2 - JAVA: Tecnologia e Introdução à Linguagem

2.1 Introdução

Ainda que o objectivo fundamental deste curso seja a **Programação por Objectos** (PPO) usando a linguagem JAVA5, JAVA é hoje em dia uma palavra reservada que significa muito mais do que uma outra linguagem de programação por objectos, pelo que se torna importante compreender as razões que fazem com que seja actualmente tão relevante conhecer tal linguagem, saber usá-la como ferramenta para **programar por objectos**, bem como conhecer a tecnologia subjacente à linguagem e perceber as razões do tão grande impacto causado na área da Informática por esta tecnologia, genericamente designada **tecnologia JAVA**.

2.2 Génese da tecnologia JAVA

A **Engenharia de *Software*** sempre procurou soluções para a máxima "escrever código uma vez e reutilizá-lo sempre que possível". A deu, por várias razões, algumas esperanças quanto à possibilidade de implementar tal máxima. Porém, à questão simples "onde, em que plataforma, tal reutilização pode ter lugar?", as respostas não foram satisfatórias. Ou seja, tal máxima poderia até ser cumprida em ambientes monolíticos, monoplataforma, mas a estes se limitava. Em termos concretos, tal não era sequer completamente aplicável às grandes organizações, que tipicamente se baseiam em ambientes multiplataforma e multitecnologia.

A linguagem Smalltalk foi durante muitos anos um bom exemplo desta filosofia de reutilização, já que se baseava num compilador que gerava um código intermédio, para o qual haveria apenas que desenvolver interpretadores específicos para cada possível plataforma. A ideia estava, pois, e desde os anos 70, perfeitamente correcta. Porém, o aparecimento da linguagem C++ e de outras linguagens de PPO tais como Objective-C e ObjectPascal, bem como outras que, não o sendo de forma muito pura, ofereciam ambientes atractivos de desenvolvimento de programas (pelo menos para PC), tais como VisualBasic, Delphi e VisualC++, fez com que a adopção de uma destas linguagens e uma séria aposta a uma escala mais alta do que a da simples programação (o projecto, o desenvolvimento, a manutenção, etc.) fosse sendo adiada pelos grandes construtores e pelas grandes empresas de *software*.

A IBM ainda chegou a anunciar em 1994/95 a linguagem Smalltalk como a sua linguagem de PPO, tendo até desenvolvido um ambiente de projecto, o VisualAge-Smalltalk. Hoje restam, pragmaticamente, C++, C# e JAVA.

A linguagem JAVA começou a ser desenvolvida no início dos anos 90 no seio de uma pequena equipa de engenheiros de *software* da Sun Microsystems, liderada por James Gosling. O objectivo era desenvolver uma pequena linguagem para equipamentos electrónicos com *chips* programáveis, tais como torradeiras, máquinas de lavar, agendas electrónicas de bolso, etc. Os principais requisitos da linguagem a desenvolver eram a robustez e a segurança (os utilizadores destes dispositivos não admitem erros ou falhas), o baixo custo (os programas teriam que ser simples) e a independência dos *chips* (dado que os construtores com grande facilidade os substituem). O projecto, de 1991, designava-se *Green*.

Numa primeira fase, o modelo do UCSD Pascal, uma das mais bem sucedidas linguagens para PC, ainda terá sido considerado, ou seja, a geração de um código intermédio após a compilação − o *p-code* − e a criação de interpretadores para arquitecturas específicas. Em geral, o *p-code* era pequeno e os interpretadores de *p-code* eram igualmente de reduzida dimensão. A maioria dos engenheiros de *software* do grupo tinha formação em Unix e, portanto, grandes conhecimentos de C e C++, pelo que pretenderam usar os conhecimentos já

adquiridos e a sua segurança nestas linguagens. Porém, as análises feitas pela equipa às linguagens C e C++ revelaram não serem estas as linguagens adequadas, por um lado, dada a sua complexidade e dificuldade associada em escrever código garantidamente seguro e, por outro, por necessitarem de um compilador específico para cada tipo de *chip*. A equipa desenvolveu então a linguagem JAVA, inicialmente designada *Oak*, **simples**, **segura e independente de arquitecturas** (1992). O primeiro produto lançado foi um controlo remoto extremamente inteligente, o *7, que, apesar de tudo, ninguém pretendeu produzir.

Em 1993, a equipa é confrontada com o aparecimento da *World Wide Web* via Internet e do extraordinário impacto produzido. JAVA perfila-se então, dadas as suas características, como uma linguagem fundamental para a Internet.

A sua neutralidade relativamente a arquitecturas torna-a ideal para o desenvolvimento de aplicações que, com facilidade, pudessem ser executadas em qualquer das diferentes máquinas ligadas à Internet.

A grande demonstração das capacidades da linguagem JAVA como linguagem de programação para a Internet (www) foi o aparecimento dos *applets* JAVA, consistindo em pequenas aplicações programadas em JAVA e que podiam ser executadas no contexto de outras aplicações. A equipa de Gosling desenvolveu, então, um *Web browser* JAVA, o *HotJava*, que foi o primeiro *browser* a poder executar *applets* no contexto de páginas HTML, tornando-as assim não apenas interactivas mas também dinâmicas, ou seja, mutáveis em conteúdo e aparência. *HotJava* foi provavelmente a maior demonstração do poder da linguagem JAVA e, possivelmente, o ponto fulcral de consagração e aceitação desta tecnologia desenvolvida pela Sun Microsystems. Companhias como a IBM, a Netscape, a Oracle, a Symantec, a Borland, a Corel e outras, aderiram, a partir de então, ao projecto JAVA. Em 1995, a Netscape lança a versão 2.0 do seu *browser*, já aceitando *applets* e *scripts* JAVA (usando a linguagem JavaScript).

Em 1995, na *Sun World Conference*, a linguagem JAVA foi oficialmente anunciada.

Alguns autores afirmam que, para além das capacidades e potencialidades intrínsecas da linguagem, com vocação — via bibliotecas adequadas — para processamento usando tecnologia cliente-servidor e fácil acesso a comunicações, JAVA terá tido a «sorte», ou a visão comercial dos seus autores, de ter visto a sua JVM distribuída com os dois *Web browsers* de maior sucesso no mundo: Netscape e Explorer. A penetração de JAVA no mercado foi, por esta razão, para além dos méritos intrínsecos da linguagem, de um nível esmagador.

Os passos seguintes da curta história de JAVA são: a criação, em 1996, da JavaSoft pela Sun, empresa destinada a desenvolver produtos JAVA para as mais diversas áreas e que em 1998 apresenta já um catálogo de centenas de produtos; o lançamento do ambiente de desenvolvimento JDK1.0 em 1996; a primeira conferência sobre produtos JAVA, a *JavaOne Conference*, em 1996; a adesão da Corel em 1996, que acaba por falhar; o lançamento do JDK1.1 em Fevereiro de 1997 já com JavaBeans e outras facilidades de desenvolvimento; o anúncio da plataforma Java Card 2.0 em 1997; e o processo em tribunal da Sun contra a Microsoft, em Outubro de 1997, por esta empresa ter desenvolvido código JAVA para o Explorer 4.0 que não passou nos testes de compatibilidade de JAVA da Sun, requisito acordado na licença de JAVA assinada pela MicroSoft. A Microsoft vingar-se-ia tecnologicamente em 1998, ao apresentar (cf. testes realizados por peritos da PC Magazine em Abril de 1998) a mais compatível JVM para Windows, e o então melhor ambiente de desenvolvimento, o Microsoft J++. Em Janeiro de 1998, a Sun conquista a TCI (Tele-Comunications Inc.), o maior fornecedor americano de infra-estruturas de TV Digital. Em 1999, a plataforma JAVA 2 e a versão beta de J2EE são lançadas. Em 2000, é lançada a J2EE e o JDK1.2. Em 2003, 75% dos programadores profissionais americanos usam JAVA como a sua principal linguagem de programação. Em finais de 2004, a plataforma J2SE5 (Projecto *Tiger*) com o JDK5 é lançada, tendo sido introduzidas as maiores alterações na linguagem JAVA desde a sua criação. Em 2005, é anunciada a plataforma JSE6 de nome *Mustang*.

2.3 O QUE É A TECNOLOGIA JAVA

A ideia forte associada à linguagem JAVA é **escrever código uma vez e executá-lo em qualquer parte**. Para que tal seja possível, o código fonte das aplicações escritas em JAVA tem que ser isolado pelo ambiente JAVA dos sistemas operativos das máquinas e dos dispositivos de *hardware*. Logo, tal código não pode ser

compilado directamente para código nativo das máquinas, mas antes para uma representação especial, neutra, designada por *byte-code*. Em seguida, este código intermédio será interpretado sobre o ambiente particular de cada máquina, para tal sendo desenvolvido um interpretador particular do *byte-code* para cada plataforma onde se pretendam executar programas JAVA. Este *runtime-engine*, ou motor de execução, designa-se por *JAVA Virtual Machine* (JVM) e deve ser instalado no computador cliente de código JAVA (Figura 2.1).

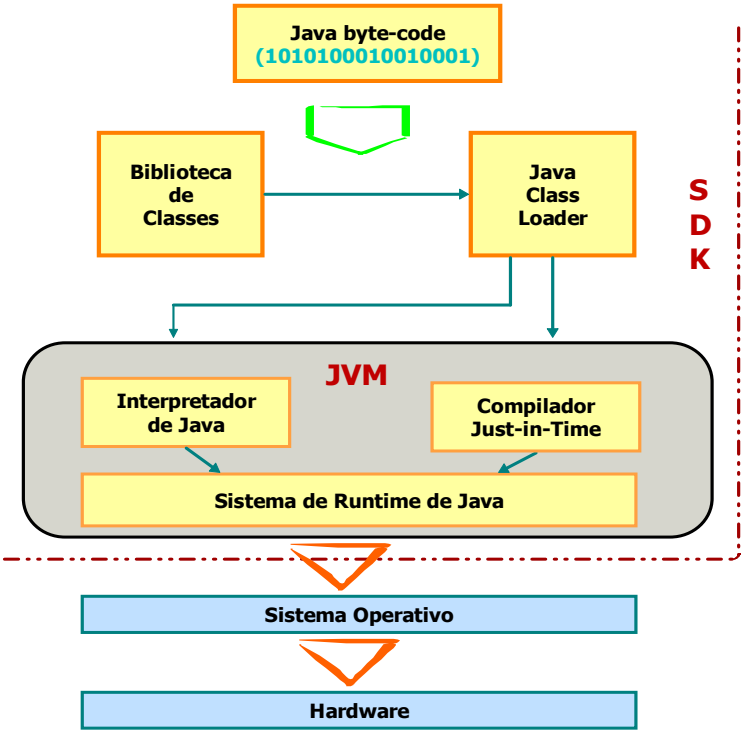

Figura 2.1 – Ambiente de execução de JAVA

A JVM recebe *byte-code* e transforma-o em instruções executáveis na máquina particular onde o ambiente JAVA é instalado. Assim, como é fácil compreender, existem JVMs para diferentes *browsers*, sistemas operativos (ex.: Windows, Linux, Solaris, Mcintosh, etc.), dispositivos electrónicos (ex.: telemóveis, PDA, consolas de jogos, satélites, etc.), enfim, qualquer dispositivo programável.

É importante desde já perceber que o que torna JAVA uma linguagem muito atractiva para toda a indústria de computadores, é não ser apenas uma nova linguagem de programação, ainda por cima por objectos e adicionalmente quase totalmente pura, mas sobretudo porque JAVA se posicionou como um atractivo e bem apetrechado **ambiente de programação e desenvolvimento de aplicações** no contexto actual, para o que foram desenvolvidas plataformas específicas consoante o tipo de desenvolvimento de *software* a realizar.

Assim, e de base, existem três tipos de plataformas: as **JSE** (*Java Standard Edition*), as **JEE** (*Java Platform Enterprise Edition*) e as **JME** (*Java Platform Micro Edition*).

As siglas destas plataformas e dos respectivos ambientes de desenvolvimento foram, com a saída da versão 1.5.0 da plataforma *Java 2 Standard Edition* (J2SE), normalizadas. De facto, quer para a linguagem JAVA quer para as plataformas, as designações 1.0, 1.1 e 1.2 referem-se a versões de desenvolvimento, enquanto que 1, 2 ou 5 se referem a versões de produto. Por isso, e relativamente à linguagem propriamente dita, há livros de JAVA, de JAVA2 e agora de JAVA5. A explicação é portanto a mesma, ou seja, as versões 1.3 e 1.4 saíram mas não foram quase notadas.

Quando a Sun lançou a versão 1.2 das suas plataformas, passou a designá-las por J2SE e J2EE, respectivamente, referindo-se-lhes como "plataforma 2". Agora, os nomes e versões são J2SE 5.0, JDK 5.0 (para o sistema de desenvolvimento da J2SE), JRE 5.0, J2EE 5.0, J2EE 5 SDK e J2ME5.

As plataformas **JSE** são plataformas vocacionadas para o desenvolvimento de aplicações em computadores pessoais ou *workstations* com arquitecturas mono ou multiprocessador e servidores. São constituídas ao mais baixo nível pelo *Java Runtime Environment* (**JRE**) e por diversas camadas de utilitários de desenvolvimento que constituem o respectivo *Java Standard Development Kit* (**SDK** ou **JDK**). As plataformas JSE são as mais comuns e mais usadas, não apenas pelas suas características, mas também porque é a partir delas que são desenvolvidas as plataformas JEE.

As plataformas **JEE** destinam-se ao desenvolvimento de aplicações tipo cliente-servidor para redes, intranets e Internet, muito centradas em serviços *Web on-line*, processamento de páginas HTML, transacções com bases de dados, transacções seguras, protocolos e múltiplas interfaces gráficas com os utilizadores. As plataformas JEE baseiam-se na tecnologia **JSP** (*Java Server Pages*) e em *servlets* (programas não autónomos a executar no servidor) para a criação de páginas HTML como forma de apresentação de dados aos clientes de uma aplicação. A comunicação com bases de dados é realizada através de um protocolo designado por *Java Database Connectivity* (**JDBC**) que é em JAVA o equivalente ao protocolo *standard* **ODBC** (*Open Data Base Connectivity*). Componentes designados por *Enterprise JavaBeans* (**EJB**), associados a um servidor próprio deste tipo de serviços, fornecem diversas funcionalidades importantes, tais como gestão de concorrência, de segurança e de memória.

As plataformas **JME** são ambientes de desenvolvimento para dispositivos móveis ou portáteis, tais como telemóveis, PDA, etc. Esta plataforma contém configurações e bibliotecas próprias para que quem desenvolve possa ultrapassar dificuldades próprias das limitações de processamento e memória destes dispositivos. Configurações como as **CLDC** (*Connected Limited Device Configuration*) e **MIDP** (*Mobile Information Device Profile*) são duas das mais utilizadas. Dentro das plataformas para microambientes, JAVA disponibiliza também uma plataforma para programação de cartões magnéticos inteligentes, designada por *Java Card Platform* (**JCP**).

Na Figura 2.2 apresenta-se de forma pormenorizada a arquitectura em camadas da plataforma actual, a **J2SE5** "Tiger", distinguindo-se JDK, JRE e JVM.

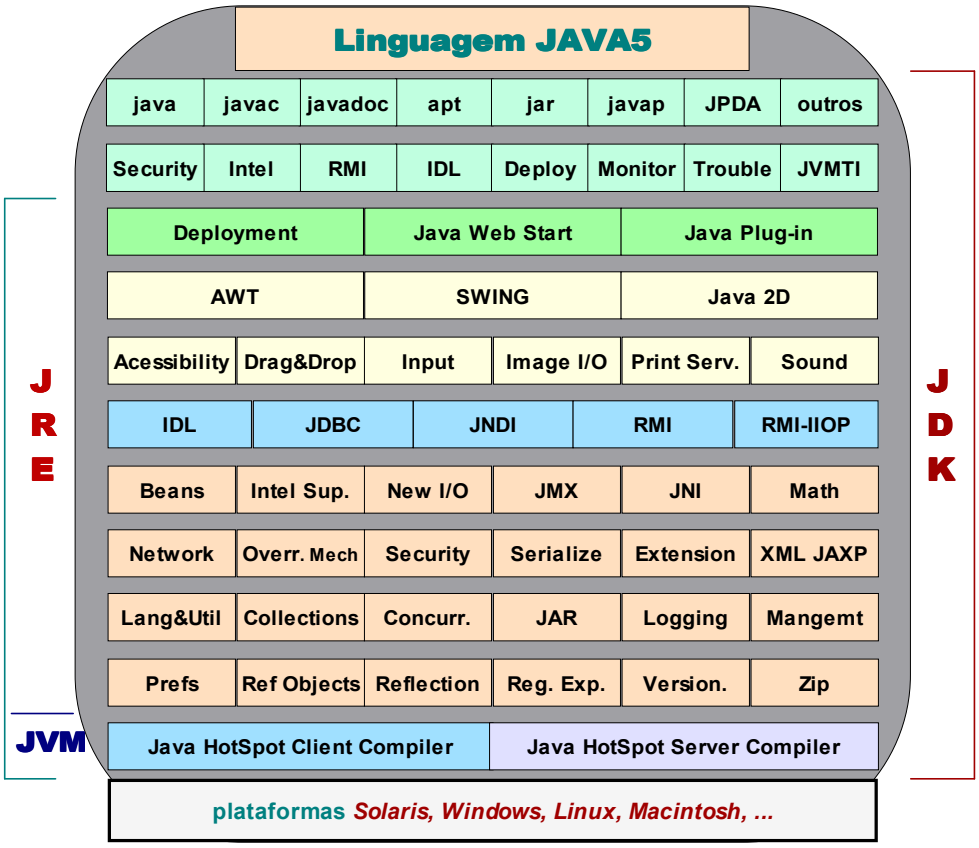

Figura 2.2 – Arquitectura da plataforma J2SE5

Assim, e como se pode verificar pela figura, o JDK é uma extensão ao JRE. Quem apenas pretender executar aplicações JAVA, apenas terá que instalar o JRE com a JVM certa para o seu sistema operativo. Quem pretender, como nós, desenvolver aplicações JAVA, tem que instalar sobre o JRE o sistema de desenvolvimento JDK. O JDK5 fornece entre outros, os serviços necessários à compilação do código fonte (*javac*), à execução de programas (*java*), à produção de documentação (*javadoc*), ao *debugging* (*JPDA*) e à produção de arquivos (*jar*).

A linguagem JAVA pode ser usada para desenvolver dois tipos de programas:

- aplicações;
- *applets* e *servlets*.

As **aplicações JAVA** são programas que, após serem compilados, apenas requerem uma JVM para serem interpretados e executados, podendo esta existir ao nível do sistema operativo ou ligada à própria aplicação, pelo que tais programas executam por si próprios, ou seja, são o que, em geral, se designa por aplicações *stand alone*.

Applets são porções de código JAVA não executáveis por si próprias, dado que são dependentes da existência de um *browser* que incorpore e execute a JVM (cf. Netscape ou Explorer) para que estas possam ser executadas. Como as *applets* são escritas muitas das vezes com o objectivo de poderem ser «descarregadas» através da rede, são em geral de dimensão muito mais pequena do que as aplicações, consistindo, por exemplo, em pequenas *forms* para entrada de dados e outros componentes de interfaces com o utilizador.

Servlets são porções de código não executável por si próprio, executado em contextos próprios pelos servidores (por exemplo, um servidor de *http*), de forma a responderem através de páginas HTML a certos pedidos dos clientes, realizados através de eventos e segundo protocolos bem estabelecidos. Ao contrário das *applets*, que têm mais interface com o utilizador do que conteúdo, os *servlets* são praticamente apenas conteúdos que devem ser enviados dos servidores às aplicações cliente.

Exemplos concretos da aplicação de *applets*, *servlets* e de todas as tecnologias de bases de dados, *networking*, encriptação, gestão de versões, etc., que referimos antes, são os modernos *websites* comerciais baseados em transacções electrónicas. No entanto, as mesmas tecnologias são empregues, e ainda mais algumas, quando os clientes passam de PC para telemóveis ou PDA, mantendo-se os servidores os mesmos.

A Figura 2.3 mostra a evolução das várias plataformas e da linguagem JAVA desde a sua versão original JAVA1.0 (1995) até JAVA5 (2004).

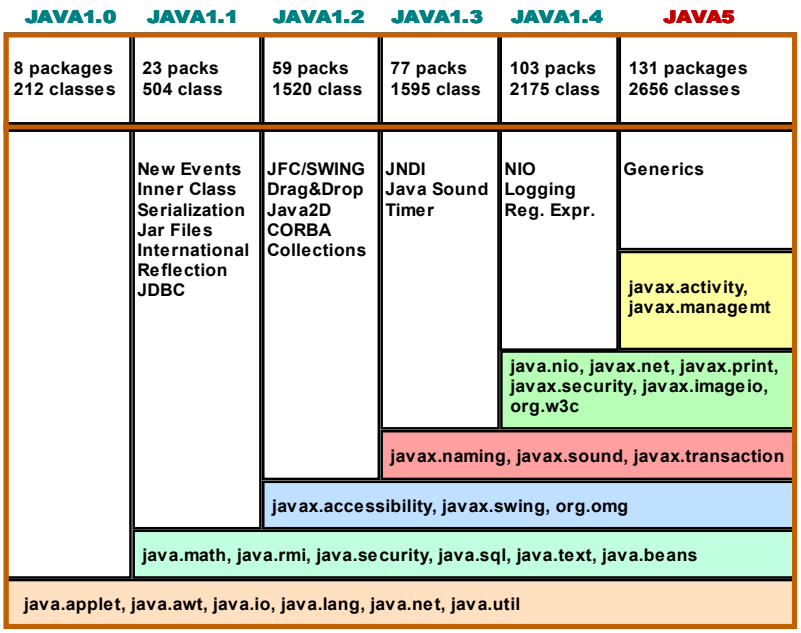

Figura 2.3 – Evolução da linguagem JAVA

Como se pode verificar, os sucessivos desenvolvimentos, em geral bibliotecas de classes que montam um determinado "serviço" a aplicações, cobrem as mais diversas áreas.

Todo este manancial tecnológico colocado ao dispor de quem desenvolve aplicações, em especial aplicações distribuídas, transaccionais, *on-line*, que têm que ser seguras, com fácil extensibilidade dadas as mudanças permanentes, de fácil manutenção, etc., tornam a tecnologia JAVA, a linguagem JAVA, que é o elemento aglutinador, e o paradigma da programação por objectos, elementos fundamentais na Informática actual.

2.4 A LINGUAGEM JAVA: CARACTERÍSTICAS

Esta secção 2.4 é completamente dedicada à apresentação do nível mais básico da linguagem, em termos de conceitos e de construções necessários para se escreverem os primeiros programas em JAVA. Não serão ainda programas de PPO, ou seja, usando classes e objectos, porque o nível dos objectos é criado a partir deste nível mais básico, onde não existem objectos mas apenas valores simples associados aos usuais tipos primitivos das linguagens de programação. Este é o designado nível primitivo (ou dos valores) mas no qual já poderemos programar como programaríamos em Pascal ou em C.

2.4.1 INTRODUÇÃO

A linguagem JAVA é uma linguagem de programação orientada aos objectos que, ao contrário de Smalltalk, e tal como muitas das actuais linguagens de PPO, não é uma linguagem por objectos cem por cento «pura», dado que nem todos os seus componentes são **objectos**.

Torna-se por isso importante desde já deixar claro que, na linguagem JAVA, existe um nível em que, tal como em qualquer linguagem não OO, lidamos com entidades que designamos por **constantes** e **variáveis**, que têm a si associados **valores** (dados). Estas entidades dos programas têm um **identificador** que nos permite designá-las nas nossas instruções, e é através desse identificador que temos acesso aos valores que elas representam e guardam. Estas constantes e variáveis, através de declarações convencionais, são associadas a **tipos de dados**, que servem para definir exactamente o conjunto de valores que tais variáveis e constantes podem guardar e representar.

JAVA possui pois um nível, existente como nível de suporte ao nível fundamental que é o nível das construções OO, em que a programação se faz declarando variáveis e constantes como sendo de um dado **tipo**, manipulando estes valores usando operadores predefinidos para tais tipos e, até, recorrendo a estruturas de controlo comuns nas linguagens de tipo imperativo, como sejam as estruturas de controlo condicionais e repetitivas, que têm até formas sintácticas muito semelhantes, ou até iguais, às existentes na linguagem C.

Este é o nível em JAVA designado por **nível dos tipos primitivos** (tais como inteiros, reais, carácter, boleano, etc.) e respectivos operadores, ou, se quisermos, o nível dos **não-objectos**. Fundamentalmente, a este nível trabalhamos directamente com valores. As variáveis contêm valores e é sobre tais valores que realizamos operações e programamos algum controlo de execução. Como veremos mais adiante, este é o nível de suporte à programação por objectos em JAVA.

A Figura 2.4 procura caracterizar melhor esta relação entre o **nível dos tipos primitivos** e o **nível dos objectos** e da programação OO.

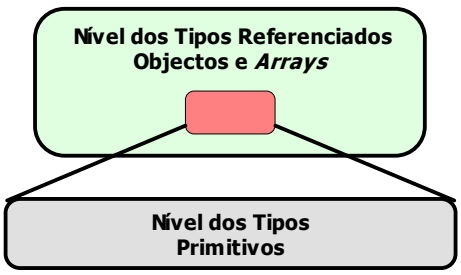

Figura 2.4 – Os níveis das entidades de JAVA

Tal como as setas da figura induzem, existirá em JAVA a possibilidade de, em certos casos, converter **valores** em **objectos** e certos objectos em valores, ou seja, em síntese, a camada dos tipos primitivos e respectivos valores é completamente convertível na camada de objectos. No entanto, o inverso não é, naturalmente, sempre possível, nem sequer aceitável ou recomendável, pois implicaria uma pura redundância.

O estudo da linguagem JAVA prosseguirá assim sob duas perspectivas: em primeiro lugar, serão apresentadas as construções relacionadas com o nível dos tipos primitivos, construções em tudo semelhantes às usadas em linguagens do tipo imperativo. Com as entidades deste nível, já será possível codificar os primeiros programas em JAVA, mas não serão certamente programas com objectos; no entanto, as entidades deste nível são toda a base da criação das entidades ao nível dos objectos; em seguida, serão introduzidos, um a um, os pilares fundamentais do paradigma da PPO, e, para cada um destes, as correspondentes construções existentes em JAVA.

2.4.2 NÍVEL DOS TIPOS PRIMITIVOS

JAVA tem oito tipos primitivos que correspondem aos valores que podemos usar e transformar quando programamos ao nível mais básico da linguagem. São estes: `int`, `short`, `long`, `double`, `float` e `byte` (para representação de valores numéricos inteiros ou reais), `char` (para valores que são caracteres) e `boolean` (para os dois valores lógicos).

Os tipos primitivos de JAVA, a sua gama de valores, os respectivos valores por omissão e a sua dimensão em *bits*, são sintetizados na Tabela 2.1. Assim, quando uma variável for declarada como sendo de um destes tipos, ser-lhe-á de imediato atribuído o valor por omissão definido para esse tipo, poderá apenas assumir valores dentro da gama de valores indicada e ocupará um número fixo de *bits* igualmente indicado.

Tipo	Valores	Omissão	*Bits*	Gama de valores
`boolean`	false, true	false	1	false a true
`char`	caract. unicode	\u0000	16	\u0000 a \uFFFF
`byte`	inteiro c/ sinal	0	8	-128 a +127
`short`	inteiro c/ sinal	0	16	-32768 a +32767
`int`	inteiro c/ sinal	0	32	-2147483648 a 2147483647
`long`	inteiro c/ sinal	0L	64	≈ -1E+20 a 1E+20
`float`	IEEE 754 FP	0.0F	32	± 3.4E+38 a ± 1.4E-45 (7 d)
`double`	IEEE 754 FP	0.0	64	± 1.8E+308 a ± 5E-324 (15d)

Tabela 2.1 – Tipos primitivos de Java

A primeira característica importante destes tipos primitivos de JAVA é o facto de as suas representações em termos de dimensão (número de *bits*) não serem dependentes do compilador, sendo portanto fixas e iguais em qualquer ambiente de execução.

O tipo `char` comporta caracteres em código Unicode (UCS – *Universal Character Set* e UTF – *Unicode Transformation Format*), que é um *standard* de 16 bits que integra em si o código ASCII usual. O próprio código fonte é todo transcrito para caracteres Unicode, pelo que caracteres tais como Δ, θ, π, δ, etc. são válidos, e não apenas o conjunto usual de caracteres ASCII. Cada carácter tem um código decimal (código ASCII ou fora deste) cujo valor em hexadecimal é o seu código Unicode. Por exemplo, a letra A que tem por código ASCII o valor 65, representa-se em Unicode por '\u0041'. O prefixo \u, quando usado na representação de caracteres, significa exactamente o código Unicode de um determinado carácter. Os caracteres podem ser representados explicitamente usando o prefixo \ (*escape characters*), sendo os mais usados '\t' de tab e '\n' de nova linha.

Os tipos numéricos são os usuais inteiros e reais. Os números reais são representados por `float` ou `double`, sendo o tipo `float` pouco utilizado devido à sua relativa falta de precisão, pois "apenas" trabalha com sete dígitos decimais. As constantes reais são, por omissão, do tipo `double` (ex.: `321.423`). Para que sejam `float`, deverão ser terminadas com o sufixo `F` ou `f` (ex.: `56.2334F`). As constantes de tipo `long` devem ser

terminadas pelo sufixo `L` ou `l` (ex.: `3400562L`) e os valores do tipo `byte` podem ser escritos em hexadecimal se iniciados pelo prefixo `0x` (ex.: `0xFAFE`) e em octal se iniciados pelo prefixo `0` (ex.: `0777`).

Os tipos `float` e `double` permitem representar valores em vírgula fixa ou flutuante. Valores em vírgula flutuante são expressos sob a forma de um número seguido da letra `E` ou `e` (representativa da base 10) e do valor do expoente inteiro. São exemplos de valores famosos representados em vírgula flutuante: 9.109E-28, 2.9979E+10, 8.5e25 e 5.52e0.

O tipo `boolean` tem apenas dois valores, `true` e `false`, sendo `false` < `true`.

2.4.3 DECLARAÇÕES DE VARIÁVEIS E INICIALIZAÇÃO

Variáveis são identificadores de dados, sendo os seus nomes importantes na codificação e, como se sabe, devem ser nomes mnemónicos, ou seja, informativos dos dados que representam. Antes disso, porém, têm que ser sintacticamente válidos.

Em JAVA, um identificador de uma variável não poderá ser nenhuma das palavras reservadas da linguagem, devendo iniciar-se por uma letra, carácter $ ou _, seguida de uma sequência de letras, dígitos ou caracteres $ ou _. O compilador distingue entre maiúsculas e minúsculas. Por convenção linguística, os **identificadores de variáveis de JAVA são sempre iniciados por letras minúsculas**.

JAVA é o que se designa por uma linguagem **fortemente tipada**, ou seja, o compilador faz uma verificação rigorosa dos tipos de dados utilizados nas expressões, tendo por base os tipos declarados para as variáveis do programa, nos vários contextos em que estas são declaradas e usadas.

Assim, qualquer variável, antes de poder ser utilizada, deve ser associada a um tipo de dados através de uma declaração, e inicializada com um valor através de uma atribuição (=). Uma variável declarada e não inicializada tem, em geral, um valor que é indefinido, gerando a sua utilização um erro de compilação.

Vejamos alguns exemplos de declarações de variáveis, algumas com inicialização:

```
int total;
double media = 0.0; // declaração e atribuição de 0.0
boolean encontrado = false;
char letra = 'z';
```

Associadas a um tipo de dados, poderemos ter uma sequência de declarações de variáveis separadas por vírgulas, eventualmente com atribuições dos valores de expressões para a inicialização de algumas delas, cf. a forma geral e os exemplos seguintes:

id_tipo *id_variável [= valor] [, id_variável [= valor] ...] ;*

```
int dim = 20, lado, delta = 30;
char um = '1';
char c = 'A';
char newline = '\n';
char letraA = '\u0041';  // formato UNICODE cf. \u (decimal 65)
char tab = '\u0009';     // formato UNICODE
byte b1 = 0x49;          // hexadecimal, cf. 0x como prefixo
long diâmetro = 34999L;
double raio = -1.7E+5;
double j = .000000123;       // parte inteira igual a 0
int altura = dim + delta;    // inicialização por expressão
```

As variáveis de tipos primitivos que forem declaradas no contexto da definição da estrutura interna de uma dada classe, variáveis que mais tarde designaremos por "variáveis de instância", caso não sejam explicitamente inicializadas através de código do programa, sê-lo-ão pelo compilador, que lhes atribuirá valores iniciais por omissão em função do seu tipo (ver Tabela 2.1).

Finalmente, não é aconselhável que na mesma linha de código se façam múltiplas declarações de variáveis, pois tal torna o código menos claro e impede que comentários individuais de clarificação de cada uma delas sejam realizados.

2.4.4 DECLARAÇÕES DE CONSTANTES

Como veremos posteriormente, as declarações de **constantes** são semelhantes às declarações de variáveis, mas requerem a inclusão de um atributo particular que especifica que tais inicializações são imutáveis (usando o atributo `final`).

As constantes JAVA são também, por razões de estilo e de legibilidade, identificadas usando apenas letras maiúsculas, em certos casos usando-se também o símbolo _.

As seguintes declarações:

```
final double PI = 3.14159273269;
final double R_CLAP = 8.314E+7;
final double GCGS = 6.670E-8;
```

definem, num dado contexto, **constantes identificadas**, cujos valores não poderão ser alterados por nenhuma instrução.

2.4.5 CONVERSÃO ENTRE TIPOS: *CASTING*

Os tipos numéricos são compatíveis entre si dentro de algumas regras simples. Numa dada operação na qual entrem dois operandos de tipos numéricos distintos, o operando com menor precisão é convertido no tipo do operando de maior precisão, sendo este o tipo do resultado. A escala crescente de precisões é, como sabemos, a seguinte:

```
byte → short → int → long → float → double
```

A um tipo numérico mais à direita, logo de maior precisão, é igualmente possível atribuir um valor correspondente a um tipo mais à esquerda (de menor precisão). São pois válidas as seguintes expressões:

```
int dim = 100;
float area = 100*100;
double vol = area*100;
double x = 3456F;
```

Por outro lado, por vezes, mesmo admitindo a perda de precisão, pretende-se realizar conversões em sentido contrário. Existe um operador que pode ser usado de forma unária para converter valores de dado tipo à sua direita para valores de dado tipo indicado à sua esquerda, designado por operador de *casting*, por exemplo `(id_tipo) operando`.

Por exemplo, é para todos aceitável que um valor do tipo `float` possa ser convertido num `int`. Assim, usando o mecanismo de *casting* ou de "conversão de tipos" e as declarações apropriadas, escreveríamos:

```
float lado = 123.45F;
int li = (int) lado;  // li = 123
```

É também aceitável que um inteiro (`short` ou `int`) possa ser associado ao tipo `char` e que a aritmética de inteiros possa ser usada como forma de tratarmos os caracteres, usando os seus códigos ASCII decimais. São, portanto, válidas expressões como as que se apresentam a seguir, nas quais caracteres são convertidos em inteiros e vice-versa:

```
char c1, c2, c3 = '6', c4, c5, c6;
int cx = 62, cy = 12;
//
c1 = (char) (cx + cy) ;
c2 = (char) ((int) c3 + 43);
c3 = (char) ((int) c2 + 21);
c4 = c2;
```

```
c5 = (char) 53;
c6 = (char) ((int) c5 - 20);
```

O tipo `boolean` não é compatível com qualquer um dos outros. A compatibilidade entre tipos numéricos, em particular o *casting*, é um assunto complexo a ser abordado com cuidado sempre que cálculos matemáticos de grande precisão sejam o objectivo principal dos programas desenvolvidos.

2.4.6 COMENTÁRIOS EM JAVA

Os **comentários** são elementos muito importantes, quer como documentação interna quer, se possível, na produção de documentação externa dos programas. Os comentários são em geral usados na clarificação das variáveis de um programa, das transformações que cada função realiza, seus parâmetros de entrada e resultado, explicação de porções de código, etc.

Em JAVA, existe a possibilidade de utilizar três tipos de comentários:

- **Comentário monolinha**: todo o texto colocado desde // até ao fim da linha;
- **Comentário multilinha**: todo o texto entre /* e */;
- **Comentário de documentação**: todo o texto entre /** e */.

Os comentários de documentação são textos de diversos tipos introduzidos pelo programador, que são posteriormente extraídos por ferramentas apropriadas de JAVA (como *javadoc*), e com os quais é produzida de forma automática a maior parte da documentação fundamental dos projectos.

2.4.7 CONJUNTO DE OPERADORES

Na Tabela 2.2, apresentam-se todos os operadores disponíveis em JAVA para os tipos primitivos e não só, indicando-se a sua precedência, a sua sintaxe, o(s) tipo(s) do(s) respectivo(s) operando(s), o sentido da sua associatividade (à direita ou à esquerda) e o seu significado.

Precedência	Operador	Tipos dos Operandos	Associação	Operação
15	`.`	objecto, membro	E	acesso a membro
–	`[]`	*array*, int	E	acesso a elemento
–	*`(args)`*	método, lista args	E	invocação de método
–	`++, --`	variável	E	pós increm/decrem.
14	`++, --`	variável	D	pré increm/decrem.
–	`+, -`	número	D	sinal; unário
–	`~`	inteiro	D	complemento de *bits*
–	`!`	boleano	D	negação
13	`new`	classe, lista args	D	criação de objectos
–	*`(tipo)`*	tipo, qualquer	D	conversão (*casting*)
12	`*, /`	número, número	E	multipl, divisão
–	`/,%`	inteiro, inteiro	E	quociente int, módulo
11	`+, -`	número, número	E	soma e subtracção
–	`+`	string, qualquer	E	concatenação
10	`<<`	inteiro, inteiro	E	deslocam. esq. de *bits*
–	`>>`	inteiro, inteiro	E	deslocam. dir. de *bits*
–	`>>>`	inteiro, inteiro	E	deslocam. dir. com 0
9	`<, <=`	números	E	comparação
–	`>, >=`	aritméticos	E	comparação
–	`instanceof`	referência, tipo	E	teste de tipo
8	`==`	primitivos	E	igual valor
–	`!=`	primitivos	E	valor diferente
–	`==`	objectos	E	mesmo objecto
–	`!=`	objectos	E	objectos diferentes

7	&	inteiros	E	E de *bits*
–	&	boleanos	E	E lógico
6	^	inteiros	E	OUEXC de *bits*
–	^	boleanos	E	OUEXC lógico
5	|	inteiros	E	OU de *bits*
–	|	boleanos	E	OU lógico
4	&&	boleano	E	E condicional
3	||	boleano	E	OU condicional
2	? :	bol, qualq, qualq	D	condicional ternário
1	=	variável, qualquer	D	atribuição
–	*= /= %=	variável, qualquer	D	atribuição após oper.
–	+= -=	variável, qualquer	D	atribuição após oper.
–	<<= >>=	variável, qualquer	D	atribuição após oper.
–	>>>= &=	variável, qualquer	D	atribuição após oper.
–	^= |=	variável, qualquer	D	atribuição após oper.

Tabela 2.2 – Operadores de JAVA

Os operadores aritméticos são os usuais operadores **+ - * /** para soma, subtracção, multiplicação e divisão de números, respectivamente. A divisão (/) de inteiros dá como resultado um número inteiro. O operador **%** (módulo) apenas se aplica a inteiros e dá como resultado o resto da divisão inteira dos operandos (ex.: 15/6 = 2 e 15%6 = 3).

O operador de atribuição =, na sua forma geral `variável = expressão`, corresponde à operação que calcula o valor da expressão do lado direito do sinal de igual e, caso esta termine com sucesso, atribui esse valor à variável do lado esquerdo, sendo perdido o valor anterior nesta contido antes de tal atribuição. Uma operação de atribuição apenas poderá ser realizada se os tipos da variável e do valor da expressão forem compatíveis:

```
x = 2 * y + 10;
z = 4.75 * (w + 5.35);
a = a + 7; // valor de a incrementado de 7 unidades
```

As expressões de atribuição que têm a mesma variável dos dois lados do sinal de =, porque uma dada operação foi realizada com o seu valor anterior (à direita) para calcular o novo valor da variável (à esquerda), são muito comuns. Por exemplo, incrementar um contador, como em `conta = conta + 1`, multiplicar uma variável por um valor dado, como em `sal = sal * taxa`, são usuais. Por tal motivo, operadores particulares existem visando simplificar tais expressões e/ou instruções.

Surgem assim operadores que combinam o operador de atribuição com os operadores aritméticos (e outros), tais como **+= -= *= /= %=**, bem como os operadores unários de incremento e decremento do valor de uma variável como **variável++** e **variável--** e **++variável** e **--variável**, com os significados seguintes:

```
x += 12;  ⇔   x = x + 12;
x %= 5;   ⇔   x = x % 5;
x *= -7;  ⇔   x = x * -7;
```

Os incrementadores e decrementadores podem realizar o incremento ou o decremento antes ou após a utilização do valor da variável, pelo que a sua verdadeira semântica depende da forma como forem utilizados. Num contexto simples e normal, `x++` será sempre equivalente a `x = x + 1`, tal como `++x` (menos usado).

Os operadores que relacionam quantidades (relacionais) são o operador de igualdade == (não confundir com o operador de atribuição =), desigualdade ou diferença !=, e os usuais **> >= < <=**. Quando duas quantidades são comparadas, o resultado dessa expressão será sempre um valor boleano, ou seja, `true` ou `false`.

```
x * 3 >= y - 5
nota > MAX_NOTA
valor != 20
encontrado == true
```

Os operadores relacionais aparecem muitas vezes em expressões mais complexas ligados aos operadores lógicos que funcionam sobre dois operandos de tipo booleano. Em JAVA temos o operador de conjunção (E-lógico) &&, o operador de disjunção (OU-lógico) || e o operador de negação !. Em conjunto com os relacionais, já nos permitem escrever testes mais complexos (mesmo que sem muito sentido), como:

```
(x + 2 > y - 5) && (y + 4) < (z - 3)
!existe && conta <= MAX_ELEM
area > 2*PI*raio || perímetro <= 345.7 - (lado/2.76)
```

Os operadores >> e >>> e <<, bem como os operadores &, |, ^ e ~ designam-se por *bitwise*, pois trabalham com os tipos `short` e `int` a nível dos respectivos *bits*. Os operadores >> e << realizam *shifts* de dado número de *bits*. Os outros correspondem às operações de E, OU, XOR e NOT de binários. Apenas um pequeno exemplo, já que ninguém vai dividir, multiplicar ou fazer aritmética desta forma:

```
int z = 100;
int r = z<<1;  // deslocar 1 bit para a esq. <=> z*2
int s = z>>2;  // deslocar 2 bits para a dir. <=> z/4
```

Um operador muito simples, e que por vezes é muito útil, é o operador ternário condicional **? :** que tem a forma geral `exp1 ? exp2 : exp3`. Esta expressão tem um valor final que será sempre ou o valor de `exp2` ou o valor de `exp3`, tal dependendo de `exp1` ao ser avaliada ter o valor `true` ou `false`. São úteis, por exemplo, em atribuições condicionais, nas quais o valor a atribuir depende do valor lógico da condição expressa:

```
raio = r <= 0 ? 1.0 : r;
maior = (x > y) ? x : y;    // parêntesis ajudam na leitura
```

Para além dos operadores aritméticos, JAVA oferece um grande conjunto de funções matemáticas de elevada precisão que podem ser encontradas na biblioteca (classe) designada por `java.lang.Math`. A classe poderá ser importada para um programa usando uma cláusula de importação `import java.lang.Math.*;`, sendo então possível invocar as suas funções, que, em geral, trabalham com valores de tipo `double`.

Vejamos alguns exemplos da sua utilização em expressões matemáticas:

```
double x = Math.sqrt(y);        // raiz quadrada
double z = Math.pow(x, 2.0);    // potência
double a = Math.sin(alfa);      // seno de alfa
double num = Math.random();     // aleatório entre 0.0 e 1.0
int aleat = 1 + (int) (Math.random() * 10);  // [ 1.. 10]
double at = Math.atan(x);       // arco tangente x
```

2.4.8 INSTRUÇÕES, ESTRUTURAS DE CONTROLO E DIRECTIVAS

O controlo mais básico da execução dos programas é em JAVA, como não podia deixar de ser, expresso sob a forma de **instruções** e **estruturas de controlo** tradicionais. Em conjunto com algumas directivas especiais de criação de contexto, poderemos definir toda a estrutura de execução dos programas.

As principais estruturas de controlo e directivas são as seguintes:

- **Condicionais**: *if/else, switch;*
- **Repetitivas**: *for, while, do/while, for(each*) para *arrays*
- **De sincronização**: *synchronized;*
- **De tratamento de erros**: *try/catch/finally;*
- **Directivas de contexto**: *import, import static, package.*

Dada a utilização massiva que mais adiante faremos de cada uma destas construções, deixaremos aqui registadas apenas, e de forma simples, a sintaxe e semântica básica de cada uma à custa de pequenos exemplos. Note-se que, dado não termos ainda definido rigorosamente o que são objectos, mas apenas os tipos primitivos, os exemplos de utilização das estruturas de controlo de momento compreensíveis, terão, por tal motivo, que ser apresentados tendo por base apenas a manipulação de valores associados a variáveis de tipos primitivos. Porém, estas mesmas estruturas de controlo, tal como aqui apresentadas, serão

posteriormente componentes fundamentais na definição do comportamento dos ***objectos*** que pretendermos definir, designadamente através do código dos respectivos ***métodos***, que são as entidades internamente responsáveis pelo adequado tratamento das ***mensagens*** recebidas.

IF/ELSE

A forma sintáctica mais simples assume a seguinte forma:

```
if (condição) instrução; // se condição então . . .
```

Se a `condição` (expressão boleana entre parêntesis) for verdadeira a `instrução` será executada, caso contrário, não:

```
if (contador < MAX) valor = valor + delta;
if (x + 5 >= j + h) h = x + 20;
```

Porém, quando a condição é verdadeira, pretendemos muitas das vezes realizar mais do que apenas uma instrução. Em tais casos, ou seja, sempre que tivermos que realizar mais de uma instrução, devemos usar um **bloco**. Um **bloco de instruções** é uma sequência de instruções separadas por **;** e delimitadas por chavetas **{ … }** onde se pode, inclusivamente, declarar variáveis locais ao bloco. Temos então a seguinte forma sintáctica:

```
if (condição) { instrução1; … ; instruçãon; }
```

e alguns exemplos desta forma, com indentações diferentes para as chavetas:

```
if (!encontrado) { x = x + 10; encontrado = (x > 20000); }

if (conta < 1000) {
     x ++; y++; z = x*2; w = y*2;
}
```

Na sua forma mais geral, a estrutura condicional é uma alternativa, ou seja, em função do valor da condição, será executada uma ou outra instrução, conforme a seguinte forma:

```
if (condição) instrução1; else instrução2;
```

Um exemplo desta última forma de expressão seria:

```
if (x >= 12)
     y = x * x;
else
     y = x + 100;
```

ou de forma mais simples

```
if (x>= 12) y = x * x; else y = x + 100;
```

Se se tratar de uma alternativa entre blocos de instruções, a sua forma geral será

```
if (condição)
   { bloco1 }
else
   { bloco2 }
```

e são exemplos desta última forma, usando vários estilos para indentação das chavetas:

```
if (tipo == 'X') {
      conta = conta + 1; x = x + 1;
}
else {
      naoLivro = naoLivro + 1; consulta = consulta + 1;
}
```

```
if (conta < MAX)
    {  valor = valor*150; multa = MAX/2;
    }
 else
    { valor = valor*500; multa = MAX/3;
    }
```

ou ainda, caso os blocos ocupem apenas uma linha,

```
if (tipo == ´L´)
   { livros = livros + 1; req = req + 1; }
else
   { naoLivro += 1; consulta += 1; }
```

Um caso particular de utilização das estruturas condicionais **if/else** por vezes um pouco complexo, é quando aparecem *if/else* dentro de outros *if/else*. Nestes casos, e qualquer que seja a indentação, a regra é que um *else* se liga sempre ao `if` mais próximo.

SWITCH

A instrução condicional **`switch`** é a estrutura de controlo condicional que possibilita a execução de uma dentre várias porções alternativas de código, em função do valor tomado por uma particular expressão ou variável. Trata-se, portanto, de uma estrutura *case*, tendo a seguinte forma sintáctica genérica:

```
switch (expressão) {
    case valor_1 : instruções ; break;
    ……
    case valor_n : instruções ; break;
    default : instruções ;
}
```

Assim, em função do valor concreto da expressão indicada – que deverá ser de um tipo simples, `char`, `int`, `short`, `long` ou `byte` – e ainda dependendo dos diferentes valores particulares que se pretende distinguir, são executadas as instruções associadas a cada um de tais valores. A instrução `break` é utilizada para se garantir que a execução do código da instrução `switch` termina aí. Caso não seja colocada a instrução `break`, após a execução do conjunto de instruções correspondentes a um valor particular, as instruções associadas ao valor seguinte serão executadas também, e assim sucessivamente, até ao aparecimento de um `break`. Caso o valor da expressão não seja igual a nenhuma das constantes associadas a cada caso, será executado o conjunto de instruções associado à palavra chave `default`, caso exista, ou nenhuma, no caso contrário.

Os exemplos seguintes procuram ilustrar a utilização de casos por omissão e diferentes formas de especificar a execução de sequências de instruções:

```
switch (letra) {
 case 'a' :
 case 'A' : { i = (int) letra; ct1++ ; break; }     // bloco
 case 'b' :       // não executa nada
 case 'B' : x-- ; break;
 default: ct2++;  // para todos os outros valores
}
```

O `switch` seguinte serve para determinar o número de dias de um determinado mês, sendo o valor da variável `mes` que determina a decisão:

```
switch (mes) {
 case 2: dias = 28; break;
 case 4:
 case 6:
 case 9:
 case 11 : dias = 30; break;
 default: dias = 31; }
```

FOR

A estrutura repetitiva **for** assume uma forma sintáctica que é, de certo modo, complexa, dado o conjunto de expressões que envolve, e as variantes possíveis para cada uma delas. A estrutura genérica assume a forma seguinte:

```
for (inicialização; condição_de_saída; iteração) instruções
```

O grupo designado `inicialização` contém em geral a declaração e inicialização da variável que irá servir como variável de controlo do ciclo. A expressão designada como `condição_de_saída` tem exactamente a missão de definir uma expressão boleana que será avaliada em cada iteração, e que determinará o fim das iterações logo que o seu valor seja `false`. Note-se que, caso esta expressão tenha o valor `false` logo no início, ou seja, antes da primeira execução, nada será executado. O grupo designado por `iteração` contém as declarações que indicam de que forma os valores de certas variáveis evoluem de uma iteração para a outra, em particular da variável de controlo do ciclo (uma ou mais).

A variável de controlo do ciclo pode ser declarada dentro da estrutura `for` e apenas existe dentro desta, não devendo ser modificada no bloco de instruções do ciclo. Vejamos alguns exemplos.

```
// soma e produto dos 100 primeiros inteiros
int soma = 0;
int produto = 1;
for (int i = 1; i <= 100; i++) {
   soma = soma + i;
   produto = produto * i;
}

// soma dos primeiros ímpares e dos primeiros pares até MAXI;
// (admite-se que MAXI foi lido anteriormente)
int somai = 0, somap = 0;
for (i = 1; i <= MAXI ; i++) {
  if (i%2 == 0)
     somap = somap + i;
  else
     somai = somai + i;
}
```

WHILE

A estrutura repetitiva **while** tem a seguinte forma sintáctica:

```
while (condição_de_iteração) {
  corpo ;
}
```

sendo a sua semântica muito simples: enquanto a `condição_de_iteração` for verdadeira o corpo do ciclo é executado. A `condição_de_iteração` pode ser ou não verdadeira no início da iteração. Se for falsa, nada será executado e o programa continuará na instrução seguinte. Se for verdadeira, a primeira iteração é realizada, isto é, todo o corpo definido é executado, sendo em seguida de novo avaliada a `condição_de_iteração`. Naturalmente que se nenhuma das variáveis usadas para exprimir esta condição for modificada ao ser executado o corpo do ciclo, então teremos uma situação de ciclo infinito.

Note-se que sendo a `condição_de_iteração` testada antes da primeira execução do corpo, pode acontecer que, sendo a primeira imediatamente falsa, este não seja executado. Assim, a principal característica determinante na utilização de uma estrutura repetitiva do tipo **while** é que a mesma conduz a 0 ou mais iterações do corpo de instruções a iterar. O número de iterações que serão efectivamente realizadas não pode nestes casos ser expresso apenas em função dos sucessivos valores tomados por uma variável de controlo. Quando o número de iterações pode de facto ser expresso de tal forma, a estrutura aconselhada é uma estrutura repetitiva do tipo **for**. É, no entanto, curioso verificar que os programadores, por erro conceptual ou simples hábito que não se aconselha que seja reproduzido, implementam estruturas *while* através de

estruturas *for*, quer usando `break`, quer através de condições adicionais de saída na expressão de `condição_de_saída` do ciclo *for*.

Aconselha-se vivamente que, sempre que pretendermos expressar uma iteração que pode ocorrer 0 ou mais vezes, se expresse tal facto usando uma estrutura **`while`**. Sempre que pretendermos expressar uma iteração que ocorre tantas vezes quantos os valores que uma dada variável deve tomar para que, qualquer que seja o incremento, partindo de um dado valor atinja um outro, então, tal deve ser codificado usando uma estrutura **`for`**.

Apesar de tudo, e tendo em atenção a sua semântica, poder-se-á dizer que as duas construções seguintes são equivalentes:

```
for (expressão1 ; expressão2 ; expressão3) instruções;

// while equivalente ao for anterior
expressão1;
while (expressão2) {
 instruções;
 expressão3;
}
```

Vejamos agora alguns exemplos de aplicação da estrutura repetitiva *while*:

```
// divisão por subtracções sucessivas
double x = 2137.56
double y = 2.5645;
int conta = 0;
//
while (x >= 0) {
   x -= s; conta++;
}

// Tentativas para acertar num número de 1 a 9
// Math é biblioteca de Java e random() uma função;
boolean found = false;
int x = 1; int chave = 1 + (int) (Math.random() *9);
while (!found) {
   if ( x == chave ) found = true;
   else x++;
}
```

DO/WHILE

A estrutura repetitiva **`do-while`** deve ser usada sempre que tivermos um bloco de instruções que vai ser executado uma ou mais vezes, ou seja, contrariamente ao que se passa com a estrutura repetitiva *while*, tal código será sempre executado pelo menos uma vez. No final desta primeira execução, e de todas as outras, uma condição de continuação de iteração é testada, e se o seu valor for `false` a iteração termina.

A forma sintáctica desta estrutura repetitiva é:

```
do { corpo; } while(condição_de_iteração);
```

O código seguinte exemplifica a sua utilização:

```
// realiza o somatório e o produtório dos
// números inteiros desde 0 a MAX_VEZES
int conta = 0; int soma = 0; int prod = 1;
do {
     soma = soma + conta;
     prod = prod * conta;
     conta++;
}
```

```
while(conta < MAX_VEZES);
// Resultados
System.out.println("Somatório = " + soma);
System.out.println("Produtório = " + prod);
```

2.4.9 *STRINGS*

As *strings* são sequências de caracteres, representáveis textualmente delimitadas por **" "**, como por exemplo, "eu sou uma *string*!". Não existe um tipo base `string` em JAVA, antes uma classe designada **`String`**, cujo nome funciona como tipo. Assim, variáveis que vão conter sequências de caracteres devem ser declaradas como sendo do tipo `String`, e até imediatamente inicializadas como quaisquer outras de tipos primitivos:

```
String titulo = ""; // string vazia ou nula
String lp = "JAVA5";
```

Podemos realizar várias operações com as variáveis de tipo `String`, sendo uma das mais comuns a operação oferecida pelo operador **+** que é o operador de concatenação (junção) de *strings*:

```
String nome = "Ze";
String apelido = "Boavida";
String completo = nome + " " + apelido; // "Ze Boavida"
```

O operador de concatenação (+) é um operador capaz de compatibilizar tipos primitivos com *strings* durante a concatenação, convertendo qualquer valor de tipo simples (mesmo os boleanos) para *string* e realizando a sua junção. São portanto válidas as seguintes expressões:

```
int idade1 = 20; int idade2 = 22;
String frase1 = "O Pedro tem " + idade1 + " anos";
String frase2 = "A Rita tem " + idade2 + " anos";
String decisao = idade2>idade1 ? "velha" : "nova";
String frase3 = "Rita é mais " + decisao + " que Pedro.\n";
```

Muitas são as operações que podem ser realizadas sobre *strings*. Porém, *strings* não são valores, são objectos, pelo que voltaremos a referir as *strings* depois de introduzirmos as noções relacionadas com os tipos referenciados aos quais as *strings* pertencem.

2.4.10 I/O SIMPLES

Precisamos agora de conhecer algumas instruções simples de escrita e leitura para podermos criar alguns pequenos programas que nos permitam experimentar este nível de programação com tipos primitivos.

As instruções básicas de escrita (*output*) de JAVA são dirigidas ao dispositivo básico de saída, o monitor, que a partir do programa é visto como um "ficheiro" de caracteres do sistema designado por `System.out`. Assim, são mensagens básicas todas as *strings* enviadas para tal ficheiro usando a instrução `println(.)`, o que se escreve:

```
System.out.println("Bom dia e bom trabalho!\n\n\n");
System.out.println("Linguagem : " + lp);
System.out.println(nome + ", eu queria " + apelido + "!");
System.out.println("Rita é mais " + decisao + " que Pedro");
```

Se nos programas em que tivermos que realizar muitas operações de saída escrevermos no seu início a cláusula de importação `import static java.lang.System.out;`, então, em vez de se escrever `System.out.println(.)` bastará escrever-se `out.println(.)`.

SAÍDA FORMATADA: `printf()`

Uma instrução `printf(stringf, lista_valores)` permitirá a saída de uma lista de valores (sob a forma de variáveis, constantes ou expressões), que serão formatados segundo as directivas dadas na **string de formatação** que é o primeiro parâmetro.

Esta *string* é bastante complexa, pois em JAVA tem que especificar as conversões, quer para tipos primitivos quer para objectos. A forma geral de formatação de valores de tipos primitivos é a seguinte:

```
%[índice_arg$] [flags] [dimensão][.decimais] conversão
```

Começando pelo fim, os **caracteres de conversão** são os que indicam o tipo de valor resultado da conversão do parâmetro, sendo: **c** (carácter), **b** (boleano), **o** (octal), **h** (hexadecimal), **d** (inteiro), **f** (real, vírgula fixa), **s** (*string*), **e** (real, vírgula flutuante) e **n** (*newline* independente da plataforma).

Por exemplo, usando apenas caracteres de conversão, podemos automaticamente fazer a conversão de um número inteiro na base 10 para a base 8 e para a base 16.

```
int x = 1261;
out.printf("Inteiro %d = Octal %o = Hexa %h%n", x,x,x);
```

Note-se desde já a possibilidade de inserirmos caracteres na forma '\n' na *string* que define os formatos. Note-se também, através deste exemplo, o interesse de podermos, em dadas circunstâncias, indicar qual o argumento a que nos estamos a referir na *string* de formatação (cf. [*índice_arg$*]). No exemplo acima tivemos que repetir a variável **x** três vezes para preencher os três conversores, mas basta dizer que cada um deles deve usar o primeiro e único parâmetro, como por exemplo:

```
out.printf("Inteiro %d = Octal %1$o = Hexa %1$h%n", x);
```

e, num caso mais geral, a ordem dos parâmetros poderá ser completamente reformulada na *string* de formatação, tal como no exemplo seguinte já com os elementos definidores das dimensões e de casas decimais.

```
float f1 = 123.45f;
double d2 = 234.678; double d3 = 12.45E-10;
out.printf("R1 %5.2f R2 %3$-17.16f Exp1 %2$8.4e%n",f1, d2, d3);
```

Finalmente, as *flags* podem permitir alinhar os resultados à esquerda (-), obrigar a incluir sempre o sinal (+), colocar zeros no início (0), colocar espaços no início (' ') ou colocar parêntesis (() se o número for negativo.

Um outro método, de nome `format(stringf, lista_valores)`, pode ser usado da mesma forma que `printf()`. Uma classe especial de nome `Formatter` oferece inúmeros métodos auxiliares para formatações complexas.

2.4.11 TIPOS REFERENCIADOS: OBJECTOS E *ARRAYS*

Em JAVA, os tipos não-primitivos são associados a objectos e a *arrays* e designam-se por **tipos referenciados**, dado que os seus representados são tratados por **referência**, ou seja, através de uma variável que, **contendo o seu endereço**, permite o acesso ao seu valor efectivo. As variáveis de tipos primitivos, como acabámos de ver, guardam valores e não endereços de valores.

Assim, as variáveis de tipos referenciados servem para conter os **endereços dos objectos** ou dos *arrays* de um dado tipo, definido numa declaração. Tal declaração indicará que tal variável poderá referenciar **instâncias** de uma dada **classe**, ou que uma variável referencia *arrays* com uma dada estrutura, dimensão e tipo de elementos.

Falemos, para já, de **instâncias de classes**. Por exemplo, a seguinte declaração:

```
Ponto ponto1;
```

indica que a variável identificada por `ponto1` é do tipo `Ponto`, e que, portanto, poderá referenciar instâncias da classe `Ponto`. **Classes são, portanto, tipos**.

Tal como para as variáveis de tipo primitivo, o valor inicial da variável `ponto1`, após esta declaração, depende do contexto, dentro do programa JAVA, em que tal declaração foi realizada. No exemplo, e admitindo que se trata da declaração de uma variável local, e não de uma **variável de instância** de uma dada

classe, a variável, após a declaração, terá um valor indefinido, pelo que, caso haja a tentativa da sua utilização sem que entretanto lhe tenha sido atribuído um valor, o compilador indicará um erro de compilação.

Caso tal declaração tivesse sido produzida no âmbito das definições das variáveis de instância de uma dada classe, também designadas por **atributos** ou **campos**, e aqui reside mais uma vez a excepção à regra geral de não inicialização de JAVA, o seu valor inicial, dado ser esta uma variável de um tipo referenciado, seria, por isso, automaticamente o valor **`null`**, valor de inicialização por omissão em JAVA para este tipo de variáveis.

Como associar agora à variável `ponto1`, do tipo referenciado `Ponto`, uma instância de tal classe, ou seja, como inicializar de facto tal variável? Ou ainda, como poderemos criar instâncias da classe `Ponto` e referenciá-las através de variáveis declaradas como sendo referências de instâncias dessa classe?

Como veremos posteriormente, quando definirmos uma dada classe de JAVA, poderemos definir um conjunto muito diversificado de métodos que servirão para criar instâncias dessa classe, e que, eventualmente, produzem inicializações muito diferentes das variáveis de instância de tais objectos. Tais métodos são, por tal motivo, designados em JAVA como **construtores** de instâncias.

Deixando a questão dos construtores para uma posterior maior elaboração, é importante afirmar desde já que JAVA associa a cada classe criada de nome C um **construtor por omissão** de nome C( ). Quando usado numa dada declaração conjuntamente com a palavra reservada **`new`**, tal como deverão ser usados todos os outros possíveis construtores, este construtor particular (dado não ser explicitamente programado) cria um objecto dessa classe, atribuindo aos campos ou variáveis de instância desses objectos valores por omissão. Caso estes campos sejam de tipos simples, ser-lhes-ão atribuídos os valores por omissão que foram apresentados na tabela 2.1. Caso tais campos sejam de tipos referenciados, assumirão, por omissão, o valor genérico representado pela palavra reservada **`null`**, valor este não manipulável dado indicar uma não referência (ou seja, é um **apontador nulo**).

Assim, e segundo tais regras de JAVA, a declaração,

```
Ponto ponto1 = new Ponto();
```

admitindo, que as duas variáveis de instância que definem a estrutura de um ponto são do tipo simples `int` e se designam por x e y, produziria, neste caso, a seguinte associação entre referência e objecto referenciado (Figura 2.5):

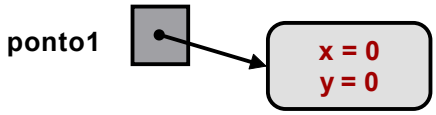

Figura 2.5 – Referência e objecto

Note-se ainda que, sendo estes objectos referenciados através de variáveis, haverá que ter em atenção que atribuições entre estas variáveis que referenciam objectos assumem uma semântica de apontadores, ou seja, o que é atribuído ou manipulado são **referências** (na terminologia de JAVA designam-se por *handlers* e não são directamente manipuláveis).

Assim, se escrevermos `Ponto ponto2 = ponto1;`, `ponto2` passa a referenciar o mesmo objecto referenciado por `ponto1`, qualquer que este seja nesse momento (Figura 2.6).

Esta partilha de objectos resultante de uma atribuição de referências, é uma operação que deve ser realizada com bastante cuidado, pois a partir do momento em que o objecto passa a estar referenciado pelas duas variáveis, qualquer delas lhe tem acesso, e, através de qualquer uma delas, é possível modificar o objecto.

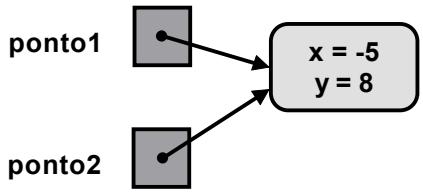

Figura 2.6 – Objecto partilhado por `ponto1` e `ponto2`

Note-se ainda que o resultado do teste de igualdade `ponto1 == ponto2` daria, neste caso, `true`, já que, no momento, ambas as variáveis referenciam exactamente o mesmo objecto. Porém, se tivéssemos a situação ilustrada na Figura 2.7:

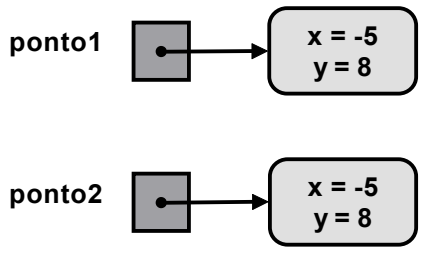

Figura 2.7 – Objectos diferentes mas de igual valor

o mesmo teste resultaria no valor `false`, já que os objectos referenciados, ainda que contendo os mesmos valores, são distintos em **identidade**.

ARRAYS

Os *arrays* são, em JAVA, entidades referenciadas mas não são objectos, dado que não são, ao invés de todos os objectos, criados a partir de uma classe. Porém, por serem manipulados por referência, não são associados a tipos-valor, mas a tipos referenciados.

Arrays são criados em JAVA dinamicamente, ou seja, em tempo de execução, e o seu espaço automaticamente reaproveitado quando deixam de estar referenciados. A criação dinâmica de um *array* faz-se usando igualmente **`new`**, indicando ainda o tipo dos seus componentes, e, opcionalmente, as suas diferentes dimensões **– são multidimensionais**.

Porém, os *arrays* são estáticos, isto é, são de dimensão fixa, não podendo aumentar de dimensão em tempo de execução. A multidimensionalidade dos *arrays* é implementada em JAVA como **aninhamento**, isto é, *arrays* multidimensionais são, de facto, apenas *arrays de arrays*.

Vejamos alguns exemplos de declarações, com e sem inicializações, chamando-se a aten-ção para a forma de definição da capacidade do *array*: `new tipoComp[dimensão]`.

```
int lista[]; // lista é um array de inteiros;
int[] lista; // declaração equivalente à anterior
int[] lista = {10, 12, 33, 23}; // declaração e inicialização
int lista[] = new int[20];      // array de 20 inteiros
byte[] pixels = new byte[600*800];
String[] texto = new String[200]; // array de 200 strings
int[][] tabela = new int[30][20]; // 30 lin x 20 col inteiros
int tabela[][] = new int[30][20];
```

A instrução **`new`**, para além de criar o *array* com a dimensão indicada, inicializa todas as suas posições. Para os tipos primitivos inicializa-as com o seu valor por omissão, e para os tipos referenciados com o valor **null**.

As duas últimas declarações definem o *array* `tabela` como sendo do tipo `int[][]`. Dinamicamente alocam um *array* de 30 elementos de linha, sendo cada um destes elementos do tipo `int[20]`, ou seja, um *array* de

20 inteiros. Alocam depois os 30 *arrays* de 20 inteiros, guardando a referência para cada um destes no *array* de 30 posições.

Este mecanismo de alocação permite que algumas das dimensões possam ser deixadas por especificar em tempo de declaração e compilação, desde que em sequência. Por exemplo, poderíamos escrever a seguinte declaração:

```
int[][] matriz = new int[30][];
```

Neste caso, seria alocado um *array* de 30 posições, tal como o anterior, mas com referências de valor `null`, de momento, para cada um dos elementos a alocar nestas posições, sabendo-se, no entanto, que tais elementos devem ser do tipo `int[]`, ou seja, cada elemento é um *array* de inteiros de dimensão não especificada.

Ao não obrigar à especificação da segunda dimensão, JAVA flexibiliza e permite que os elementos do tipo `int[]` (*array* de inteiros) deste *array* possam ter dimensões distintas. Esta flexibilidade não implica para o programador qualquer tipo de perda de controlo, tal como ter que saber de facto qual a verdadeira dimensão de cada um dos *arrays* alocados, dado ser possível em JAVA saber a dimensão efectiva de um *array* através da construção sintáctica `array.length`, que dá como resultado a sua capacidade (não o número de elementos). O ciclo *foreach* apresentado a seguir torna mesmo irrelevantes tais questões.

Quanto às inicializações, elas podem ser feitas na altura da declaração do *array*, usando a forma sintáctica que se exemplifica a seguir:

```
int[] potencias2 = {2, 4, 8, 16, 32, 64, 128, 256, 512, 1024};
String[] moedas = { "Escudo", "Peseta", "Franco", "Coroa"};
// inicializador com aninhamento
int[][] matriz = { {1,2}, {10, 20, 30}, {19, 2, 34, 21, 16} };
String[][] linhas = { {"O", "mi"}, {"e"}, {"o", "si", "#"};
```

ACESSO AOS ELEMENTOS DE UM *ARRAY*

O acesso aos elementos de um *array* faz-se através de um índice inteiro, que pode tomar valores desde 0 (a 1ª posição do *array*) até length-1. A sintaxe para acesso é a que existe praticamente em todas as linguagens de programação, ou seja**, id[exp_indice]**. Exemplos:

```
x = lista[2*3];
nome = nomes[i];
nota = notas[ano][discp];
nomes[i] = "Rui";
notas[2][12] = 19;
```

No caso dos *arrays* multidimensionais, o primeiro índice corresponde à linha, e o segundo corresponde à coluna dentro da linha.

CICLO `for` COM *ARRAYS*

A maior parte das operações que temos que fazer com elementos de um *array*, sejam estes de tipos primitivos ou objectos, são operações de duas classes: **varrimentos** e **pesquisas**. As operações de pesquisa, porque possuem condições lógicas especiais para paragem, são realizadas usando **`while`** ou **`do-while`**. As de "varrimento" têm por objectivo percorrer todos os elementos do *array* e com cada um deles realizar certas operações.

Sendo os *arrays* indexados, os varrimentos fazem-se usando os índices de 0 até length-1, valores contidos numa variável de controlo que os vai tomando um a um, e que pode ser a variável de controlo de um ciclo `for`, como em:

```
// contagem de pares e ímpares num arrays de inteiros
int par = 0, impar = 0;
for(int i = 0; i < a.lenght; i++)
   if (a[i]%2 == 0) par++; else impar++;
```

Quando o *array* é multidimensional, a estratégia é fazer o aninhamento de vários ciclos `for`, tantos quantas as dimensões a tratar. No exemplo, o ciclo mais exterior percorre as linhas e o interior as colunas. Para cada linha todas as suas colunas são acedidas.

```
// inteiros > MAX de um array de arrays de inteiros
int maiores = 0; int MAX = -999999;
for(int l = 0; l < nums.length; l++) {
   for(int c = 0; c < nums[l].length; c++)
      if (nums[l][c] > MAX) maiores++;
}
```

`for`*(each)* COM *ARRAYS*

JAVA5 introduziu um ciclo **`for`** particular que funciona para colecções de objectos. Os *arrays* de JAVA permitem coleccionar quer valores quer objectos, pelo que têm este ciclo `for` disponível. A forma geral é muito simples:

```
for (IdTipo elem : IdArray) { instruções }
```

lendo-se, "para cada elemento `elem` de tipo `IdTipo` obtido do *array* `IdArray` fazer ..", sendo `IdTipo` o tipo dos elementos do *array*. Este tipo de ciclo é, em algumas linguagens, designado por ciclo *foreach*, exactamente por iterar usando "cada um" dos elementos da colecção. Em muitos casos, esta nova forma de expressar a iteração sobre *arrays* poderá tornar a escrita de código mais simples e mais legível. Vejamos dois exemplos:

```
int[] a = {1, 34, -2, 5, -12, 34, 7};
// cálculo da soma 1
int soma = 0;
for(int i = 0; i < a.length; i++) { soma += a[i]; }
// cálculo da soma 2
soma = 0;
for(int x : a) { soma += x; }
```

Agora, a concatenação numa única *string* de um *array* contendo *arrays* de *strings*:

```
String[][] nomes = { {"Rita", "Pedro"}, …… };
String sfinal = "";
// concatenação 1
for(int l = 0; l < nomes.length; l++) {
    for(int c = 0; c < nomes[l].length; c++)
      sfinal += nomes[l][c];
}
sfinal = "";
// concatenação 2
for(String[] lnomes : nomes) {
    for(String nome : lnomes)  sfinal += nome;
}
```

Como podemos verificar, a solução com o ciclo `for`*(each)* torna-se muito mais simples para este caso, porque um maior número de detalhes de controlo da iteração é escondido. De salientar, aliás, que o primeiro algoritmo é a implementação da segunda forma.

Para concluir esta breve apresentação sobre os *arrays* em JAVA, duas notas importantes. Existe uma classe `java.util.Arrays` que oferece várias operações sobre *arrays* que lhes sejam passados como parâmetros, tais como ordenação, pesquisa binária, conversão para lista, etc. Na classe `System`, uma operação `arraycopy()` permite copiar elementos de um *array* para outro de forma muito eficiente.

Os *arrays* não são objectos, logo, não são importantes em PPO, até porque são estáticos (não crescem) e há classes que são "*arrays* dinâmicos". Porém, é com base nos *arrays* que estão construídas as classes de JAVA que implementam as várias colecções de objectos (estruturas de dados e algoritmos), dada a sua eficiência (memória contígua).

2.4.12 ESTRUTURA DE UM PROGRAMA

Um programa fonte JAVA tem uma estrutura, à partida, muito simples, porque em cada ficheiro de texto fonte JAVA apenas é possível definir uma classe, e o nome do ficheiro deve ser igual ao nome da classe, tendo como extensão *java*. Assim, o ficheiro contendo a definição de uma classe C dever-se-á chamar C.java.

A estrutura de um **programa principal** (programa executável) de JAVA é a apresentada na Figura 2.8:

```
import java..............;
...........

public class ProgPrincipal {
  ..............

  public static void main(String[] args) {
  ------------------------------------
  ------------------------------------
  ------------------------------------
  }

}
```

Figura 2.8 – Estrutura base de programa JAVA

Assim, o que noutras linguagens se designa por **programa principal** em JAVA é uma classe declarada como `public class` seguida do nome, neste caso, `ProgPrincipal`, na qual definimos uma função chamada `main()`, declarada como `public static void` e com um parâmetro `args` que é do tipo `String[]`, ou seja, um *array* de *strings*. Ambas as declarações são seguidas de blocos `{. .}` de que falaremos em seguida.

O que acabámos de escrever é absolutamente fixo e, sempre que tivermos que escrever um pequeno programa em JAVA, podemos copiar este formato padrão. No entanto, esta é uma classe especial pois não vai, nem foi definida para tal, criar instâncias. Serve apenas para conter um método `main()` no qual vamos escrever código para ser executado.

Tal como noutras linguagens, sempre que necessitarmos de usar bibliotecas da linguagem no nosso programa, antes de iniciarmos a declaração da classe, escrevemos as cláusulas de importação de tais bibliotecas (vimos anteriormente algumas úteis).

A função `main()` será a primeira a ser invocada pelo interpretador de JAVA ao ser executado o programa. Caso esta função não existisse, o programa não executaria. Assim, o código que pretendemos executar neste nosso programa deve ser introduzido no bloco associado à função `main()`.

Outras funções externas a esta função poderiam ser codificadas no corpo principal da classe usando `public static`, podendo a função `main()` invocá-las de forma a estruturar o seu código.

Apresentam-se em seguida quatro pequenos exemplos:

```
import static java.lang.System.out;
public class Prog1 {
   // cálculo de uma área circular
   public static void main(String[] args) {
      double raio = 2.56;
      double pi = 3.14159;
      out.printf("Circulo de raio %5.2f%n", raio);
      out.printf("tem area igual a %10.4f%n", raio*pi);
   }
}
```

Os resultados produzidos em ecrã são:

```
Circulo de raio  2,56
tem area igual a     8,0425
```

No exemplo, declarámos na função `main()` duas variáveis do tipo `double` e, em seguida, foi feita a saída formatada dos textos com os valores do raio e do cálculo da área do círculo. Os resultados da execução do programa são os apresentados.

É portanto no bloco de instruções de `main()` que declaramos todas as variáveis com que vamos trabalhar no âmbito da função, e escrevemos as instruções para a realização do problema a resolver. O programa seguinte declara um *array* de inteiros, atribui-lhe valores e determina qual o menor valor e a sua posição no *array*, escrevendo-os no ecrã.

```
import static java.lang.System.out;
public class TstArrays1 {
   // programa principal para testes com arrays
   public static void main(String[] args) {
      // declara e inicializa os elementos de um array
      int[] lista = {128, -2, 245, 616, 77, 213};
      // determinação do menor elemento do array
      int min = lista[0]; // primeiro é, para já, o menor
      int pos = 0;        // está no índice 0
      for(int i = 1; i < lista.length; i++)
        if (lista[i] < min)  {
            min = lista[i]; pos = i;
        }
      out.println("\nMenor = " + min);
      out.println("Indice = " + pos);
  }
}
```

Os resultados produzidos em ecrã são:

```
Menor = -2
Indice = 1
```

O programa seguinte é interessante pelo facto de que, para além de usar um algoritmo bem conhecido, o algoritmo de pesquisa binária num conjunto ordenado de inteiros, tem a vantagem de necessitar de estruturas não usadas nos exemplos anteriores.

```
import static java.lang.System.out;
public class TstArrays2 {
   // ALGORITMO DE PESQUISA BINARIA
   // (arrays não vazios e por ordem crescente)
   public static void main(String[] args) {
      // declara e inicializa os elementos de um array
      // usando a construção de inicialização {}
      int[] a = {0, 4, 8, 9, 21, 34, 45, 57, 51, 61, 70, 90, 93};
      int chave = 70; // valor usualmente lido !
      int menor, meio, maior;
      boolean existe = false;
      menor = 0; maior = a.length - 1;
```

```
      // procurar sempre o índice médio destes indices
      do {
        meio = (menor + maior)/2;
        if ( chave > a[meio] )
          menor = meio + 1;
        else
          maior = meio - 1;
      }
      while( (a[meio] != chave) && (menor <= maior) );
      // encontrado ou não ??
      existe = (a[meio] == chave);
      if (!existe )
        out.println("N. " + chave + " nao existe !");
      else
        out.println("N. " + chave + " no índice " + meio);
   }
}
```

Os resultados produzidos em ecrã são: `N. 70 no índice 10`

Repare-se, neste algoritmo, que a condição de saída do ciclo *do-while* assume já uma expressão boleana de conjunção (&&) com dois operandos resultantes de duas expressões de tipo relacional de alguma complexidade. Será que os parêntesis nas expressões de tipo relacional seriam mesmo necessários? De facto não eram, porque o operador lógico tem menor precedência do que os relacionais, mas também é verdade que desta forma o código se lê com muito maior facilidade e a legibilidade é sempre muito importante.

Finalmente, repare-se no alinhamento das chavetas de início e fim de bloco. Depois de definido um estilo, a coerência na sua utilização é também factor a favor da legibilidade.

Enquanto a saída de resultados tem em JAVA múltiplas soluções, a entrada de dados, ou seja, as operações de leitura são relativamente complexas caso não sejam "escondidas" (convenientemente abstraídas) a quem as pretende usar.

Por agora, deixamos apenas um pequeno exemplo da utilização da classe `Scanner`, que lê do teclado valores de tipos primitivos e *strings*:

```
import java.util.Scanner;
public class Leitura1 {
   public static void main(String[] args) {
      // Scanner: classe para leitura
      Scanner input = new Scanner(System.in); // lê via teclado
      System.out.print("Nome: ");
      String nome = input.next();
      System.out.print("Idade: ");
      int idade = input.nextInt();
      System.out.println(nome + " tem " + idade + " anos.");
   }
}
```

Neste programa foi utilizada a cláusula de importação que nos permite ter acesso à classe `Scanner` da biblioteca `java.util`. Uma instância da classe `Scanner` foi criada para trabalhar sobre o "ficheiro" predefinido `System.in` que está associado ao teclado.

A variável `input` refere o "*scanne*r" com que vamos trabalhar a partir de agora, e do qual vamos ler o próximo *token* usando o método `next()` (lê uma sequência de caracteres terminada por *newline*) e, em seguida, um valor inteiro com `nextInt()`. Estes métodos esperam valores correctos, pelo que a leitura de um inteiro incorrecto gerará um erro de execução e aterminação imediata do programa.

Neste caso, deu mesmo muito jeito uma propriedade de JAVA que é a possibilidade de declararmos as variáveis onde quer que seja antes da sua utilização, dado que assim não necessitámos de inicializar as variáveis `nome` e `idade` pois elas recebem de imediato os valores lidos.

Para completar estes exemplos que visam agrupar conjuntos de instruções apresentadas em separado, apresenta-se um excerto de código que permite apresentar ao utilizador, de forma formatada, os vários componentes da data actual do sistema:

```
Calendar hoje = Calendar.getInstance();  // dia + hora/min/seg/miliseg actual
out.printf("%tT", hoje);                    // 07:53:02
out.printf("%1$tY-%1$tm-%1$td%n", hoje);    // 2006-01-01
out.printf("%1$tH-%1$<tM-%1$<tS%n", hoje); // 07-53-02
```

As várias formas `%tY, %tm, %td, %tM, %tS`, etc., correspondem aos formatos dos respectivos componentes do objecto calendário (`Calendar`) que foi criado, cf. ano, mês, dia, hora, min, etc.

2.5 CRIAÇÃO DO AMBIENTE JAVA

O J2SE Development Kit 5.0 (JDK5.0) é o ambiente de desenvolvimento de aplicações JAVA mais recentemente disponibilizado pela Sun Microsystems, podendo ser obtido em http://java.sun.com/j2se/1.5.0/download.jsp ou http://java.sun.com/downloads/index.html ou http://java.sun.com/j2se/. Todo o ambiente de desenvolvimento, bem como toda a documentação fundamental para a instalação em cada tipo de sistema operativo, pode ser obtida a partir de um destes endereços.

O ficheiro http://java.sun.com/j2se/1.5.0/docs/relnotes/version-5.0.html é de leitura muito importante, pois documenta a estrutura de directórios após a instalação, bem como os que são de inclusão obrigatória nas variáveis de ambiente.

Destas variáveis de ambiente, é importante configurar a variável PATH por forma a que passe a conter o caminho para o compilador e outros utilitários de JAVA, por exemplo, o caminho C:\Programas\Java\jdk1.5.0_06\bin, e a variável CLASSPATH, que dá indicação dos locais onde o compilador pode encontar classes necessárias à compilação, e o interpretador à execução. Por omissão é usada a directoria actual de trabalho.