados.]. Munidos destas ferramentas o leitor poderá iniciar já o desenvolvimento de software.

No entanto este tipo de ferramentas, apesar de úteis, pode limitar a produtividade do programador, e são comummente consideradas por programadores que estão a aprender, difíceis de usar. Não porque as ferramentas possam ser complicadas, mas porque obrigam a conhecer e decorar vários comandos e não oferecem ajudas para a sintaxe da linguagem. Por essa razão, poderá ser mais simples usar um Ambiente de Desenvolvimento de Software, ou IDE.

A disponibilidade de IDEs é elevada, cada um com as suas vantagens e desvantagens, sendo que, na maioria dos casos, o IDE que se usa não limita em nada o tipo de software que se pode criar [1].

Alguns dos IDEs mais comuns:

- blue, http://www.bluej.org/index.html

[1] Existem algumas excepções para tecnologias que estão presas a determinado IDE. Ex: A *Swing Application Framework* é implementada pelo NetBeans IDE mas não pelo Eclipse.

- JCreator, http://www.jcreator.com
- Eclipse IDE, http://www.eclipse.org
- JBuilder, http://www.codegear.com/products/jbuilder
- NetBeans IDE, http://www.netbeans.org

Para quem pretender usar um editor de texto mais simples, mas com suporte para linguagens de programação, aqui ficam algumas alternativas:

Windows	• Notepad++, http://notepad-plus.sourceforge.net
	• Notepad2,http://www.flos-freeware.ch/notepad2.html
GNU Linux	• gEdit, editor presente no Gnome. Apenas acessível a utilizadores de GNU Linux.
	• Kate, editor presente no KDE. Apenas acessível a utilizadores de GNU Linux.
Mac OS X	• Smultron, http://tuppis.com/smultron
	• TextWrangler, http://www.barebones.com/products/TextWrangler
	• MacVim, http://code.google.com/p/macvim/

Ao longo do manual não serão introduzidos quaisquer conceitos dependentes de um IDE específico, assim, é possível usar qualquer IDE ou editor apresentado nesta secção para escrever e compilar os exemplos, bem como treinar a utilização da linguagem de programação Java. Exemplos que acompanhem o manual serão desenvolvidos em NetBeans IDE.

Configuração das Ferramentas

A instalação e configuração das ferramentas a utilizar dependerá em grande forma do sistema operativo usado. A instalação em Windows deverá ser feita usando os ficheiros de instalação oficiais e seguidos todos os passos do assistente.

Os utilizadores de distribuições Linux poderão primeiro verificar se as ferramentas estão disponíveis nos repositórios oficiais da distribuição que usam, dado que várias distribuições incluem já pacotes de desenvolvimento com o JDK, quer o disponível pela Oracle, quer dos criados por projectos de software livre.

Os utilizadores de sistemas operativos OS X devem instalar o pacote de desenvolvimento, disponível no DVD de instalação do sistema operativo, ou para download em http://developer.apple.com/mac.

Outros sistemas operativos terão formas diferentes de instalar as ferramentas, nomeadamente o JDK.

Capítulo 3. Programação Orientada a Objectos

Programação Orientada a Objectos (**POO**) é dos pontos tidos como mais complicados na aprendizagem de Java, não sendo uma questão exclusiva de Java, é mesmo assim um ponto fundamental na sua aprendizagem e correcta utilização.

Java é uma linguagem orientada a objectos e que depende extensivamente desse paradigma, é fundamental a correcta compreensão do paradigma para que se possa programar e usar com sucesso a linguagem [1].

É importante compreender a teoria que sustenta a programação orientada a objectos, especialmente para se usar correctamente a linguagem de programação Java.

O Que É

A **P**rogramação **O**rientada a **O**bjectos pretende modelar os problemas aproximando-os da vida real e dos objectos que vemos no dia a dia. Consiste assim, em ver um programa como um conjunto de entidades, que interagem entre si através do uso de mensagens.

Olhando em volta, podemos ver todo um conjunto de objectos reais com que interagimos, sejam eles o computador onde trabalhamos, composto por vários outros pequenos objectos, seja a cadeira, secretária, ou qualquer outro material que se encontra à nossa volta. Estes objectos possuem várias características ou atributos como a cor, o volume, a forma, alguns o som que emitem, etc. E nós interagimos com estes objectos fazendo uso dos seus vários comportamentos. Por exemplo, usando a teclas do teclado enviamos mensagens para o computador de modo a escrever texto. Usando a maçaneta da porta abrimos ou fechamos a porta. Todos os objectos que nos rodeiam possuem um comportamento.

Os objectos que desenvolvemos em POO são similares no sentido em que também eles possuem atributos e comportamentos. Estes são os elementos que nós, como programadores, temos de criar.

A definição dos objectos é criada através de classes, os atributos são guardados usando variáveis, e os comportamentos é definido com a criação de métodos.

Uma Classe não é um Objecto

Apesar de falarmos sempre em objectos, e de pensarmos em objectos, ao programarmos e desenharmos o nosso sistema, usamos classes. Estas classes, embora intimamente ligadas aos objectos, não significam a mesma coisa. Esta distinção pode ser algo estranha de perceber.

Chamamos a uma classe a representação, o esquema, a planta ou até, o molde do objecto. A classe define o comportamento do objecto, os seus atributos, a relações que estabelece com outros objectos, mas é apenas uma representação. A classe é apenas o ficheiro de código e o código que escrevemos nele [2].

[1] Java não é uma linguagem orientada a objectos pura uma vez que possui tipos de dados primitivos, estes tipos de dados, embora possam ser encapsulados dentro de objectos, não o são nativamente.
[2] Em Java, um ficheiro tem tipicamente apenas uma classe.

Ao programarmos definimos um objecto através da construção da classe, essa construção é feita em código, onde dizemos quais os atributos que o objecto irá ter e que comportamentos irá expor aos outros objectos do programa. Mas esse objecto apenas existirá durante a execução do programa.

Um objecto existe em memória, contém valores no seus atributos, exprime um estado no nosso sistema. Por exemplo, se definirmos a classe **Roda**, com os atributo **raio** e o método **calcularDiametro**, estamos a definir como será um objecto do tipo *Roda*. Enquanto o nosso programa não estiver em execução, a classe só por si não faz nada, mas ao iniciarmos o programa, podemos instanciar o nosso objecto e dessa forma termos um objecto *roda* a partir da nossa classe.

Objecto ou Instância?

Objecto ou instância é o mesmo, nós obtemos um objecto através do processo de instanciação, daí podermos dizer que o que temos é um objecto ou uma instância de determinada classe.

Durante o restante manual tentaremos referir-nos sempre do mesmo modo, para não causar confusão, usaremos sempre *"objecto"* para nos referimos a um elemento com estado, que existe em memória no nosso computador durante a execução da aplicação. Evitaremos usar a palavra instância [3].

Resumo

Esta primeira abordagem a **POO** oferece apenas uma visão teórica da programação, mas essa visão teórica é importante para que a parte prática possa ser aplicada correctamente.

Nem sempre nos será possível ou útil seguir à risca o conceito de objectos, mas é importante perceber quando se pode ignorar a criação de um objecto ou quando é útil exagerar no número de objectos que o sistema tem. No fundo, tudo no nosso programa poderá ser transformado em objectos e é também possível desenvolver todo um programa em Java, perfeitamente funcional, usando apenas um ficheiro com o método **main**, mas tanto uma situação como a outra serão extremos para os quais a linguagem Java não é uma boa escolha.

Nas secções seguintes iremos abordar alguns pontos fundamentais de POO, que são importantes para que consigamos aplicar a teoria no nosso desenvolvimento prático: - Herança - Encapsulamento - Polimorfismo - Composição

E ver como os conceitos teóricos se aplicam à linguagem Java: - Classes - Utilização e Manipulação de Objectos - Interfaces - Conversões (Cast) - Modificadores de Acesso - Redefinição de Métodos

Herança

A herança, tal como na vida real, é o processo pelo qual os objectos ganham algumas das características dos seus pais. Embora, tal como na vida real, uma classe deva sempre introduzir comportamento e características próprias, através do mecanismo de herança é possível criar relações entre diferentes classes e reutilizar código. É um mecanismo muito importante em linguagens POO.

Como exemplo podemos pensar numa bicicleta. Se criarmos a classe *Bicicleta* e definirmos um conjunto base de atributos (**cor**, **tamanhoRoda**, **tipoTravao**) e um conjunto de métodos para o comportamento (**travar**, **pedalar**) podemos achar que estes atributos e métodos se aplicam a vários tipos de bicicleta. Uma bicicleta de montanha (*BicicletaMontanha*) terá os mesmos atributos, podendo

[3] A escolha de uma palavra por outra é meramente arbitrária, ambas designam a mesma entidade, ambas se referem ao mesmo.

acrescentar um novo (**tipoSuspensao**), e os mesmos métodos, acrescentando também o seu método novo (**mudarMudanca**). Estas duas bicicletas partilham código.

Se não usarmos herança termos de criar duas classes e implementar na classe Bicicleta os três atributos e os dois métodos, e na classe *BicicletaMontanha* termos de implementar, novamente, os três atributos mais o atributo novo e os dois métodos, além do método novo. Ora isto para o exemplo até poderia ser rápido mas o exemplo peca pela sua simplicidade, num sistema real a quantidade de objectos e código que se pode repetir é tanta que justifica podermos juntar tudo isto numa relação de herança.

Assim, o que fazemos é criar a classe *Bicicleta*, definir os três atributos e os dois métodos, e na classe *BicicletaMontanha* indicar que esta classe herda da anterior. Neste caso, na classe *BicicletaMontanha*, apenas nos resta implementar um atributo e um método [4].

Transformando em código [5]:

```
public class Bicicleta {

    public String cor;
    public int tamanhoRoda;
    public String tipoTravao;

    public void travar() {
    }

    public void pedalar() {
    }
}

public class BicicletaMontanha extends Bicicleta {

    public tipoSuspensao;

    public void mudarMudanca() {
    }
}
```

Encapsulamento

Este é o nome que damos ao processo de limitar o acesso às variáveis e os métodos de uma classe, ou de controlar o acesso aos atributos e métodos de um objecto.

O processo de encapsulamento permite que tenhamos algum controlo sobre quem acede aos métodos e de que forma podem afectar o estado interno dos nossos objectos. Isto é conseguido através do uso de **modificadores de acesso**, e afecta a forma como a herança funciona.

Se pensarmos num exemplo real, não nos é dado acesso ao interior dos nossos discos rígidos, aliás, não nos interessa se é composto por cilindros magnéticos, circuitos integrados ou que mais, apenas nos interessa que podemos guardar lá dados ou, no caso de estarmos a trabalhar com hardware, que tipo de conector é necessário para que possamos ligar o disco ao computador. A forma como os dados está guardada não nos é revelada, se o fosse, facilmente conseguimos destruir disco atrás de disco, ou danificar a informação de forma irrecuperável.

Do mesmo modo, não é conveniente que os objectos possam aceder livremente ao estado interno uns dos outros. Devem fazê-lo apenas através dos métodos que nós definimos (tal como nós acedemos ao

[4] As explicações sobre as classes surgem apenas nas secções seguintes
[5] Neste caso todos os métodos e atributos serão públicos, num caso real isso não seria uma boa opção

disco através do cabo específico), e devem poder alterar o estado interno de formas que por nós foram criadas (tal como nós acedemos aos dados através de drivers do disco).

O encapsulamento tem também o efeito de impedir que um programador mais preguiçoso baseie a sua implementação no conhecimento que tem de outro objecto, isto porque é possível, com um bom esquema de encapsulamento, definir apenas o comportamento importante para o exterior como visível e esconder outros pormenores que não sejam tão importantes footnote: [Convém relembrar que não escondemos o código, no entanto o programador só conseguirá aceder aos métodos para os quais tiver permissões, definidas através do uso dos modificadores de acesso.].

Como níveis de acesso, temos à nossa disposição três níveis base [6]: público, privado, protegido.

Todos os níveis permitem que o próprio objecto aceda, sem restrições ao seu estado e aos seus métodos, sendo que ao aplicarmos um nível de acesso estamos a afectar o que os outros objectos podem fazer.. O nível **público** permite que todos os atributos ou métodos sejam acedidos e modificados livremente, por qualquer objecto, quer seja uma subclasse (criada por herança) ou não. O nível **protegido** permite que apenas tenham acesso aos dados as subclasses. O nível privado é o mais restritivo de todos e impede o acesso completo ao estado e aos métodos. Métodos e atributos com modificador privado será visíveis apenas ao próprio objecto e nunca podem ser acedidos directamente por outros objectos.

Quando falamos em acesso e em aceder aos atributos estamos a referir-nos ao facto de que, ao programarmos a classe, não conseguimos invocar um método ou usar um atributo para o qual a classe onde estamos não tenha acesso.

Olhar Prático

Um correcto encapsulamento é um processo iterativo. Muitas vezes não conseguimos definir logo de início que tipos de acesso devem ser dados aos métodos ou aos atributos, mas como regra prática podemos considerar que todos os atributos devem ser privados, e que os métodos devem ser privados caso sejam usados internamente ou públicos caso sejam usados por outros objectos. O acesso protegido é, muitas vezes, desnecessário ou sinal de mau encapsulamento ou má análise do problema.

Como todas as regras práticas, esta deve ser sempre avaliada em presença de uma situação real, e certamente não se aplica a toda e qualquer situação.

Nunca devemos cair no erro de colocar tudo como público, só porque somos o único programador a desenvolver, ou porque facilita o desenvolvimento dado que escrevemos menos código. Mesmo que o programa esteja a ser desenvolvido apenas por nós, devemos usar correctamente as características da linguagem, em última análise, porque também nos ajuda impedindo que façamos alguma asneira por engano.

Polimorfismo

Polimorfismo é a capacidade de uma variável assumir o papel de diferentes tipos de objectos que se relacionem entre si através de uma classe base, oferecendo assim a possibilidade de redefinir o comportamento herdado sem alterar a interface de comunicação.

Tentando transpor a teoria para um exemplo, se considerarmos que **Joao** é subclasse de **Pessoa** então um objecto do tipo **Pessoa** consegue guardar um objecto do tipo **Joao**. Desta forma, uma variável que

[6] Em Java existe um quarto nível que iremos falar na secção de modificadores de acesso.

esteja definida como sendo do tipo **Pessoa** consegue guardar objectos do tipo **Pessoa**, **Joao**. De outra forma, uma variável consegue, através de polimorfismo, guardar objectos do seu tipo e de qualquer tipo descendente directo ou indirecto (netos, bisnetos, etc).

Ao usarmos polimorfismo podemos cair em situações onde não sabemos, como programadores, qual será o tipo de dados exacto que está guardado na variável, isso pode simplesmente não nos interessar se estivermos a aceder a métodos que existam em todos os objectos de determinada hierarquia, mas é necessário que o compilador e o interpretador consigam saber com exactidão quais os objectos que estão a ser manipulados.

Para esse efeito existem dois tipos de detecção: - *Static binding*, *binding* estático, em que o compilador determina, durante o processo de compilação, qual o tipo de objecto e que método deve ser invocado. - *Dynamic binding*, *binding* dinâmico, em que cabe ao interpretador determinar, durante a execução do programa, qual o tipo de objecto e que método invocar. Este processo aplica uma penalização de *performance* a todas as linguagens orientadas a objectos.

Caso seja necessário determinar o tipo de dados de forma programática, é possível usar o operador **instanceof** para determinar qual o tipo de dados, com alguma certeza, que está a ser manipulado. A exactidão da verificação dependerá das classe envolvidas. Por exemplo, assumindo a hierarquia *SerVivo > Mamifero > Humano e SerVivo > Mamifero > Golfinho*, vamos aplicar o operador algumas vezes

```
SerVivo ser;
Humano maria;
Golfinho saltitao;

ser instanceof Humano > false
ser instanceof SerVivo > true
maria instanceof SerVivo > true
maria instanceof Mamifero > true
maria instanceof Humano > true
saltitao instanceof Humano > false
saltitao instanceof SerVivo > true
```

Se verificarmos, a variável *saltitao* vai responder afirmativamente quando perguntamos se é um *SerVivo* ou se é um *Mamifero*, tal como a variável maria. Tendo uma variável que responda afirmativamente nestes dois casos não conseguimos saber se, além de *SerVivo* e *Mamifero*, ela é outra coisa qualquer, a não ser que experimentemos todas as combinações possíveis.

Usar o *instanceof* é, também, um processo muito lento e que deve ser usado quando não existem alternativas, por exemplo, ler objectos de um ficheiro que não controlamos ou filtrar uma lista de objectos que contém objectos misturados e que pretendemos separar.

Limitar Polimorfismo

Como o processo de *binding* dinâmico é um processo dispendioso, não faz sentido usá-lo em situações onde sabemos, exactamente, que tipo de dados vai ser usado. Assim, se temos uma classe que sabemos não tem subclasses, nem irá ter, podemos aplicar o modificador **final**. Este modificador permite indicar que uma classe não vai usar *binding* dinâmico para a determinação do seu tipo. Do mesmo modo, podemos aplicar o modificador a métodos, impedindo que os métodos sejam redefinidos e que seja necessário efectuar uma pesquisa para determinar qual o método correcto a invocar.

A utilização do modificador **final** permite ao compilador e interpretador a execução de algumas optimizações para tornar a execução da nossa aplicação mais rápida.

Composição

Composição pode não ser considerado por alguns autores como sendo uma das bases de POO, no entanto é importante que seja mencionada.

Composição não é mais que a criação de classes usando como base outras classes. Criamos assim uma relação entre objectos em que um determinado objecto contém no seu interior outros objectos que criamos.

De uma forma genérica poderemos pensar que estamos sempre a trabalhar com composição, afinal, todas as classes que criamos contém no seu interior alguma outra classe, mas o termo aplica-se mais quando estamos a falar de classes criadas por nós no domínio do problema que estamos a tentar resolver e não às classes que constituem a base da plataforma Java.

Composição aparece frequentemente associada a herança, surgindo em discussões de *Herança Vs. Composição*. A questão é que o mecanismo de herança, apesar de extremamente útil e poderoso, acarreta pequenos problemas, nomeadamente questões de performance, que embora não seja a única, é de longe a mais invocada nas comparações. Mas porquê a discussão se os dois mecanismos podem, aparentemente, coexistir sem problemas?

E na verdade não só podem como o fazem. Ao modelarmos um problema fazê-mo-lo quase sempre recorrendo a uma mistura entre composição e herança, criamos classes com tarefas especificas e conjugamos as mesmas de forma a construirmos classes de nível mais elevado e que fazem uso das classes de nível mais baixo.

A discussão surge, em casos mais extremos, por se considerar a herança um mecanismo prejudicial. Em casos mais ponderados, aqueles que todos devemos seguir, o uso de herança e composição deve ser visto à luz do problema que estamos a tentar resolver e escolher um ou outro em conformidade.

A composição oferece-nos uma forma de programarmos em POO evitando o uso de herança e, no caso do Java, evitando o problema de não existir herança múltipla.

Lembrando o exemplo das nossas bicicletas, em que criamos duas classes para representar dois tipos de bicicleta, e olhando para uma bicicleta real, podemos ver que facilmente se encontravam mais objectos: uma roda, travões, pedaleira, etc. Todos estes objectos fazem parte de uma bicicleta, portanto, podemos adicionar à nossa classe Bicicleta as classes Roda, Travao e Pedaleira. Com esta adição usámos o processo de composição: construímos uma classe a partir de outras classes, sem o uso de herança.

Em código o processo seria algo como [7]:

```
//Definir as partes que constituem a nossa bicicleta
public class Roda {

    public int raio;
    public float pressao;
}

public class Travao {

    public String tipo;

    public void travar() {
```

[7] Este código não está completo nem irá funcionar, é apenas uma aproximação.

```
    }
}

public class Pedaleira {

    public int nrVelocidades;
    public int tamanhoCorrente;

    public void pedalar() {
    }
}

//Criar a nossa bicicleta usado as partes definidas acima
public class Bicicleta {

    public String cor;

    //Substituindo o tamanhoRoda por duas rodas
    public Roda rodaFrente;
    public Roda rodaTras;

    //Substituindo o tipoTravao por dois travões
    public Travao travaoFrente;
    public Travao travaoTras;

    //Se temos travões, o método travar da bicicleta pede aos travões para actuarem
    public void travar() {
        travaoFrente.travar();
        travaoTras.travar();
    }

    //Se temos pedaleira, então pedimos à pedaleira para trabalhar
    public void pedalar() {
        pedaleira.pedalar();
    }
```

Capítulo 4. Características

Embora alguns destes pontos pudessem ser introduzidos nos capítulos anteriores, será mais simples separar os conceitos teóricos apresentados, das especificações técnicas de como o Java os implementa.

Neste capítulo são explicadas as particularidades de trabalhar com classes, objectos e outras características de programação orientada a objectos. Alguns destes pormenores foram já usados embora possam não ter sido explicados com a profundidade devida.

Classes

As classes permitem definir o comportamento e as várias características dos nossos objectos. Servem de molde para as instâncias que usamos e possibilitam a criação de código onde desenvolvemos o nosso programa.

É nas classes que escrevemos o nosso código, e cada ficheiro de código Java contém uma, e só uma, classe pública. É possível que existam mais classes definidas no mesmo ficheiro, mas estas classes não podem conter o modificador **public** e são consideradas *inner classes*. O ficheiro de código tem, obrigatoriamente, o nome da classe pública que está definida no seu interior.

Uma classe em Java, é composta por três partes: - Atributos - Construtores e Destrutores - Métodos que definem comportamento

Embora tenhamos dividido a classe em três partes, esta divisão é apenas lógica, não afecta em nada a escrita do nosso código, e a ordem das secções pode ser completamente alterada. É comum ver a declaração das variáveis, os atributos da classe, no fim do ficheiro.

Características do Uso de Classes

De acordo com as convenções da linguagem (não apresentadas), todas as classes devem ter nomes começados por maiúsculas e os seus atributos e métodos devem ter nomes começados por minúsculas.

Como dito anteriormente, cada classe deve ter o nome do ficheiro em que está guardada. Se tentarmos criar uma classe com o nome diferente do ficheiro onde a estamos a guardar, o compilador irá mostrar um erro quando compilarmos a classe. Muitos IDEs actuais mostram como erro antes de compilarmos mas na verdade o erro é sempre de compilação.

Uma classe não possui memória associada, não possui dados, ou outro tipo de informação que exista em memória; uma classe é apenas o código que escrevemos e que compilamos, dando assim origem a ficheiros com extensão *.class*. Estes ficheiros possuem no seu interior, o código Java compilado para *bytecode* e são os ficheiros que executamos quando usamos o nosso programa.

Quando falamos em classes e no facto delas não terem dados, temos de salvaguardar o facto de que existem atributos aos quais chamamos atributos de classe. No entanto, atributos de classe são atributos que existem a partir de uma instância de um objecto fundamental da plataforma Java: a instância de **Class**.

Em Java, quando executamos o nosso código, para cada classe que escrevemos, vai ser instanciado um objecto do tipo **Class** que contém todas as características que definimos no nosso código. É essa instância que nos permite à plataforma Java executar a nossa aplicação.

Atributos de Classe

Todos os atributos que definirmos sem o modificador **static** são atributos de instância, isto significa que só estarão disponíveis depois de instanciarmos o nosso objectoExemplos de instanciação poderão ser vistos na secção seguinte onde falamos sobre objectos.. Os atributos, e métodos, que forem definidos com o modificador **static** são chamados de atributos, ou métodos, de classe [1] e estão acessíveis sem necessidade de se instanciar um objecto.

Este tipo de atributos é o que se pode chamar de atributos globais footnote[O nome não está correcto, não existe qualquer noção de atributo ou variável global em Java, mas o comportamento é aproximado.], assim, enquanto que um atributo de instância pode conter valores diferentes para objectos diferentes, o conteúdo de um atributo de classe é único para todos os acessos ao atributo e quaisquer modificações são visíveis a todos os elementos que acederem ao atributo.

Para os leitores familiarizados com os problemas de variáveis globais, os atributos de classe oferecem, em grande parte, os mesmos problemas. Como qualquer modificação ao valor do atributo vai afectar todo o código que usa o atributo, as implicações de modificar um valor podem ser difíceis de prever.

Para acedermos a um atributo, ou método, de classe usamos simplesmente o nome da classe, um ponto, e o nome do atributo ou método que queremos invocar:

```
System.in = null;    //Atributo de classe
String.getClass();   //Método de classe
```

Hierarquia de Classes Imposta

A criação das nossas classes, em Java, é afectada por esta característica: uma hierarquia imposta, ou forçada. Em Java, todas as classes têm obrigatoriamente uma superclasse. Reforçando a ideia: **em Java, não há classe nenhuma que não tenha uma superclasse**.

Esta imposição é conseguida porque, em última análise, todas as classes serão sub-classes de Object. Esta é a classe especial que oferece alguns métodos úteis que podem ser redefinidos pelas sub-classes e que é usada como super-classe de todos os objectos, sem que seja necessário o programador especificar esta relação.

O compilador é responsável por garantir que todas as classes estendem da classe Object sem que seja necessário incluir no nosso código a relação explicita, e na maioria dos casos esta imposição não afecta o desenvolvimento das nossas aplicações. No entanto é necessário que esta relação esteja presente para o programador.

Classes Abstractas

Embora as nossas classes representem objectos do mundo real, há situações onde os conceitos do mundo real não podem ser representados por classes que depois possam ser instanciadas, não faz sentido termos objectos que representem conceitos do mundo real quando, no mundo real, esses conceitos não se traduzem em objectos com os quais possamos interagir.

Se pensarmos na representação, através de classes, de formas geométricas, o próprio conceito de "forma geométrica" não nos permite concretizar em objectos específicos. Claro que podemos concretizar um

[1] Devido ao uso do modificador **static** estes atributos e métodos são também chamados de atributos estáticos ou métodos estáticos.

quadrado, um cubo, uma esfera, uma linha, mas uma "forma geométrica" é um conceito que nos permite agrupar um conjunto de outros conceitos.

Assim, para classes que não fazem sentido ser instanciadas, podemos utilizar a palavra reservada **abstract** nas suas definições e essas classes passarão a ser consideradas classes abstractas pelo Java.

Uma classe abstracta: - Não pode ser instanciada. Se tentarmos criar uma instância da classe o compilador irá emitir um erro; - Pode conter métodos abstractos ou métodos com implementação; - Não pode ser a última classe da hierarquia se ainda existirem métodos abstractos por implementar. O compilador irá emitir erros caso existam métodos abstractos na hierarquia que não tenham sido implementados.

Construtores

Construtores são métodos especiais, com o mesmo nome que a classe onde estão, e que permitem criar uma instância da classe. São os métodos usados para criar toda a memória e para definir os valores iniciais dos atributos da classe.

Vimos numa secção anterior que o compilador força todas as nossas classes a serem subclasses da classe *Object*, mas o compilador impõe muitas outras alterações ao nosso código, algumas sem que tenhamos de lhe dar qualquer indicação.

Uma dessas outras situações afecta os construtores das classes e a forma como estes são invocados.

Uma classe não precisa declarar explicitamente um construtor, se não o fizer o compilador adiciona ao nosso código um construtor sem argumentos, no entanto, se o programador definir explicitamente um construtor para a classe, o compilador não só não acrescenta qualquer outro construtor como só reconhece os que estiverem explicitamente implementados. Além desta característica, todas as subclasses são obrigadas a invocar o construtor da superclasse nos seus construtores.

Mas, se o leitor já experimentou fazer uma classe em Java, e tendo em conta que todas as classes estendem de *Object*, poderá estar agora a pensar que nunca colocou qualquer invocação explicita ao construtor dessa classe. Afinal, se todas estendem de *Object* e somos obrigados a invocar o construtor da superclasse, porque é que, não tendo o leitor cumprido este último requisito o seu código compilou sem problemas?

A verdade é que, novamente, o compilador faz isso por nós, apenas e exclusivamente no caso da classe *Object*, em todas as restante situações, teremos de ser nós, programadores a cumprir com o requisito de invocação do construtor da superclasse.

Os nossos construtores são afectados da seguinte forma: - Se não existe um construtor na classe, o compilador adiciona um construtor sem argumentos. Se a classe não tiver superclasse explicita, então nada é alterado, por outro lado, se a classe tem uma superclasse explicita o compilar exige que exista uma chamada ao construtor da superclasse se esta tiver um construtor sem argumentos; - Se a superclasse da nossa classe não possuir construtor ou o construtor não tiver argumentos, o compilador aceita o nosso código e acrescenta automaticamente uma chamada ao construtor da superclasse; - Se a nossa classe definir um construtor explicitamente e não tiver uma superclasse explicita, o compilador nada faz; - Se a nossa classe definir um construtor e tiver uma superclasse explicita, o compilador exige que seja invocado o construtor da superclasse;

Esta característica é, por vezes, a fonte de muitas frustrações porque quem está a aprender não se lembra de invocar o construtor da superclasse, tipicamente isto resulta num erro de compilador que indica que o construtor que temos na nossa classe não existe na superclasse, o que pode causar alguma confusão.

Destrutor

Se os construtores permitem criar uma instância de uma classe, o destrutor permite remover essa instância, previamente criada, da memória e libertar quaisquer recursos que estejam associados à instância.

Em situações normais não é comum fazer-se uso de destrutores. A linguagem de programação Java tem mecanismos próprios para remover os objectos que já não são usados, de forma automática, e é nesses mecanismos que nos deveremos apoiar. Iremos ver o mecanismo de *Garbage Collector* em secções seguintes.

Para fazermos uso do destrutor da classe é necessário implementar o método **finalize**. Este método é usado pelo *Garbage Collector* imediatamente antes da instância ser removida e é possível colocar código que permita remover outros recursos ou que permita efectuar alguma operação necessária à correcta remoção da instância, por exemplo fechar algum ficheiro em uso.

No entanto, este é um método que não deve ser usado indiscriminadamente para libertar recursos, o programador deve ter o cuidado de estrutura o seu código de modo a não precisar de usar o método **finalize**.

Métodos Abstractos e Classes Finais

Na secção de classes abstractas indicamos que este tipo de classes não pode estar no fim da hierarquia. Esta situação está relacionada com métodos abstractos e com o facto de que uma classe abstracta não poder ser instanciada.

Consideremos uma hierarquia onde a primeira classe que fazemos é uma classe abstracta com o método **public abstract void teste();**. Enquanto o método não for implementado por alguma subclasse, todas as subclasses na hierarquia serão, obrigatoriamente, abstractas. Se até ao fim da hierarquia não existir uma classe que implemente este método e se todas as classes forem abstractas, então não é possível ao compilador aceitar a nossa hierarquia.

Se o compilador nos deixasse compilar o código então poderíamos correr o risco de tentarmos usar um método que, para todos os efeitos, não está implementado por classe alguma, colocando assim o programador numa situação impossível: por um lado ter uma declaração do método por outro não ter a sua implementação. Por essa razão, as classes que se situam no fim da hierarquia têm de ser classes não abstractas, o que significa que ou implementam todos os métodos abstractos que ainda possam existir ou já se encontram num lugar da hierarquia onde não há mais métodos abstractos a implementar.

Quando falamos em classes do fim da hierarquia surgem classes especiais designadas por **classes finais**.

Classes finais, definidas através do uso da palavra reservada **final** são classes que não podem ser usadas como superclasses de outras classes. Podemos considerar as classes finais como as folhas de um ramo, situadas no fim da hierarquia não admitem qualquer subclasse.

Uma classe final: - Não aceita métodos abstractos; - Não permite subclasses, efectivamente impedindo qualquer herança a partir do ponto onde são colocadas; - Permite melhorar a performance de aplicações

dado que o compilador tem a garantia que a classe não contém subclasses e pode efectuar várias opções de optimização, especialmente no que toca a polimorfismo, que de outro modo não poderia fazer.

Classes Aninhadas

O Java permite que sejam definidas classes dentro de outras classes, às quais damos o nome de classes aninhadas [2], e que oferecem a possibilidade de agrupar de forma lógica classes. Esta forma de agrupar as classes não deve substituir o uso de *packages* mas apenas ser utilizada em situações específicas em que os *packages* não fazem sentido.

Tipicamente devem ser usadas quando precisamos de uma classe que não queremos expor para fora ou que, estando exposta, apenas faça sentido no âmbito da classe mãe em que está definida, por exemplo, ao definirmos uma lista ligada, podemos criar os nós da lista como classes aninhadas. Estes nós, úteis ao funcionamento interno da nossa lista, de nada servem para a utilização da lista enquanto estrutura de dados.

As classes aninhadas podem ser de 2 tipos: estáticas e não estáticas. Em que as estáticas são referidas como classes estáticas aninhadas [3] e as não estáticas como classes internas [4].

As vantagens de classes aninhadas são: - Agrupamento lógico de classes se as mesmas são apenas úties para uma outra classe, a classe que as contém; - Aumento do encapsulamento. As classes aninhadas podem ser escondidas do exterior e ajudar no funcionado da classe que as detém de forma privada; - Código mais simples de ler e de manter já que as classes de ajuda estão imediatamente acessíveis dentro das classes que estas ajudam.

Exemplo de sintaxe:

```
class ClasseExterior {
    //...
    class ClasseInterna {
        //...
    }
}
```

```
class ClasseExterior {
    //...
    static class ClasseEstaticaInterna {
        //...
    }

    class ClasseInterna {
        //...
    }
}
```

Classes Aninhadas: Classes Anónimas

Classes anónimas são classes definidas de forma especial, muito úteis quando se trabalha com interfaces gráficas, e que permitem fazer código que noutras linguagens é feito com funções anónimas.

De forma simples, uma classe anónima é uma classe que, no código, não tem nome, e é definida dentro da invocação de outros métodos. Este tipo de classes deve ser pequena, e ser usada apenas quando não faz

[2] tradução de nested classes.
[3] static nested classes.
[4] inner classes.

sentido implementar a classe com interna ou estática interna ou num ficheiro próprio. Devem também ser usadas, apenas quando são subclasses de outras classes ou implementações de interfaces.

Quando o nosso código é compilado, estas classes anónimas são, na verdade, extraídas para um ficheiro próprio, com o nome da classe onde foram criadas seguido de **$<numero da classe>**. Se existirem várias classes internas, anónimas ou não, o número é atribuído de forma sequencial dependente da ordem com que aparecem no código fonte.

Exemplos:

```
new Thread(new Runnable() {

    public void run() {
        while(true) {
            System.out.println("Thread a executar infinitamente...");
        }
    }
}).start();
```

Se decompusermos o código acima, vamos ver que estamos a usar a classe *Thread* [5], esta classe tem um método, **run()**, onde temos de implementar o código que a *thread* vai executar. Tem também um método, **start()** que indica que a *thread* deve começar a execução.

Uma forma comum de se implementar uma *thread* é estender a classe *Thread*, e na sub-classe redefinir o método **run()** como pretendermos. Mas se pretendermos apenas implementar o método **run()** uma outra solução é usar o construtor especial da classe *Thread* que aceita um objecto que implemente a interface *Runnable*, esta interface define apenas um método, o **run()**, que é depois usado pela *thread* para executar.

Assim, o que fazemos é criar um objecto que implemente *Runnable*, implementar o método **run()** desse objecto, criar um objecto do tipo *Thread* e passar par ao construtor o objecto do tipo *Runnable* anterior. Depois disto ao iniciarmos a thread com o método **start()**, o nosso código no método **run()** do objecto *Runnable* vai ser executado.

Embora na descrição possa parecer que estamos a ter mais trabalho, lembrem-se que pretendemos usar classes anónimas.

```
//numa execução normal, este código iria iniciar uma thread.
//Atenção que este código não tem qualquer efeito como está.
new Thread().start();
```

```
//Podemos usar o construtor que recebe um Runnable para
//passarmos o código que queremos executar.
//Ser tentarem executar o código seguinte vão receber um bom
//erro de compilação :), reparem que Runnable é uma interface, não
//pode ser instanciada.
new Thread(new Runnable()).start();
```

```
//Reparem nas chavetas depois da classe Runnable... o que estamos a
//dizer é: new Thread(new <sem nome> implements Runnable { ... }).start();
//mas estamos a omitir várias palavras reservadas, nomeadamente "implements"
//e o nome da classe que estamos a construir
new Thread(new Runnable() {

    public void run() {
        while(true) {
            System.out.println("Thread a executar infinitamente...");
        }
```

[5] O conceito de *Threads* não será abordado neste manual.

```
    }
}).start();
```

Esta técnica funciona para classes que estendam outras classes ou que implementem interfaces e seguem sempre o mesmo formato: o nome da classe e a palavra **extends**/**implements** são omitidos, apenas a superclasse o interface é mantida. De seguida são colocados os parêntesis e as chavetas, dentro das quais implementamos o corpo da classe anónima.

Notas finais: - Classes anónimas não são reutilizáveis e são conhecidas apenas dentro do método onde foram criadas. - Se quisermos passar parâmetros para métodos de classes anónimas que estejam fora da classe, por exemplo no método onde a classe é criada ou na classe principal do ficheiro, as variáveis terão de ser atributos de instância da classe principal ou serem métodos definidos como finais.

Manipulação de Objectos

Objectos são os elementos com que qualquer programador de Java tem de lidar, são os elementos com que são construídos os vários programas, são os elementos que representam e guardam os nossos dados, são os elementos através dos quais os nossos programas levam a cabo as tarefas para as quais foram desenhados.

Tirando os tipo primitivos, todos os nosso objectos são acedidos através de referências para as quais as nossas variáveis apontam e é importante perceber a diferença entre uma variável e uma referência: uma variável é o que usamos para guardar as referências, uma referência é um ponteiro que indica em que zona de memória estão os nossos dados.

Embora seja importante conhecer a diferença, na utilização diária de linguagem não pensamos nesta distinção, e as nossas variáveis ou referências acabam por se misturar.

```
//declarar uma variável, neste momento a
//variável não contém uma referência válida
String umaVar;

//colocar na variável uma referência válida
umaVar = new String("Esta variável está iniciada");
```

O exemplo anterior introduz a palavra reservada mais usada em conjunto com os objectos: **new**, é a palavra reservada que permite instanciar um objecto, efectivamente criando espaço em memória para o novo objecto e devolvendo uma referência para esse espaço, é por essa razão que para criarmos objectos fazemos: **new Objecto();** e atribuímos o resultado desta operação a uma variável.

Em Java as variáveis apenas podem conter referências para objectos do tipo com que foram declaradas, podem trocar de referência em qualquer altura do seu ciclo de vida, mas todas as referências que se tentem colocar numa variável têm de ser do mesmo tipo de dados que foi usado para declarar a variável [6]. Se necessário reveja o conceito de polimorfismo, importante neste caso.

Iniciação

Para podermos utilizar um objecto é necessário termos uma referência para ele através de uma variável. As variáveis antes de serem usadas precisam ser iniciadas, processo em que se define o valor da variável. Esta definição pode ser feita de várias maneiras:

[6] Em oposição a linguagens onde as variáveis não ficam presas a um tipo de dados, e que permitem a utilização da mesma variável para tipos diferentes

- Por omissão, todos os tipos de dados possuem um valor por omissão, no caso de objectos é **null**, no caso de inteiros é zero, etc. Este tipo de iniciação ocorre sempre que declararmos uma variável sem indicarmos o seu valor de forma explicita, mas apenas para os atributos de instância, isto é, as variáveis usadas pela classe. No caso das variáveis usadas em métodos, o compilador irá emitir erros sempre que não seja feita a iniciação explicita de variáveis que estejam a ser usadas.

- Por definição, este tipo de iniciação é usado quando, ao declararmos a variável, definimos o seu valor na mesma linha, exemplo: **int x = 5;**.

- Por construtor, as variáveis podem ser iniciadas através do construtor da classe.

Nota: O interpretador irá iniciar as variáveis pela ordem definida acima.

```
public class MinhaClasse {

    //iniciação por definição
    private int id = 1;
    //iniciação por omissão, eu = null
    private MinhaClasse eu;
    //iniciação no construtor...
    private String nome;

    public MinhaClasse(String nome) {
        this.nome = nome;
    }
}
```

Instanciação

Para instanciar um objecto, como já vimos, precisamos usar o operador **new**, e em conjunto com o operador o construtor que desejarmos ou que estiver disponível para a classe.

O construtor é um método especial no sentido em que não pode ser invocado directamente, a única forma de podermos usar o construtor é se for em conjunto com o operador **new**, todas as outras tentativas são ilegais e originam erros de compilação.

```
public class MinhaClasse {

    private int id;
    private String nome;

    public MinhaClasse(int id, String nome) {
        this.id = id;
        this.nome = nome;
    }
}

//....

//utilização válida
MinhaClasse mc = new MinhaClasse(5, "Uma classe");

//utilização inválida, tentativa de aceder directamente ao construtor
mc.MinhaClasse(5, "má tentativa");
```

Nota: Como já foi referindo, o processo de instanciação apenas é importante no caso de objectos, e não existe em tipos de dados primitivos.

Manipulação de Objectos: Métodos e Atributos

O acesso a um método ou a um atributo [7] de um objecto é feito através do operador ponto, **.**. Este operador é colocado a seguir a uma variável e depois dele vem o nome do método a invocar ou do atributo a aceder.

```
variavel.metodo(parametros);
variavel.atributo = 3
variavel2 = variavel.atributo;
```

Os métodos e atributos que estão disponíveis para invocação dependem da visibilidade e do local onde é feita a invocação.

Na passagem de parâmetros para métodos, são passadas sempre as referências, e nunca são passadas cópias dos objectos, dessa forma, qualquer modificação que um objecto sofra dentro do método estará visível fora do método.

```
public class MClass {

    public String nome;

    //método que recebe um objecto e altera o nome
    public void alterarNome(MClass obj) {
        obj.nome = "teste";
    }
}

//usar construtor por omissão
Mclass var = new MClass();

System.out.println("Nome: "+ var.nome);
var.alterarNome(var);
System.out.println("Nome: " + var.nome);
```

A execução das parcelas de código escritas acima iriam levar a: Nome: Nome: teste

Atributos da Classe e Atributos de Instância

Já vimos dois tipos de atributos ou variáveis, os que são definidos no topo da classe a que chamamos atributos de instância [8], os que são declarados dentro dos métodos, que são apenas variáveis locais de cada método e desaparecem quando o método termina, mas temos também atributos que se referem à classe e estão acessíveis sem que seja necessário instanciar um objecto para os usar: os atributos de classe.

Os atributos de instância apenas existem se instanciarmos a classe onde estão definidos, não existe outra forma de lhes aceder, precisamos sempre de ter uma instância e usar o operador ponto.

Os atributos de classe, em contrapartida, existem na definição da classe e não precisam de uma instância. São acedidos com o mesmo operador ponto mas não necessitam de uma instância. Podemos ver um uso desse tipo de atributos sempre que mostramos dados na consola através de **System.out.println()**. Efectivamente o atributo **out** é um atributo de classe que invocamos usando o nome da classe, seguido do operador de acesso.

Estes atributos de classe são criados usando o modificador **static** e também são afectados pelos modificadores de acesso.

[7] Atributo ou variável.
[8] Existem convenções de código que colocam os atributos de classe no fundo da classe em vez de no topo.

Um atributo de classe é instanciado quando acedemos pela primeira vez à classe e o seu valor é comum para todos os acessos à classe, isto é, se aceder e alterar o valor a alteração é visível em todos os blocos de código que usem esse atributo [9].

```
public class MClass {
    public static String nome = "Eu sou de classe!";
}

//aceder ao atributo sem instanciar e modificar o seu valor
MClass.nome = "Eu também.";
```

Referência Especial this

Em alguns exemplos pode ser visto o uso de uma palavra especial, **this**, para a invocação de métodos ou acesso a variáveis. Esta palavra é uma referência especial, uma variável sempre presente em todos os nossos objectos, que corresponde ao objecto em que estamos.

Embora no código usemos a referência especial sem lhe definirmos um valor, sempre que instanciamos um novo objecto, esse objecto ganha uma variável especial que tem uma referência para ele próprio e que é acedida através da palavra reservada **this**.

Esta referência está disponível apenas dentro dos métodos de instância e dentro do construtor e permite-nos indicar explicitamente a que objecto queremos aceder quando o nome do método ou do atributo é dúbio, por exmplo, se existe um atributo de instância chamado **nome** e se estamos num método que recebe um parâmetro **nome** como é que podemos distinguir entre o atributo de instância e a variável local do método?

```
public void setNome(String nome) {
    //usar o this para indicar que queremos atribuir ao atributo de instância
    //o valor que vem no parâmetro de entrada do método
    this.nome = nome;
}
```

No exemplo anterior, durante a execução do programa, a referência especial **this** estará a apontar para o objecto que estiver a ser usado no momento.

Operador instanceof

O Java permite determinar o tipo de um objecto através de um operador especial, **instanceof**, que nos devolve verdadeiro ou falso consoante o objecto que estamos a comparar é ou não do tipo especificado.

É importante referir que o uso deste operador tem um impacto muito grande na performance de uma aplicação, que o seu resultado deve ser visto à luz do conceito de herança e que as situações onde a utilização deste operador podem ser aceites são limitadas a um ou dois casos. Todas as outras utilizações revelam erros na definição da hierarquia e consequentemente um má resolução do problema.

A sintaxe de utilização é simples: variavel instanceof Object

O operador devolve verdadeiro caso a variável seja do tipo *Object* e falso caso contrário. E este é um exemplo do que pode correr mal na utilização deste operador: consideremos a classe *MClass* sem superclasse específica, isto é, foi declarada sem ter qualquer super-classe, neste caso sabemos que por omissão todas as classes são subclasses de *Object*. Assim, ao fazermos:

```
MClass m = new MClass();
```

[9] Atributos de classe são similares a variáveis globais em outras linguagens e, quando editáveis, sofrem do mesmo problema.

```
if(m instanceof Object) {
    System.out.println("É um Object");
} else {
    System.out.println("Não é um Object");
}
```

O resultado será sempre verdadeiro, dado que todas as classes são subclasses de *Object*. Na verdade, o operador **instanceof** vai devolver **true** se testarmos uma variável com todas as classes na hierarquia, começando pela classe com que foi declarada a variável e subindo uma superclasse de cada vez.

E qual será o problema? Se nos for dado um objecto, não conseguimos saber exactamente qual o tipo com que foi declarado sem testarmos todas as classes possíveis.

A utilização deste operador é útil em dois casos genéricos: quando estamos a ler objectos que foram guardados num ficheiro, e precisamos de os instanciar, ou quando estamos a filtrar objectos que estão numa estrutura de dados com vários objectos diferentes. Fora estas duas situações, a utilização do **instanceof** deverá ser bastante ponderada de modo a confirmar que não existe outra alternativa.

Operador de Comparação ==

Uma das primeiras dúvidas que costumam aparecer sobre objectos é a comparação. É comum um programador que está a iniciar no desenvolvimento de Java pensar que pode comparar dois objectos do mesmo modo que compara tipos primitivos, mas em Java, o operador de comparação == não compara objectos.

Este comparador compara referências, e apenas referências. A comparação de objectos é feita com o método **equals** que todas as classes devem re-implementar.

Dado que o operador de comparação apenas compara referências, dois objectos são iguais à luz deste comparador, apenas se forem o mesmo objecto, isto é, o operador devolverá verdadeiro apenas para variáveis que contenham referências para o mesmo objecto e nunca compara os valores internos dos objectos.

Garbage Collector e Memória

O *Garbage Collector* é um mecanismo automático de gestão de memória fornecido pela plataforma Java e que permite remover instâncias que já não estão em uso. Tipicamente, para um programador de C/ C\++ ou outra linguagem de programação sem gestão automática de memória, uma variável deixa de existir quando a função ou procedimento onde foi declarada terminar a sua execução ou, se for uma variável com memória declarada manualmente, quando o programador remover a alocação de memória. Ora se o programador se esquecer de remover a memória, ou se a função a executar só terminar quando o programa terminar, todas as variáveis se manterão em memória mesmo que não sejam precisas ou não sejam mais usadas.

Interfaces

Uma interface permite definir um contrato para determinados objectos, isto é, uma interface permite que o programador indique um conjunto de métodos que terão de ser respeitados por todas as classes que implementem a interface.

O mecanismo de interfaces encontra-se à parte do sistema normal de herança e de classes do Java, embora exista herança entre interfaces, devido às particularidades das interfaces, não funciona exactamente do

mesmo modo. Além disso, as interfaces permite contornar, até certo ponto, a não existência de herança múltipla.

No fundo, uma interface é apenas um conjunto de métodos agrupados logicamente, que garantem ao programador que todas as classes que implementem a interface irão ter esses métodos definidos.

Interfaces em Java

Uma interface apenas pode conter definições de métodos, atributos estáticos e tipos internos [10]. Por outras palavras, não podem existir atributos de instância, não pode existir código dentro dos métodos, não pode existir nada na interface para o qual seja necessário instanciar a interface. As interfaces não podem ser instanciadas.

Devido à natureza das interfaces, não podem existir elementos privados dentro da interface, isto implica que todos os métodos e atributos estáticos são públicos e a herança entre interfaces, embora possível, apenas passa as definições de métodos.

Uma classe pode implementar várias interfaces, tendo para isso que implementar todos os métodos que as interfaces definem. Se a classe não implementar todos os métodos terá de ser declarada como abstracta e os métodos deverão ser implementados pela primeira classe não abstracta que surja ao descrer na hierarquia.

As interfaces surgem por vezes com o papel de marcadores. Estes marcadores são interfaces sem qualquer código, seja referente a definição de métodos ou atributos estáticos, e que são usados para identificar se uma determinada classe é de um determinado tipo. Por exemplo, a interface *Serializable*, não possui atributos estáticos, declarações de métodos ou tipos internos, é uma interface vazia, mas é usada pelo sistema de serialização. Todos os objectos que queriam ser serializados devem implementar esta interface, apesar dela não obrigar a implementar método algum [11].

Exemplo em Código

```
public interface ComandoUniversal {

    //Não é colocado modificador de acesso, todos os métodos são públicos
    void ligar();
    void desligar();
    void mudarCanal(int canal);
    void pause();
    void start();
    void stop();
    void semBaterias();
}

//Herança entre interfaces: todos os métodos definidos na interface ComandoUniversal são herdados
public interface ComandoUniversal2 extends ComandoUnivesal {

    int mostrarCanal();
}

public class ComandoTelevisao implements ComandoUniversal {

    //aqui já estão definidos os modificadores de acesso
    public void ligar() {
```

[10] Tipos internos são classes definidas dentro de classes, ou interfaces neste caso.
[11] O sistema de serialização poderá obrigar a implementar dois métodos especiais, mas esses métodos não estão definidos na interface *Serializable*

```
    }

    public void desligar() {
    }

    public void mudarCanal(int canal) {
    }

    public void pause() {
    }

    public void start() {
    }

    public void stop() {
    }

    public void semBaterias() {
    }

}

public class ComandoBox implements ComandoUniversal2 {

    //métodos herdados pela interface ComandoUniversal2
    public void ligar() {
    }

    public void desligar() {
    }

    void mudarCanal(int canal) {
    }

    void pause() {
    }

    public void start() {
    }

    public void stop() {
    }

    public void semBaterias() {
    }

    //método extra definido na interface ComandoUniversal2
    public int mostrarCanal() {
    }
}

//Esta classe, sem a implementação dos métodos, provocará um erro de compilação.
public class ComandoErrado implements ComandoUniversal {
    //sem qualquer implementação
}

//Nesta classe faltam dois métodos, logo terá de ser uma classe abstracta e os dois métodos em
//falta terão de ser implementados numa subclasse
public class ComandoAbstracto implements ComandoUniversal {

    public void ligar() {
    }

    public void desligar() {
    }
```

```
    public void mudarCanal(int canal) {
    }

    public void pause() {
    }

    public void start() {
    }
}

//Esta subclasse implementa os dois métodos em falta na classe mãe.
//Devido ao mecanismo de herança, esta classe também implementa a interface ComandoUniversal
public class Comando extends ComandoAbstacto {

    public void stop() {
    }

    public void semBaterias() {
    }
}
```

Conversões (Cast)

A conversão de tipos de dados permite que o conteúdo de uma variável possa ser transformado noutro tipo de forma legal. Esta conversão, quando explicita, é feita colocando o tipo de dados para o qual queremos converter entre parêntesis antes do valor a converter:

```
double valorAConverter = 3455.22;
int valorConvertido = (int) valorAConverter;
```

Para os tipos primitivos as conversões não oferecem problemas de maior, a única situação anómala que resulta de uma conversão de tipos primitivos é a possível perda de precisão. Por exemplo, se convertermos o **float** 3.14 para inteiro, **int pi = (int)3.14**, acabamos com o valor 3, já que os tipos inteiros não possuem casas decimais e estas são simplesmente descartadas.

Quando falamos de objectos, outros problemas podem surgir. Em conversões entre tipos de objectos existem duas situações possíveis: - Upcast - Downcast

O **upcast** acontece sempre que um objecto é convertido para um dos seus super tipos, por exemplo, podemos converter qualquer objecto para a classe **Object** já que esta está no topo da hierarquia. Do mesmo modo, se possuirmos a hierarquia *SerVivo* > *Mamifero* > *Humano*, podemos converter uma variável do tipo **Humano** para **Mamifero** ou para **SerVivo** sem quaisquer problemas.

Este tipo de conversão é automático (não precisamos de dizer explicitamente que queremos uma conversão) e sempre seguro e deriva da aplicação directa do conceito de polimorfismo.

O **downcast** é uma conversão que acontece quando convertemos um tipo de dados num subtipo, por exemplo, continuando com a hierarquia apresentada anteriormente, se quisermos converter um **Mamifero** para **Humano** estamos perante um **downcast**.

Esta operação tem de ser feita explicitamente pelo programador, através da utilização dos parêntesis, e tem de ser o programador a garantir que a operação é válida. Se a conversão não for possível o interpretador irá lançar uma excepção que, se não for tratada, termina a nossa aplicação. Sempre que um programador faz um **downcast** o interpretador verifica a conversão.

Modificadores de Acesso

Os modificadores de acesso permitem controlar o acesso à classe, suas variáveis e métodos. Existem em quatro tipos: **acesso privado, acesso protegido, acesso publico e acesso de package**.

Estes modificadores controlam que outras classes podem aceder às classes, atributos e métodos que estão a afectar. A lista de modificadores, do mais restritivo para o mais permissivo é:

privado	O acesso privado implica que apenas a classe dona dos atributos e dos métodos os consegue usar. Este modificador não faz sentido ser aplicado a uma classe, sendo usado apenas em métodos e em atributos. É usado através da palavra **private**.
package	O acesso de *package* permite que apenas a própria classe e as classes no mesmo *package* possam usar a classe, os métodos ou atributos. Este acesso é também designado de acesso por omissão dado que é definido apenas quando não se usa qualquer palavra reservada.
protegido	Pode ser usado em classes, métodos e atributos e permite o acesso aos elementos pela própria classe, suas subclasses e classes que estejam no mesmo *package*. É usado através da palavra **protected**.
público	O acesso público é o mais permissivo, no sentido em que todas as classes podem aceder aos elementos que estão definidos com públicos, sem qualquer restrição. Para este modificador é preciso usar a palavra **public**.

Redefinição de Métodos

A programação orientada a objectos introduz o conceito de redefinição de métodos como **a possibilidade das subclasses alterarem o comportamento dos métodos das superclasses fornecendo a sua própria implementação**.

Este conceito está relacionado com herança e polimorfismo, que vimos anteriormente, e de forma simples significa que as subclasses podem fornecer um comportamento diferente para um método que tenha sido definido numa das suas superclasses, se a este tiverem acesso. Desta forma é possível que uma subclasse acrescente comportamento a um método existente ou que o altere por completo.

Capítulo 5. Tabelas (Arrays, Vectores ou Matrizes)

Antes de iniciarmos o tema, vamos tentar definir um termo a usar que possa ser uniforme em todo o tutorial. *Array* pode ser traduzido para vector, sendo esta uma tradução comum, ou para matriz quando falamos em duas dimensões (linhas e colunas) ou, neste caso, para tabelas. A tradução ou termo a usar não é consensual, e piora com o tema que pretendemos introduzir.

Desta forma, iremos usar o termo original, *array*, quando nos referirmos a um *array* de dimensão 1 [1], e *array de arrays* quando nos referirmos a um *array* de dimensão maior que 1, por exemplo, uma matriz.

A razão pela qual não iremos usar o termo matriz, é porque em Java, um *array* multidimensional, ou neste caso de duas dimensões, não tem de ter todas as linhas do mesmo tamanho ao contrário de outras linguagens. Por exemplo, em C, ao declarar a variável **int m[2][3]**, estaremos a declarar um *array* com duas linhas e cada linha tem 3 colunas, C não nos permite fazer, **int m[2][]** dado que na declaração, o tamanho total do *array* tem de ser conhecido. Na verdade é possível ter *arrays* com linhas de tamanhos variáveis em C. O que se pretende salientar é a diferença na forma como são obtidos dado que para fazer o mesmo que em Java, teremos de usar memória dinâmica e a declaração das variáveis torna-se diferente, bem como o modo de acesso.

Portanto, a nível de declaração de *arrays* de tamanhos variáveis, C e outras linguagens, fazem uso de funções/mecanismos extra, enquanto que em Java os *arrays* são tratados como objectos e isso reflecte-se na forma como são declarados e usados.

Arrays como Objectos

Os *arrays* de Java são objectos e tal como uma String, ou qualquer outro objecto que seja por nós criado, os *arrays* em Java precisam ser instanciados recorrendo à palavra reservada **new**. E porque são objectos, podem ser facilmente passadas como parâmetros de métodos, ser feita a atribuição directa entre dois *arrays* ou serem devolvidas por métodos. No fundo, o que é passado, como em qualquer outro objecto, é uma referência e não uma cópia dos elementos.

A declaração de *arrays* é similar ao que é usado noutras linguagens:

```
//Para uma dimensão
TipoDeDados[] nomeDaVariavel;
//ou
TipoDeDados nomeDaVariavel[];

//Para várias dimensões
TipoDeDados[][] nomeDaVariavel;
//ou
TipoDeDados nomeDaVariavel[][];

TipoDeDados[][][] nomeDaVariavel;
//ou
TipoDeDados nomeDaVariavel[][][];
```

Mas, contrariamente a outras linguagens, ao declararmos o *array* como indicado acima não conseguimos usá-lo para guardar dados sem antes reservarmos memória para os valores através de instanciação:

[1] Não confundir com um *array* de uma posição, um *array* de dimensão 1 tem apenas uma linha.

```
TipoDeDados[] nomeDaVariavel;

//Instanciar o array para 5 posições
nomeDaVariavel = new TipoDeDados[5];

//Instanciar e atribuir valor no mesmo passo
nomeDaVariavel = new TipoDeDados{valor1, valor2, valor3, ...};

//Declaração com instanciação e atribução, array de array
String[][] nomes = new String{{"maria", "joao", "paulo"}, {"andré", "carlos", "costa"}, ...}
```

Este processo de instanciação apenas nos oferece metade da solução. Já vimos as diferenças entre dados primitivos e objectos, sabemos que os dados primitivos não precisam ser instanciados e que os objectos precisam ser instanciados para que possamos usar as variáveis que para eles referenciam.

No caso de *arrays*, se o *array* é composto apenas por dados primitivos, a instanciação do *array* resulta numa variável que podemos usar imediatamente, mas se estivermos a trabalhar com *arrays* de objectos, então a instanciação do *array* apenas nos dá um *array* com todas as posições a **null**. Isto significa que, antes de podermos aceder a uma posição do *array*, precisamos instanciar o objecto que vai ficar guardado nessa posição.

Esta questão é particularmente importante quando pensamos em *arrays* de *arrays* dado que nos permite fazer instanciações parciais, por exemplo, numa matriz de 5x5 podemos instanciar apenas o primeiro *array*, correspondente às linhas, e mais tarde o segundo, correspondente a 5 colunas. Se trabalharmos com *arrays* que contenham *arrays* variáveis, em que cada linha tem diferentes números de colunas, esta característica é muito importante.

Utilitários

Para a manipulação de *arrays*, o Java oferece-nos a classe **Arrays** e o método **System.arraycopy**.

Com a classe temos à nossa disposição métodos de ordenação e de pesquisa que podemos usar para trabalhar os nossos *arrays* de forma facilitada. Com o **arraycopy** podemos fazer cópias rápidas dos valores dos nossos *arrays*, no entanto é necessário salvaguardar que o método **arraycopy** apenas efectua uma **shallow copy**, copiando apenas as referências. O resultado desta cópia é que ficamos com dois *arrays*, a cópia e o original, a apontar para os mesmos objectos.

Capítulo 6. Strings

Strings são os tipos de dados usados quando queremos manipular texto. Em Java, as strings são objectos e não são vectores, ao contrario de outras linguagens onde as strings são cadeias de caracteres, tipicamente terminadas com um caractere especial, em Java as strings são objectos, com tratamento especial em determinadas situações, mas objectos mesmo assim, criados a partir da classe String.

Em Java as strings são imutáveis, não é possível alterar o valor de uma string depois desta ter sido criada. Este é um ponto que deve ser reforçado, ao criarmos uma string o seu valor nunca poderá ser alterado, seja por que operação for, a JVM nunca irá modificar os dados guardados na string. Ao somarmos duas strings, estamos na verdade a criar uma terceira string, nova, que irá conter o resultado da soma. Ao substituirmos um caractere dentro de uma string, estamos a criar uma segunda string, que vai ter todos os caracteres menos o(s) que retiramos.

Isto pode parecer algo estranho, mas é a forma como Java lida com strings, e esta forma de lidar com strings implica que "editar" strings se torna um processo computacionalmente exigente, não sendo as strings, os objectos correctos quando precisamos de editar texto. No entanto, as strings têm a sua utilidade, são talvez dos tipos de objectos mais usados em qualquer programa.

Embora não sejam vectores, as strings possuem vários métodos que permite aceder aos dados através da posição, onde a posição do primeiro caractere é a posição zero e a do último a posição **n-1**, sendo **n** o tamanho da string. Para quem já tenha usado C ou C++, não existe caractere terminador **\0**.

Casos Especiais

O compilador trata algumas situações como especiais no que toca ao tratamento de strings, por exemplo, a criação de uma string pode ser feita sem o operador **new**

Instanciar uma String:

```
String s = new String("Isto é uma String");
String s2 = "Isto é uma segunda String"; //processo especial, omissão do construtor e operador new
```

Concatenar uma String:

```
String s3 = "Parte 1" + ", outra parte"; //criação de duas strings,
                                         //soma dos seus valores e
                                         //conseguente criação de uma terceira string, s3
```

Alguns dos métodos de uso comum da classe String:

```
String s = "Uma string de teste";

s.indexOf("U");         //Devolve zero
s.charAt(4);            //Devolve 's', o 5º caractere
s.equals(in);
s.compareTo(in);
s.compareToIgnoreCase("");
s.length();
s.concat(s)
s.contains(s);
s.format(s, args);      //Formata uma string. Cria uma string nova formatada segundo os parâmetros
s.matches("");          //Permite pesquisas com expressões regulares.
s.trim();               //Removes espaços brancos no início e fim da string.
s.toCharArray();
```

```
s.substring("");        //Obtém uma nova string que é uma sub-string da string usada.
s.split("");            //Divide uma string. Devolve um array com novas strings que representam os
s.toUpperCase();        //Devolve uma nova string com os caracteres todos em maiúscula.
s.toLowerCase();        //Devolve uma nova string com os caracteres todos em minúscula.
s.replace('s', '0');    //Substituição de um caractere por outro.
s.replaceAll('s', '0'); //Substituir todas as ocorrências de um caractere.
```

Existem muitos mais métodos na classe que nos são úteis.

No que toca a processamento de strings, as classes *StringBuilder* e *StringBuffer* permitem maior performance e são as classes que devem ser usadas sempre que seja necessário "editar" muitas strings.

Capítulo 7. Packages

Packages são usados para agrupar unidades de código relacionadas. Contêm classes Java organizadas de forma a identificar as relações que existem nas suas funcionalidades.

Ao contrário de outras linguagens footnome:[Embora o termo seja apenas de Java, outras linguagens possuem conceitos idênticos.], em Java os **packages** representam estruturas de directórios que necessitam existir em disco. Um **package** representa sempre um conjunto de pastas que contém os ficheiro de código fonte Java, e posteriormente, os ficheiros compilados.

Essa característica impões algumas restrições à forma como as classes podem ser organizadas. Duas classes precisam estar dentro da mesma pasta no disco para poderem estar no mesmo **package**.

O **package** a que uma classe pertence definem também o nome completo da classe. Desta forma, o nome completo de uma classe é o nome de todas as pastas, desde a raiz footnote_[Considera-se raiz a pasta base onde se coloca o código fonte, tipicamente denominada **src**]. Por exemplo, a classe *String* tem como nome completo o nome **java.lang.String**.

Para indicar a que package pertence, uma classe usa a primeira linha de código no seu ficheiro de código fonte com a estrutura: package java.lang;

Como podem ver, as pastas que constituem um **package** são especificadas separando cada uma por um ponto. Se olharmos para a estrutura no disco do código fonte da plataforma Java, podemos ver que dentro da pasta **src** existe um pasta chamada **java**, que contém, entre outras, a pasta **lang**, e é dentro dessa pasta que encontramos a classe *String*.

Em Java, uma classe pertence sempre a um **package**, se não for especificado nenhuma instrução para identificar o **package** e a classe está dentro de alguma pasta que seja sub-pasta de **src**, então o compilador apresentará um erro e essa classe não compilará. Se não for indicado o **package**, e a classe estiver directamente dentro da pasta **src**, então essa classe pertence ao **package** *default*.

O sistema de **packages** permite agrupar classes com comportamento similar, mas também identificar de forma inequívoca uma classe. Isto porque se nos lembrarmos de uma das características da plataforma Java, o Java é distribuído, permite carregar uma classe a partir da rede e nesse cenário é bastante fácil dois programadores criarem classes diferentes com o mesmo nome, e alguém as tentar usar no mesmo sistema. Se forem criados **packages** únicos, então essas classes nunca irão entrar em conflito já que o seu nome completo é diferente.

Por este motivo é desaconselhado que sejam criadas classes no **package** default.

Mas se é fácil alguém criar classes com o mesmo nome, não será igualmente fácil dois programadores escolherem a mesma estrutura de directórios e assim, o mesmo **package**? Para evitar essa situação, é comum criarem-se **packages** usando o domínio de *Internet* do projecto, invertendo os nomes. Por exemplo, todos os ficheiros de código fonte Java, criados para este manual, possuem como **package** o valor **org.sergio-lopes.formacao**, que pretende representar o domínio onde o manual se encontra alojado, nomeadamente **formacao.sergio-lopes.org**.

É necessário ter em atenção que existem algumas limitações nos nomes que podem ser dados aos **packages**: - nomes não devem necessitar de distinção maiúsculas/minúsculas já que para alguns sistemas operativos, uma pasta com o nome java e Java resultam na mesma pasta. - símbolos especiais não devem ser usados. - não se devem começar nomes de **packages** com números.

Capítulo 8. Métodos da Classe Object

A classe *Object* contém alguns objectos que todas as sub-classes devem redefinir, isto é, todas as classes que implementarmos devem redefinir estes métodos de modo a garantir um funcionamento adequado em toda a plataforma Java.

Nem todos estes métodos são necessários, e a classe *Object* oferece implementações que são usadas quando as subclasses não fornecem uma implementação própria, mas é conveniente que estejam implementados dado que alguns métodos e classes que existem na plataforma esperam que cada nova classe re-implemente estes métodos (ex: as classes do package java.util.collections, que fornecem estruturas de dados, esperam que os métodos **equals** e **hashCode** sejam redefinidos).

Método clone

Permite criar e devolver uma cópia do objecto. Para que este método possa ser usado, as classes precisam implementar a interface *Cloneable*. Por omissão este método devolve uma cópia da referência do objecto, efectuando um *shallow copy* [1]

Por omissão, o método clone da classe *Object* tem apenas:

```
public Object clone() {
    return this;
}
```

Como vemos, devolve apenas a referência para o objecto, o que não significa uma cópia real. Para que a cópia passe a ser correcta teremos de implementar o método clone **e** a interface *Cloneable* nas nossas classes.

```
public class Ponto implements Cloneable {
    private int x;
    private int y;

    public Ponto() {
        this(0,0);
    }

    public Ponto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    //criando um novo objecto usando o construtor para definir os valores
    public Ponto clone() {
        return new Ponto(x, y);
    }

    //ou definir os valores depois de usar o construtor e devolver a referência
    //util quando se pretende fazer outras operações
    public Ponto clone() {
       Ponto p = new Ponto();
       p.x = this.x;
       p.y = this.y;
```

[1] Uma shallow copy é o processo pelo qual apenas as referências são copias, ficamos assim com duas referências que apontam para o mesmo objecto e não com dois objectos iguais.

```
        return p;
    }
}
```

Método equals

Indica se o objecto passado por parâmetro é igual ao objecto usado para invocar o método. Por omissão, o método equals compara apenas referências, e como duas referências só são iguais se apontarem para o mesmo objecto, o método equals da classe *Object* apenas devolve verdadeiro se compararmos um objecto com ele próprio.

A implementação existente na classe *Object* é:

```
public boolean equals(Object obj) {
    return this == obj;
}
```

Como vimos esta implementação não nos serve de muito, uma implementação mais útil seria:

```
public class Ponto {
    private int x;
    private int y;

    public Ponto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {
        //se é o próprio ponto então são iguais
        if(this == obj) {
            return true;
        }

        //se não é instância da classe Ponto, nunca podem ser iguais
        if(!(obj instanceof Ponto)) {
            return false;
        }

        //Se não é o próprio ponto mas é uma instância de ponto
        //então vamos comparar os valores para ver se são iguais
        Ponto outro = (Ponto)obj;

        return this.x = outro.x && this.y = outros.y;
        }
}
```

Existem vários exemplos onde, em vez do operador **instanceof** se usa o método **getClass** para a comparação. As diferenças dependem do objectivo e são negligenciáveis. O resultado é o mesmo: confirmar que o objecto passado pode ser convertido para a classe e comparado.

Método hashCode

Este método deve ser re-implementado sempre que se re-implemente o método **equals** e sempre que se usem estruturas de dados *hashtable*, sejam a class *HashTable*, *HashMap* ou qualquer uma das várias implementações derivadas. Nestas estruturas, o valor devolvido pelo método **hasCode** é usado como índice.

O método **hashCode** deve devolver um inteiro que permita verificar se dois objectos são iguais ou diferentes, é um complemento ao método **equals**. Por omissão, a implementação deste método na classe *Object* apenas compara a referência, e não os valores, mas deve ser implementado de modo a que os valores dos objectos sejam comparados para que possamos garantir que, quando o método **equals** devolve verdadeiro para dois objectos, o método **hashCode** devolve um inteiro igual nos dois objectos.

Uma implementação típica, considerando como exemplo a classe seguinte:

```
public class Pessoa() {

    private nome;

    //...

    public boolean equals(Object obj) {
        //...
    }

    @Override
    public int hashCode() {
        //User dois inteiros aleatórios,
        //preferencialmente números primos
        int hash = 7;
        hash = 13 * hash + (this.nome != null ? this.nome.hashCode() : 0);
        return hash;
    }
}
```

Método toString

O método **toString** devolve uma *String* que representa o objecto. O valor devolvido pode ser qualquer coisa mas é tipicamente organizado de modo a devolver uma representação que facilite a nossa leitura, por exemplo, uma classe que represente um ponto poderá devolver as coordenadas sob a forma **(x, y)**, uma classe que represente uma pessoa poderá devolver o nome, etc.

Se não for re-implementado, este método devolve apenas uma hash que representa o endereço de memória que o objecto ocupa.

Exemplo.

```
public class Ponto {
    private int x;
    private int y;

    public Ponto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "("+ x + ", "+ y + ")";
    }
}
```

Capítulo 9. Sintaxe da Linguagem

Optou-se por introduzir a sintaxe da linguagem apenas no fim de modo a dar ênfase à componente de POO bem como à forma como a linguagem de programação Java usa o conceito de POO.

Recomenda-se que este capítulo seja consultado como referência e seja lido como os capítulos anteriores uma vez que pretende expor toda a sintaxe e regras associadas à programação em linguagem Java.

Características da Linguagem

- Linguagem **orientada a objectos**. Java segue o paradigma de programação orientada a objectos footnote: [O conceito de POO foi explicado nos capítulos iniciais.].

- Linguagem **independente da plataforma e da arquitectura**. Um programa criado em Java é compilado para //bytecode//, este código é universal e entendido por todas as JVMs. Desta forma o código não fica limitado à plataforma onde foi compilado, e pode ser executado sem alterações em qualquer plataforma onde exista uma JVM. No entanto o //bytecode// criado pode ser compilado para a plataforma, tornando-se assim código nativo. Isto é conseguido recorrendo ao compilador Just-in-Time, JIT, que analisa o código e o torna nativo footnote: [Existem implementações de compiladores que transformam o código Java directamente para código nativo, no entanto esses compiladores não fazem parte da filosofia de portabilidade do Java.].

- **Distribuída.** Possui primitivas de comunicação de alto nível como carregamento de recursos via URL, Remote Method Invocation, RMI, e CORBA IIOP. Possui primitivas de comunicação de baixo nível através de Sockets. Permite também carregar código de forma transparente a partir de qualquer lugar [1].

- **Robusta**. Não possui ponteiros. Em Java os ponteiros são substituídos por referências e toda a aritmética de ponteiros desaparece. Existe a verificação de acesso a tabelas e operações de //cast// em //runtime//.

- **Segura**. Acrescentando ao facto de que o código é verificado durante a execução, de que código desconhecido é executado isoladamente, de que não é possível aceder a índices fora de tabelas e de que não existem ponteiros com os seus acessos perigosos à memória, o Java permite ainda que o código seja autenticado através de assinaturas digitais e certificados. Bem como a criação de gestores de segurança que limitam a execução de código a tarefas inócuas.

- Suporta **multi-threading**. As bibliotecas são //thread safe//, e o suporte para //threads//, além de fazer parte da biblioteca padrão, através da classe Thread e interface Runnable, está embutido na linguagem através de métodos sincronizados e semáforos associados a cada classe.

Tipos de dados

Java possui uma mistura de tipos de dados primitivos e objectos footnote: [O grosso dos tipos de dados de Java são primitivos, os tipos de dados que sejam objectos nativos da linguagem são casos específicos usados em principalmente para sistemas com necessidade de elevada precisão.], todos os tipos de dados

[1] É possível que a nossa aplicação esteja dividida, com umas classes num servidor e outras classes noutra, a execução da aplicação, e até a programação da mesma, é transparente para o utilizador ou para o programador, não alterando em nada o facto da aplicação estar repartida por máquinas diferentes.

têm tamanho fixo ou seja, quer em sistemas de 64bit quer em sistemas de 32bit, em Java, os tipos de dados mantém sempre o mesmo tamanho, ao contrário de outras linguagem compiladas nativamente.

A precisão de um tipo de dados é garantida em qualquer plataforma, e mantém-se sempre igual.

O quadro seguinte apresenta uma descrição dos tipos de dados primitivos existentes em Java, bem como dos seus intervalos, valor máximo, mínimo e classe de encapsulamento que permite passar um tipo primitivo para um objecto.

Tipo Primitivo	**Tamanho**	**Valor mínimo**	**Valor máximo**	**Valores possíveis**	**Classe de adaptação** [a]
boolean	1 bit	-	-	true, false	Boolean
char	16 bit	Unicode 0	Unicode 2^16 - 1	*a* b' *c*, etc.	Character
byte	8 bit	-128	128	Qualquer valor dentro da gama. Formato pode ser decimal(10, 12, -1), octal(010, 017) ou hexadecimal(0x2e, 0xff)	Byte
short	16 bit	-32.762	32.762	Qualquer valor dentro da gama. Formato pode ser decimal(10, 12, -1), octal(010, 017) ou hexadecimal(0x2e, 0xff)	Short
int	32 bit	-2.147.483.648	2.147.483.648	Qualquer valor dentro da gama. Formato pode ser decimal(10, 12, -1), octal(010, 017) ou hexadecimal(0x2e, 0xff)	Integer
long	64 bit	-9.223.372.036.854.775.808	9.223.372.036.854.775.808	Qualquer valor dentro da gama. Suporta os formatos decimal, octal e hexadecimal acrescentando um L ou l ao final. Ex.: 0x7fffffffffffffL	Long
float	32 bit	+/-3,4e-38	+/-3,4e+38	Qualquer valor dentro da gama. Formato idêntico ao do long mas com a letra F ou f.	Float

Tipo Primitivo	Tamanho	Valor mínimo	Valor máximo	Valores possíveis	Classe de adaptação [a]
double	64 bit	+/-1.7e-308	+/-1.7e+308	Qualquer valor dentro da gama. Formato decimal ou científico(2.2e-2 ou 2.2E-2)	Double

[a]Todos os tipos de dados nativos possuem um objecto que é responsável por fornecer encapsulamento a esse tipo de dados. Dessa forma podemos “transformar” um tipo primitivo num objecto.

Todos os tipos de dados possuem sinal e como tal, não existe a necessidade nem o suporte para o modificador *unsigned*, que noutras linguagens tornava uma variável com sinal numa sem sinal.

Variáveis

Java é uma linguagem *strongly-typed*, isto significa que todas as variáveis precisam ser declaradas antes de serem usadas. Embora possam ser declaradas em qualquer parte do ficheiro de código, é comum agrupar as declarações no topo da classe ou no fim da classe, não deixando assim que existam variáveis "perdidas" no meio outros blocos de código.

Declarar uma variável não é mais que indicar a sua visibilidade, se aplicável, o seu tipo de dados, o seu nome e, opcionalmente, uma iniciação. Exemplos:

```
private int idade = 55; //declaração com indicação de visibilidade e valor inicial

double taxa;            //apenas declaração, a visibilidade será assumida
                        //através do contexto em que a variável é declarada
```

As declarações acima indicadas são para os tipos primitivos já apresentados, a declaração de objectos é similar, contendo apenas o uso da palavra reservada **new** quando se pretende indicar o valor inicial. Exemplo: private Pessoa p = new Pessoa();

Sem valor inicial, a declaração é em tudo igual a uma declaração dos tipos primitivos: private Pessoa p;

Além de indicar à plataforma que uma variável existe e está pronta a receber dados, a declaração permite também definir o tipo de operações que podem ser aplicadas sobre as variáveis. Tipos primitivos não permitirão o mesmo tipo de operações que objectos, e vice-versa.

Java possui *boxing* e *unboxing* automático de tipos primitivos desde a sua versão 5. Embora não da mesma forma que em C#, este tipo de operação permite aliviar o programador de usar explicitamente algumas das classes de encapsulamento de dados primitivos, sendo estas usadas pela JVM quando necessário.

O processo de *boxing* e *unboxing* faz com que, em locais onde seja esperado um objecto numérico, caso o programador passe um valor numérico primitivo, a JVM coloca a variável primitiva dentro de um objecto automaticamente. Veremos casos destes no código exemplo.

Operadores

Os operadores de Java são comuns a muitas outras linguagens, efectivamente são em grande parte similares aos operadores presentes em C e em C+. Mas ao contrário de C+, Java não permite o

overloading de operadores, isto é, em Java não é possível alterar o significado de um operador de soma ou subtracção, que está definido apenas para valores numéricos e para tipos primitivos.

Em alguns casos é natural que não haja necessidade de se poder somar dois objectos, afinal não faz sentido somar um objecto **Carro** com um objecto **Poste**, embora na vida real existam muito condutores que o façam. Mas para outros tipos de objectos, por exemplo um objecto **Ponto2D**, faz sentido poder somar duas instâncias da classe **Ponto2D**, afinal, a soma de dois pontos num sistema de duas dimensões é um processo perfeitamente normal.

Como foi referido, Java não permite a redefinição dos operadores mas possui algumas redefinições, que embora não cubram todos os casos, cobrem os mais comuns, é assim possível somar dois objectos de um tipo de dados que encapsule um tipo primitivo, por exemplo, somar dois objectos **Double**.

Tabela de precedência de operadores, os operadores mais perto do topo são os que possuem maior precedência, e dessa forma são avaliados primeiro.

Operadores	**Precedência**
postfix	expr++ expr--
unitários	++expr --expr +expr -expr ~ !
multiplicação	* / %
adição	+ -
shift	<< >> >>>
relacionais	< > <= >= instanceof
igualdade	== !=
E bit-a-bit	&
E exclusivo bit-a-bit	%%^%%
OU exclusivo bit-a-bit	%%
%%	E lógico
&&	OU lógico
%%	
%%	ternário
? :	atribuição

Atribuição

O operador de atribuição é, possivelmente, o operador mais simples. É este operador que permite atribuir valores a variáveis, pegando no valor da direita e atribuindo-o ao operando da esquerda.

```
int altura = 10;
long preco = 5.5;

altura = 15 + altura;
```

Operadores Aritméticos

Os operadores aritméticos são equivalentes aos usados na matemática e, dessa forma, não serão estranho. O único que poderá não ser conhecido é o operador de resto, que devolve o resto de uma divisão inteira.

- Operador de adição, também redefinido para concatenação de strings: **+**
- Operador de subtracção: **-**
- Operador de multiplicação: ***
- Operador de divisão: **/**
- Operador de resto: **%**

Talvez seja necessário introduzir o conceito de divisão inteira: uma divisão entre dois números inteiro resultará sempre num resultado inteiro. Isto é uma regra que, pelo menos em Java, não é quebrada. Assim ao efectuarmos a conta **5/2** o resultado é **2** e não **2,5**.

Neste ponto entra o operador de resto, ao efectuarmos a operação **5%2** o resultado será **1**, que é o resto de uma divisão inteira entre 5 e 2. Podemos fazer como na primária, *5 a dividir por 2 dá 2, resto 1*.

Operadores Unitários. Os operadores unitários resultam da aplicação especial dos operadores aritméticos que requer apenas um operando, estas operações permitem efectuar operações de incrementação, decrementação ou alterações de sinal. O único operador que não faz parte dos operadores aritméticos e que é um operador unitário, é o operador de negação.

- Definição de números positivos: **+**, em Java todos os números são positivos por omissão
- Definição de números negativos: **-**, permite colocar uma expressão como negativa
- Operação de incremento: **++**
- Operação de decremento: **--**
- Complemento lógico: **!**, inverte o valor lógico de um operando

Operadores Condicionais, de Igualdade e Relação

- Igualdade: **==**
- Diferença: **!=**
- Maior que: **>**
- Maior ou igual: **>=**
- Menor que: **<**
- Menor ou igual a: **<=**
- E condicional: **&&**
- OU condicional: **||**

- Comparador de tipos: **instanceof**, o operador de comparação de tipos permite verificar se um objecto corresponde a um tipo (classe), ex:

```
String s1 = "s1";
if(s1 instanceof String) {
    System.out.println("Objecto é uma String");
} else {
    System.out.println("Objecto não é uma String");
}
```

Operadores Bitwise e Bit Shift

Para terminar os operadores são apresentados operadores que alteram bits. Estes operadores actuam sobre a representação binária, interna, dos valores das variáveis.

- Complemento unitário bit-a-bit: ~, inverte um padrão de bits, tornando todos os zeros em uns e todos os uns em zeros, ex: **111000110** passará a **00111001**
- Shift à esquerda com sinal: <<, move para a esquerda um conjunto de bit
- Shift à direita com sinal: >> , move um conjunto de bits para a direita

Os operadores de shift com sinal movem um conjunto de bits cujo padrão é definido pelo operando da esquerda e o número de posições é definido pelo operando da direita.

- Shif para a direita *sem* sinal: >>>, este operador introduz um zero na posição mais à esquerda.
- E bit-a-bit: **&**
- OU exclusivo bit-a-bit: ^
- OU inclusivo bit-a-bit: |

Expressões, Blocos e Condições

Expressões são linhas de código composta por variáveis, operadores e invocação de métodos, agrupados de acordo com a sintaxe da linguagem que são avaliadas como um valor único. Exemplo:

```
int **idade = 5**;
**vector[5] = 1**;
System.out.println(**"Valor em 5: " + vector[5]**);
```

Blocos são todos os pedaços de código delimitados por chavetas, {}. Não são mais que uma forma de agrupar determinadas expressões ou condições que devem ser executadas em conjunto.

Através da delimitação de blocos conseguimos definir o código a executar depois de uma condição ou durante um processo especial de iniciação. Permitem-nos também definir o início e o fim de uma classe ou um método.

Estruturas de Controlo de Fluxo de Programa

Designamos *estruturas de controlo de fluxo de programa*, as estruturas que permitem determinar que linhas de código executar de um modo condicional. São estruturas que nos permitem dizer em que

condições determina linha ou conjunto de linhas pode ser executado e são estas estruturas que nos permitem criar programas mais complexos. No entanto, as estruturas em si são bastante simples.

Em Java todas as condições são avaliadas como valores verdadeiro/falso footnote: [Condições verdadeiro/falso são expressões cujo resultado seja sempre um valor do tipo verdadeiro(true) ou falso(false). Por vezes podem ser identificadas como expressões *boleanas*, mas o termo não corresponde a uma tradução correcta.], assim, todas as estruturas de controlo de fluxo usam os valores verdadeiro/falso como valores de controlo [2].

if-then e if-then-else

Esta é a estrutura mais simples que se pode ter num programa [3]. Esta estrutura permite decidir que código executar mediante o resultado de uma condição.

```
if(condição) {
    //código a executar se a condição for verdadeira
}

if(condição) {
    //código a executar se a condição for verdadeira
} else {
    //código a executar se a condição for falsa.
}
```

Na sua primeira versão, o código dentro das chavetas [4] é executado quando a condição é verdadeira. Se a condição for falsa o programa não executa o código dentro das chavetas e "salta" para a chaveta de final, retomando a execução nesse ponto [5].

Na segunda versão, como adicionamos um bloco para contemplar o facto de a condição poder ser falsa, o programa avalia a condição, se for verdadeira executa o código dentro do bloco do **if**, se for falsa executa o código dentro do bloco do **else**. Depois, a execução prossegue normalmente a partir da chaveta final do bloco do **else**.

Esta estrutura pode ser encadeada de forma a permitir código mais complexo. Podemos colocar novas secções **if** em cada secção **else**, por exemplo:

```
if(condicao1) {
    //código a executar se a condição 1 for verdadeira
} else if(condicao2) {
    //código a executar se a condição 2 for verdadeira
} else if(condicao3) {
    //código a executar se a condição 3 for verdadeira
} else if(condicao4) {
    //código a executar se a condição 4 for verdadeira
} else if(condicao5) {
    //código a executar se a condição 5 for verdadeira
} else {
    //código a executar se todas as condições anteriores forem falsas
}
```

[2] A estrutura **switch** pode parecer contraditória a esta afirmação mas internamente o resultado é que as suas cláusulas são avaliadas como condições verdadeiro/falso

[3] Embora nos refiramos a esta estrutura como **if-then/if-then-else**, a palavra *then* não é usada como palavra reservada. Serve apenas para ajudar na leitura.

[4] O uso das chavetas não é obrigatório, mas se não usarmos chavetas apenas a linha imediatamente abaixo da condição é executada.

[5] Este é um comportamento comum a todas as estruturas de controlo de fluxo de código, se a condição é falsa, a execução retoma na chaveta que delimita o bloco correspondente à condição verdadeira. Se existir um bloco de código para situações onde a condição é falsa, esse bloco é executado.

Embora simples e versátil, a estrutura **if-then-else** obriga à escrita de estruturas complexas e longas.

switch

A estrutura **switch** actua como um selector, escolhendo blocos de código mediante um valor que é passado. Exemplo:

```
switch(valor) {
    case 1:
         //código a executar se o valor é igual a 1
         break;
    case 2:
         //código a executar se o valor é igual a 2
    case 3:
         //código a executar se o valor é igual a 3
         break;
    default:
         //código a executar se o valor não corresponde a nenhuma das opções mencionadas acima.
}
```

Um **switch** aceita valores primitivos do tipo byte, **short**, **char**, **int**, tipos enumerados, (que não foram abordados neste manual) e classes encapsuladoras como *Character*, *Byte*, *Short*, e *Integer* [6].

No nosso exemplo, a estrutura irá seleccionar o bloco de código consoante o valor passado. Assim, se valor for 1 o primeiro bloco de código é executado, ao encontrar a instrução **break**, o código termina e retoma a execução na chaveta final do bloco do **switch**. Caso o valor seja 2, o bloco de código correspondente vai ser executado, como não existe uma instrução **break**, o código continuará a executar para o bloco seguinte [7].

O bloco **default** será executado se o valor passado for diferente de todos os valores existentes no **switch**, como é o último bloco não é necessário ter uma instrução **break**

while e do-while

Tal como a estrutura **if-then** a estrutura **while** executa um bloco de código quando a condição é verdadeira, e ao contrário da estrutura **if-then-else** não é bloco para condição falsa. Mas enquanto as duas estruturas que vimos inicialmente executam o código uma vez apenas, a estrutura **while** executa o bloco de código enquanto a condição se mantiver verdadeira.

```
while(condição) {
    //código a executar enquanto a condição é verdadeira
}
```

A estrutura **while** é útil quando pretendemos executar um determinado bloco um número de vezes não especificado, isto é, sabemos que queremos repetir o código, mas não temos a certeza de quantas vezes o vamos repetir.

A par com a estrutura **while** temos a estrutura **do-while**, em tudo similar mas com a particularidade de testar a condição apenas no fim do bloco. O resultado é que o bloco de código é executado sempre, pelo menos, uma vez. Se a condição for falsa o bloco não volta a executar, se for verdadeira é executado novamente [8].

[6] Ao usarmos classes encapsuladores o compilador aplica o conceito de *boxing* e *unboxing* automático que já vimos anteriormente.
[7] Esta é uma característica do **switch**, a execução não termina ao encontrar o fim do bloco mas sim quando chegar ao fim do **switch**, pelo que é comum que um bloco de código termine com uma instrução **break**.
[8] Esta é a estrutura mais comum quando, durante aprendizagem de uma linguagem de programação, se pretendem fazer menus em modo de texto. Permite mostrar sempre o menu uma vez e se o utilizador pretender sair, ter a opção para o fazer.

```
do {
    //código a executar
} while (condição)
```

for e for each

O ciclo, ou estrutura de controlo **for** permite iterar facilmente um conjunto de valores, tipicamente presentes num vector [9]. O uso típico desta estrutura é quando pretendemos executar o código um número de vezes conhecido [10].

```
for(iniciação de variáveis; condição; incremento de variáveis) {
    //código a executar
}
```

Nesta estrutura, possuímos três secções, a primeira onde podemos fazer alguma iniciação de variáveis, a segundo onde colocamos a condição de controlo, e a terceira onde executamos incrementos. Nenhum das três secções é obrigatória e podemos omitir um, duas, ou todas as secções [11]. Exemplo:

```
for(int i = 0; i < 6; i++) {
    System.out.println(i); //imprimir o valor da variável i
}

int j;
for(j = 5; j > 0; j--) {//iniciamos em 5 e, em vez de incrementar, decrementamos o valor de j
    System.out.println(j);//imprimir o valor da variável j
}

for(; i < 6; i++) {//omitimos o código de iniciação
    System.out.println(i);//imprimir o valor da variável i
}

for(int i = 0, j = 5; i < 5 && j > 0; ++i, --j) {//uso de mais que uma variável, e condição mais c
    System.out.println("I: " + i + "\tJ: " + j);//imprimir o valor das duas variáveis
}

int i = 0;
for(;; i++) {//omitimos a condição de paragem, ciclo infinito.
    System.out.println(i);//imprimir o valor da variável i
}
```

No Java 5, foi adicionada uma nova estrutura, que em outras linguagem se denomina por **foreach**, mas que em Java reutiliza uma estrutura existente: o **for**.

Assim, quando o que estamos a percorrer são objectos iteráveis [12] podemos usar o **for** para facilitar o acesso aos seus elementos.

Antes do Java 5, para percorrer uma lista seria necessário fazer:

```
Iterator it = lista.iterator();// obter o iterador da lista que pretendemos percorrer

while(it.hasNext()) {
    Object o = it.next();//obter o elemento.
    System.out.println(o);//imprimir o valor de o
}
</code>
```

[9] O ciclo **for** é, possivelmente, a estrutura com mais variações, sendo usada de formas bastante variadas, muitas vezes dependentes do estilo de programação de quem a usa. Vamos apenas concentrar-nos na forma tradicional de usar a estrutura.
[10] Ao contrário da estrutura **while**.
[11] Neste caso ficamos com um ciclo em execução constante, um ciclo infinito.
[12] Objectos iteráveis são objectos criados de classes que implementam alguma das interfaces de iteradores, tipicamente são listas ou vectores.

```
Com o Java 5 o código anterior pode ser alterado para:
<code java5>
for(Object o : lista) {//Estamos a usar uma lista com elementos do tipo Object
    System.out.println(o);//imprimir o valor de o
}
```

Um **foreach** é mais útil quando usamos também genéricos, isto porque o método **next()** da classe *Iterator* devolve um objecto do tipo *Object*, mesmo que na nossa lista tenhamos colocado objectos do tipo *String*, o que nos obriga a fazer operações de **cast** [13] sempre que depois queiramos usar correctamente o nosso objecto. Com o **foreach** somos obrigados a declarar o tipo da variável, se o declararmos directamente para o tipo de objectos que estão dentro da lista, evitamos estar a fazer um **cast** [14].

break, continue e return

Estas são três instruções usadas para alterar a execução de ciclos quando a condição de paragem não é suficiente ou não se aplica, permitindo terminar a estrutura, ou saltar ciclos, evitando assim a execução de algum código mas não terminando o ciclo.

Quando aplicadas dentro de ciclos aninhados [15], estas instruções alteram o funcionamento do ciclo interior se forem usadas na sua forma mais simples, ou alteram o ciclo exterior quando são usadas em conjunto com etiquetas [16].

O uso de **labels** não goza da melhor das reputações, em outras linguagens as **labels** funcionam de forma diferente, muitas vezes associadas à instrução **goto**, e permitem saltos incondicionais para qualquer local no código. Por muito útil que um salto incondicional possa parecer, a sua utilização é extremamente perigosa, dando origem a código confuso, difícil de ler e de manter e onde e detecção de erros é muito difícil de fazer. Em Java a instrução **goto** não existe.

Labels em Java, mais limitadas mas mais seguras, permitem à instrução **break** ou **continue** alterar o ciclo externo, mas estão sempre presas ao ciclo que iniciaram e não permitem a mesma utilização que a instrução **goto** permite noutras linguagens.

Já a vimos a instrução **break** na estrutura **swith** mas ela pode ser usada em estruturas **for**, tanto na sua forma tradicional como quando usada como **foreach**, e em estruturas **while** e **do-while**.

Para terminar, encontramos a instrução **return**. Esta instrução não termina apenas ciclos, devolve o controlo de execução, ou fluxo de programa, para o código que invocou o método que está a ser executado. Assim, ao encontrar uma instrução **return** num ciclo, esse ciclo é terminado imediatamente, o próprio método onde o ciclo está a ser executado termina e o controlo é devolvido a quem chamou o método.

A instrução **return** permite devolver um valor, caso seja necessário, bastando para isso colocar o valor depois da instrução:

```
return 5;//devolver o valor 5
return "Terminei";//devolver a string
return new Object(); // devolver um objecto criado neste ponto.
```

[13] Fazer um **cast** não é mais que converter um tipo noutro tipo. Uma conversão de **cast** pode ser impossível de fazer se os tipos forem incompatíveis, e pode também provocar perda de informação como vimos nos capítulos anteriores

[14] Na verdade existe sempre uma cast, mas neste caso não é explicito pelo programador e pode ser optimizado pelo compilador sempre que possível.

[15] Ciclos que se encontram dentro de outros ciclos.

[16] Labels no original e daqui em diante.

Método main

Todo o programa precisa de um ponto de entrada, de um local onde possa ter início a execução e processamento. Várias linguagens conseguem isso de formas distintas, no caso do Java, o ponto de entrada de um programa é o método **main**. Este método pode existir em qualquer classe do nosso programa, podem inclusive existir vários métodos **main**. Caso isto aconteça, apenas um será invocado de cada vez que a aplicação é iniciada, sendo o método escolhido pelo utilizador, falaremos deste ponto mais abaixo.

O método é *public* porque a JVM, fora do nosso *package* e que não entra na nossa hierarquia de classes precisa de poder aceder ao método, se o método fosse *private*, apenas podia ser acedido pela classe que o contém, se fosse *protected* apenas pela classe e suas subclasses, se possuísse visibilidade de *package*, apenas podia ser acedido por outras classes dentro do mesmo *package*. Desta forma é imprescindível que o método seja público.

Possui como tipo de retorno *void* porque o método não retorna qualquer valor. Em outras linguagens, como C/C++, é comum o método principal retornar um valor inteiro que representa o estado com que o programa terminou, em Java isso não acontece, e quaisquer erros existentes deverão ser tratados pelos mecanismos apropriados.

Recebe um vector de *String* que contém os parâmetros passados pela linha de comandos, ou através de qualquer sistema gráfico que permita passar parâmetros a aplicações. Neste caso, apenas os parâmetros da aplicação são passados, o nome da aplicação não é passado como primeiro argumento, novamente, ao contrário de outras linguagens onde o nome da aplicação é o contado como um argumento.

E finalmente o ponto mais complicado, ou pelo menos o que costuma causar mais dúvidas: porque é que é um método estático?. A resposta é simples, se o método é marcado como *static* significa que é um método de classe, logo não é necessário obter uma instância do objecto para o poder invocar, e este é o ponto fundamental. Se a JVM necessitasse de instanciar um objecto para invocar o método **main** contido na sua classe existiriam algumas restrições a esse objecto. O que acontecia se a classe possuísse vários construtores? Qual seria o construtor invocado? Que parâmetros seriam passados aos construtores? Que consequências teria a complexidade dos construtores no arranque da aplicação? Enfim, um conjunto de dúvidas que surgiriam e que para serem resolvidas obrigariam a que a classe que contivesse o método **main** fosse quase uma classe especial, e obrigando o programador a restringir em grande parte o que poderia programar nessa classe.

Ao ser um método estático todos os problemas acima são eliminados, o programador é livre de criar qualquer classe como entender e, se pretender adicionar um ponto de entrada na aplicação, apenas precisa de adicionar um método estático com a assinatura acima definida.

Não existe nada mais associado ao método **main** que o facto de ter de ser escrito sempre da forma indicada e servir para iniciar a nossa aplicação.

Foi anteriormente mencionado que é possível ter na mesma aplicação mais que um método **main**. Esse não será o caso mais comum, uma aplicação tem apenas um ponto de entrada, ao ter dois, podemos assumir que são duas aplicações distintas, mas por vezes é útil que assim seja. Por exemplo, podemos construir uma aplicação com uma interface gráfica e ao mesmo tempo oferecer uma opção de linha de comandos. Se possuirmos dois métodos **main** é mais simples oferecer-mos ao utilizador este tipo de escolha. Podemos também pretender criar classes e métodos de teste, que ao serem colocados numa segunda linha de execução, podem permitir a sua fácil gestão e não perturbam o código normal da aplicação.

Os motivos para estas opções são vários, o que é importante é que se perceba que é possível, e que não acarreta qualquer problema ou invalida a nossa aplicação, a existência de mais do que um método main em classes diferentes.

Determinar qual o método a executar já dependerá da forma como a aplicação é iniciada. Se distribuída num *JAR* deverá ser definido um método a ser invocado por omissão através do *Manifest-file*, se a aplicação não for compactada num ficheiro *JAR*, o utilizador terá de indicar a classe que contém o método **main** que deseja iniciar.

A forma de escrever o método **main** tem de ser sempre:

```
public static void main(String[] args) {
    // restante código
    //...
}
```

Palavras Reservadas

Esta secção apresenta uma lista com as palavras reservadas e seu significado.

abstract	Define um método ou classe abstracta.
assert	Permite o uso de testes. Deve ser usada apenas para situações de depuração de código e testes.
boolean	Tipo de dados primitivo que permite dois valores apenas *true* e *false*.
break	Permite alterar o fluxo de programa.
byte	Tipo de dados primitivo, aceita valores na ordem de um byte.
catch	Componente da estrutura de tratamento de excepções.
case	Componente da estrutura **switch**
char	Tipo de dados primitivo que representa um caractere UTF-8.
class	Palavra reservada que identifica a declaração de uma classe.
const	Palavra reservada sem uso actual.
continue	Permite continuar para a próxima iteração de um ciclo sem executar as instruções que ainda faltam.
default	Identifica o bloco de código a executar num *switch* quando nenhuma das opções *case* é escolhida
do	Componente do clico *do…while*.
double	Tipo de dados primitivo, permite valores com precisão dupla.
else	Palavra reservada parte da estrutura de decisão **if-then-else**.
enum	Permite criar uma enumeração.
extends	Parte do mecanismo de herança, indica a classe da qual uma outra classe descende.
final	Modificador que impede a alteração de um atributo, criando assim uma constante, a redefinição de um método ou a herança de uma classe.
finally	Componente do bloco try..cath que permite a execução, em qualquer circunstância, de um bloco de código.
float	Tipo de dados primitivo, valores de vírgula flutuante de precisão simples.
for	Estrutura de repetição.

goto	Palavra reservada mas não utilizada na linguagem.
if	Estrutura de decisão.
implements	Palavra reservada que identifica as interfaces implementadas pela classe afectada.
import	Palavra reservada que permite importar classes de outros *packages* de modo a serem usados na classe importadora.
instanceof	Permite inferir o tipo de objecto.
int	Tipo de dados primitivo que representa inteiros.
interface	Permite especificar uma interface.
long	Tipo de dados primitivo, valor de virgula flutuante de precisão dupla.
native	Indica que um método é implementado de forma nativa, em C ou C\++.
new	Permite instanciar um objecto, reservando a memória e efectuando os passos necessários para a correcta utilização do objecto.
package	Permite definir a organização das classes, interfaces ou enumerações.
private	Modificador de visibilidade mais restritivo de todos, coloca a visibilidade do elemento afectado como privada
protected	Modificador de acesso, o recurso afectado, que pode ser uma classe, um método ou uma variável, passa a ser visível apenas pelas subclasses e classes do mesmo *package*.
return	Devolve o controlo do código para o método que o invocou.
short	Tipo de dados primitivo, representa inteiros de 16 bit.
static	Indica um método, atributo ou bloco estático.
strictfp	Força o uso das regras definidas pelo IEEE754, permitindo garantir o valor de vírgula flutuante que se está a manipular.
super	Permite aceder aos atributos e métodos da super-classe que possuem visibilidade suficiente. Todos menos os privados.
switch	Estrutura de decisão.
synchronized	Palavra reservada, usada no mecanismo de *threads*, e que permite sincronizar um bloco de código, impedindo o acesso a esse código por mais que uma *thread* de cada vez.
this	Referência especial para o objecto em que estamos.
throw	Palavra reservada que permite levantar uma excepção. throws::Permite indicar as excepções que um método lança e que devem ser apanhadas por quem os invoca.
transient	Marca um atributo de um objecto como não sendo persistido por mecanismos de serialização.
try	Parte do mecanismo de excepções.
void	Indica que um método não possui valor de retorno.
volatile	Permite indicar que uma variável vai ser acedida e modificada por várias *threads*.
while	Estrutura de repetição.

Comentários

Como qualquer linguagem, também Java possui comentários que permitem explicar métodos, atributos ou simples pedaços de código. Estes comentários podem ser feitos de várias formas, podendo existir em formato de uma linha, várias linhas, comentários especiais para documentação, etc.

De seguida são apresentados exemplos de 3 tipos de comentários.

```
/*Isto é um

comentário que se estende por

várias linhas*/

//Isto é um comentário de linha única

/** Isto é um comentário para ser usado na documentação ou JavaDoc
 *  Alguns IDEs geram documentação automaticamente a partir deste tipo de comentários deste tipo.
 *  tags que podem ser usadas para formatar o aspecto final da documentação mas isso é outra histó
 */
```

Documentar o código é uma das tarefas mais importantes quando se programa, embora seja complicado para o programador iniciante perceber porque motivo tem de comentar o código que ele próprio faz, e que supõe ninguém irá ver, a verdade é que não é possível prever quem realmente irá ler o nosso código e, mesmo que consideremos remota a hipótese de alguém que não nós venha a ler o código, facilmente esquecemos o motivo que nos levou a escrever determinada linha se olharmos para ela daqui a 6 meses.

Uso Completo da Sintaxe Apresentada

As seguintes cinco classes pretendem servir de exemplo e mostrar vários aspectos da sintaxe da linguagem. As classes representam um exemplo de como criar um sistema muito simples para gerir pessoas. É possível criar pessoas e definir os seus atributos base.

```
package org.sergiolopes.formacao;

import java.util.GregorianCalendar;
//Importe de uma enumeração dentro de uma classe:
import java.util.TimeZone;
import org.pap.wiki.javatutorial.Person.Sex;

/**
 * Classe de teste.
 * Pode ser usada para correr a aplicação e ver alguns resultados.
 */
public class RunMe {

    public static void main(String[] args) {
        Person antonio = new Person("António");
        GregorianCalendar gMaria = new GregorianCalendar(TimeZone.getTimeZone("UTC"));      //invo
        gMaria.set(1980, 5, 6);     //Ano: 1980, Mês: 5 (Junho dado que começam em zero), Dia: 6
        Person maria = new Person("Maria", "Luz", gMaria, Sex.FEMALE);

        System.out.println("== Exemplo de sintaxe Java ==");
        System.out.println(antonio);
        System.out.println("A pessoa mencionada " + (antonio.producesMilk() ? "" : "não ") + "amam
        System.out.println("--\n");
        System.out.println("Maria, idade " + maria.getAge() + "anos.\nSexo: " +
                (maria.getSex() == Sex.FEMALE ? "Feminino" : "Masculino"));
    }
```

```
}
```

```
package org.sergiolopes.formacao;

/**
 * Interface criada apenas para mostra a possibilidade de se usar mais que uma
 * interface na mesma classe e para mostrar a sintaxe simples que permite criar
 * uma interface.
 */
public interface Driver {

    /**
     * Este é um método definido numa interface. Todos os métodos de uma
     * interface precisam ser públicos ou possuir visibilidade de package,
     * métodos privados ou protegidos não podem ser definidos em interfaces.
     *
     * Nem podia ser de outra forma, se uma interface obriga à implementação do
     * método não podia querer que o método fosse privado, de outra forma como é
     * que poderia ser implementado? Se fosse protegido apenas as subclasses
     * podiam implementar o método, mas se estamos numa interface, a subclasse
     * de uma interface é também uma interface.
     *
     * @param km parâmetro sem sentido que serve apenas para confirmar que um
     * método de uma interface é similar a um método de uma classe, seja
     * abstracta ou não, não possuindo diferenças na sua sintaxe de declaração.
     */
    public void drive(int km);
}
```

```
package org.sergiolopes.formacao;

/**
 * Classe que representa um mamífero.
 * Esta classe pretende apenas mostrar a sintaxe de uma classe abstracta e
 * permitir mostrar a sintaxe usada para criar subclasses.
 */
public abstract class Mamal {

    /**
     * Métodos abstractos.
     * Estes métodos terão de ser implementado numa das subclasses, não têm
     * necessariamente de ser implementado no primeiro nível ou em qualquer
     * outro nível específico, mas algures na hierarquia descendente desta
     * classe terão de ser implementados.
     *
     * As diferenças de um método abstracto para outros métodos são o uso da
     * palavra reservada <em>abstract</em> e o facto de não existir o corpo do
     * método com o código correspondente.
     *
     * Este comentário mostra também o uso de <i>tags</i> de formatação,
     * <b>tags</b> emprestadas do HTML, e que permitem dar alguma formatação aos
     * comentários. Esta formatação aparece nas ajudas de alguns IDEs e através
     * de ferramentas de geração de documentação, como o <em>Javadoc</em>.
     */
    public abstract void drinkMilk();
    public abstract boolean producesMilk();
}
```

```
package org.sergiolopes.formacao;

/**
 * Classe que representa uma morada.
 * Contém os campos e métodos necessários para criar e manter uma morada
 * correcta.
 */
public class Address {
```

```
    private String street;
    private String city;
    private String zipCode;
    private String country;

    public Address(String street, String city, String zipCode, String country) {
        setStreet(street);
        setCity(city);
        setZipCode(zipCode);
        setCountry(country);
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        if (city == null || city.isEmpty()) {
            throw new IllegalArgumentException("Uma morada necessita de uma cidade!");
        }
        this.city = city;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        if (street == null || street.isEmpty()) {
            throw new IllegalArgumentException("Uma morada necessita da rua!");
        }
        this.street = street;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        if (zipCode == null || zipCode.isEmpty()) {
            throw new IllegalArgumentException("Uma morada necessita de um código" +
                    "postal!");
        }
        this.zipCode = zipCode;
    }

    @Override
    public boolean equals(Object obj) {
        return false;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 41 * hash + (this.street != null ? this.street.hashCode() : 0);
        hash = 41 * hash + (this.city != null ? this.city.hashCode() : 0);
        hash = 41 * hash + (this.zipCode != null ? this.zipCode.hashCode() : 0);
        hash = 41 * hash + (this.country != null ? this.country.hashCode() : 0);
```

```
        return hash;
    }

    @Override
    public String toString() {
        return street + "\\n" + zipCode + "\\n" + city + ((country != null && !country.isEmpty())
                ? " - " + country : "");
    }
}
```

```
package org.sergiolopes.formacao;

import java.io.Serializable;
import java.util.GregorianCalendar;

/**
 * Classe que representa uma pessoa.
 *
 */
public class Person extends Mamal implements Serializable, Driver {

    private static final long uid = 0l;
    private String name;
    private String surname;
    private GregorianCalendar birthDate;
    private Sex sex;
    private Address address;

    public Person(String name) {
        this(name, "", null, Sex.MALE, null);
    }

    public Person(String name, String surname, GregorianCalendar birthdate, Sex sex) {
        this(name, surname, birthdate, sex, null);
    }

    public Person(String name, String surname, GregorianCalendar birthdate, Sex sexType, Address a
        this.name = name;
        this.surname = surname;
        this.birthDate = birthdate;
        sex = sexType;
        address = addrs;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public GregorianCalendar getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(GregorianCalendar birthDate) {
        this.birthDate = birthDate;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public Sex getSex() {
        return sex;
    }

    public void setSex(Sex sex) {
        this.sex = sex;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public int getAge() {
        return calculateAge();
    }

    /**
     * Método privado que permite efectuar o cálculo da idade da pessoa.
     * @return
     */
    private int calculateAge() {
        return (int) ((System.currentTimeMillis() - birthDate.getTimeInMillis()) /
                1000 / 3600 / 24 / 355);
    }

    /**
     * Método que somos obrigados a redefinir para que esta classe possa ser
     * concreta, se não o redefinirmos, a classe terá de ser abstracta.
     *
     * Este método introduz também as anotações que podem ser usadas em
     * métodos e atributos. Neste caso a anotação @Overried que
     * indica que um método é a redefinição de outro acima na hierarquia.
     *
     * Este método, apesar de implementado, não produz qualquer resultado útil.
     * Uma invocação do método irá provocar o lançamento de uma excepção.
     */
    @Override
    public void drinkMilk() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public boolean producesMilk() {
        return sex == Sex.FEMALE;
    }

    /**
     * Se repararem este método, que provém de uma interface, não possui a
     * anotação @Override, isto porque o que estamos a fazer não é redefinir um
     * método mas sim a definir. Por outras palavras, este método não existe em
     * lugar algum na hierarquia de classes da classe onde estamos, é sim,
     * definido por uma interface que esta classe, por acaso, implementa.
     *
     * Desta forma a anotação não faz sentido e se for colocada o compilador irá
     * emitir um erro.
     * @param km
     */
    public void drive(int km) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
```

```
    public void saveToFile(String filename) {
        //TODO
    }

    /**
     * Embora não esteja implementado, este método exemplifica a sintaxe de um
     * método estático.
     *
     * @param filename Nome do ficheiro a ler.
     */
    public static void readFromFile(String filename) {
    }

    public String toString() {
        return name;
    }

    /**
     * Demonstração de uma enumeração.
     *
     * Nota-se a semelhança com a declaração de classes e interfaces, onde se
     * altera apenas a palavra <em>class</em>/<em>interface</em> para
     * <em>enum</em>.
     *
     * A sintaxe usada no corpo da enumeração é ligeiramente diferente, mas
     * introduz apenas algumas coisas novas que são especificas das enumerações.
     *
     * Tal como as enumerações, é possível ter classes internas, isto é,
     * classes dentro de outras, e tal como as classes, as enumerações podem
     * existir no seu próprio ficheiro.
     */
    public enum Sex {

        /**
         * Uso do construtor privado para que apenas possam ser criados dois
         * tipos de sexo, com os valores que pretendemos.
         */
        MALE("Male"),
        FEMALE("Female");
        private final String type;

        /**
         * Um construtor, como qualquer outro método ou atributo, pode ser
         * privado, dessa forma apenas a classe que o define lhe pode aceder.
         * @param type
         */
        private Sex(String type) {
            this.type = type;
        }

        @Override
        public String toString() {
            return type;
        }
    }
}
```

Bibliografia

- Bruce Eckel. Thinking in Java, 3rd Edition. Prentice Hall PTR. 6 de Dezembro de 2002.
- Apontamentos das cadeiras de Programação 3 e Programação 4 da ESTG Leiria. 2004.