3 - Classes e Instâncias

3.1 Definição de classe

Uma **classe** é um objecto especial que serve de *molde* ou *padrão* para a criação de objectos designados por **instâncias** da classe, objectos similares, já que possuem a mesma estrutura interna, ou seja, as mesmas **variáveis de instância**, o mesmo comportamento, isto é, os mesmos **métodos de instância**, e a mesma interface ou API, pois são capazes de responder às mesmas **mensagens**.

Neste sentido, como vimos já, uma classe é um *módulo* onde se especifica quer a estrutura quer o comportamento das instâncias que a partir da mesma pretendemos criar ao longo da execução de um dado programa.

Sendo as instâncias de uma classe um conjunto de objectos que depois de serem criados, e porque possuem a mesma interface, respondem ao mesmo conjunto de mensagens, isto é, são exteriormente utilizáveis de igual forma, então, a classe pode também ser vista como um **tipo**.

As classes das linguagens de PPO podem ser ainda vistas como mecanismos de implemementação de Tipos Abstractos de Dados (TAD), dado que estes, como o seu nome indica, são especificações abstractas de propriedades e não implementações, que as classes podem corporizar.

Finalmente, tal como a Figura 3.1 procura ilustrar, as classes representam ainda um muito importante mecanismo de **partilha de código**, dado que os métodos, que implementam a funcionalidade de todas as suas instâncias, têm o seu código "guardado" num único local — a sua respectiva classe —, contendo as instâncias apenas os "valores" que representam o seu estado interno.

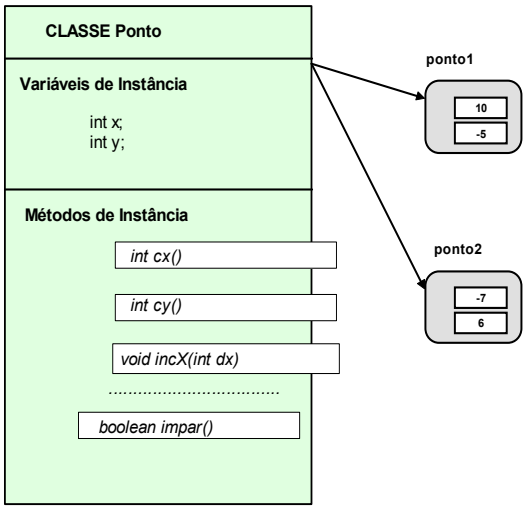

Figura 3.1 – Uma Classe e duas das suas instâncias

De facto, não fazendo qualquer sentido que, por exemplo, todo o código dos métodos de instância de Ponto tivesse que ser replicado em todas as instâncias criadas, o que seria naturalmente uma redundância e um desperdício, já que o código é exactamente igual, então tal código tem uma única cópia da qual a classe é a natural depositária.

Quando uma instância de uma determinada classe recebe uma dada mensagem, apenas terá que solicitar à sua classe a execução do método correspondente, caso tal método exista, execução que será realizada sobre o espaço de variáveis da instância receptora, isto é, usando os valores das variáveis de instância que constituem o estado interno actual desta.

No exemplo apresentado na Figura 3.2, o envio da mensagem *cx()* às duas instâncias da classe Ponto apresentadas geraria resultados diferentes, qualquer que seja o código do método *cx()*, dado que *ponto1* e *ponto2* identificam dois pontos que, tal como podemos verificar pelo exemplo, possuem diferentes valores para as coordenadas em x em y.

Dado que, ao escrever o código do método *cx()*, o programador sabe que qualquer instância de Ponto possui variáveis de instância designadas por *x* e *y*, tal como especificado na classe, então, tal código passa a ser de imediato genérico para todas as instâncias, ficando o resultado do mesmo apenas dependente dos valores que tais variáveis tiverem no momento em que esta recebe a mensagem.

Saliente-se também, mais uma vez, a diferenciação clara entre o que é uma mensagem enviada a uma instância, e o resultado de tal mensagem ser enviada, que é a activação do método que lhe corresponde. Mensagens e métodos são, portanto, entidades distintas.

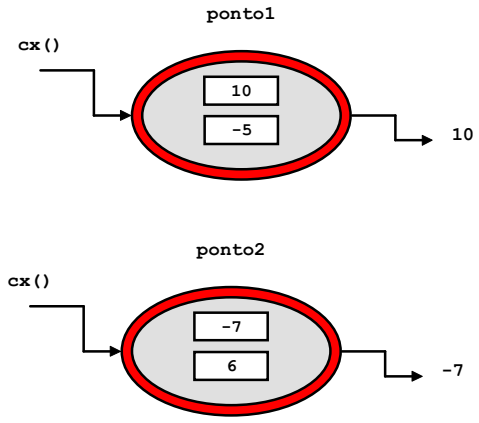

Figura 3.2 – Mensagem-receptor-resposta

3.2 CRIAÇÃO DE CLASSES: INTRODUÇÃO

A noção de classe, tal como foi definida anteriormente, é suficientemente clara para que, de imediato, possam ser criadas algumas classes simples, e estas sejam utilizadas para criar instâncias com as quais se realizem algumas computações. Vamos igualmente aproveitar a necessidade de definir classes novas para introduzir gradualmente outros conceitos muito importantes em PPO, analisando como os mesmos estão representados na linguagem JAVA.

3.2.1 CLASSE Ponto: REQUISITOS INICIAIS

Pretende-se com este primeiro pequeno projecto criar uma classe que especifica a estrutura e o comportamento de objectos do tipo `Ponto`, que serão instâncias de tal classe, e que se pretende que satisfaçam o seguinte conjunto de requisitos:

- Os pontos deverão possuir coordenadas inteiras;
- Deverá ser possível criar pontos com coordenadas iniciais iguais a 0;
- Deverá ser possível criar pontos com coordenada em X e coordenada em Y, iguais aos valores dados como parâmetros;
- Deverá ser possível saber qual a coordenada em X actual de um ponto;
- Deverá ser possível saber qual a coordenada em Y actual de um ponto;
- Deverá ser possível incrementar as coordenadas em X e em Y com os valores dados como parâmetro;
- Deverá ser possível decrementar as coordenadas em X e em Y com os valores dados como parâmetro;
- Deverá ser possível determinar se o ponto tem ambas as coordenadas positivas;
- Deverá ser possível determinar se o ponto é simétrico, ou seja, se dista igualmente do eixo dos X e do eixo dos Y.

3.2.2 DEFINIÇÃO DA ESTRUTURA

Em PPO, tal como veremos mais adiante, quando se pretende construir uma nova classe, devemos começar por analisar se tal classe não poderá ser definida à custa de outras classes já existentes, ou seja, reutilizando o que já existe em JAVA. No entanto, e porque tal análise implica possuir já bastantes conhecimentos sobre a linguagem de programação, em especial sobre as classes já existentes, vamos de momento admitir que esta nova classe vai ser por nós contruída a partir do zero.

Assim, a primeira decisão será a que diz respeito à definição da **estrutura interna** de cada um dos *pontos* que pretendemos, ou seja, pensarmos quais deverão ser as respectivas **variáveis de instância** (os seus nomes e os seus tipos).

Neste caso, cada ponto deverá ter apenas que conter dois valores de tipo inteiro, um correspondente ao valor da sua coordenada em X e o outro ao valor da coordenada em Y. Assim sendo, os pontos que pretendemos criar necessitam de ter duas variáveis de instância a que vamos dar os nomes de `x` e `y` e que vão ser do tipo `int`.

Vamos, então, ver como gradualmente vai evoluindo a classe `Ponto` que se quer construir. A definição de uma classe é, em JAVA, realizada dentro de um bloco que é antecedido pela palavra reservada `class`, à qual se segue o identificador da classe que deve sempre começar por uma letra maiúscula. Dentro do bloco, serão dadas de forma apropriada todas as definições necessárias à criação da classe, ou seja, as definições dos diversos possíveis **membros da classe** (*class members*). Tais definições consistem, por agora, na declaração das diversas variáveis de instância e na codificação de cada um dos métodos de instância.

Assim, partindo de um esqueleto básico contendo apenas a identificação da classe, como:

```
class Ponto {
  // definições
}
```

vamos introduzir a definição de cada variável de instância, separando sempre as várias declarações com comentários que auxiliam à legibilidade dos programas.

```
class Ponto {
   // variáveis de instância
   int x;
   int y;
}
```

É, em geral, boa prática colocar cada variável de instância numa linha separada, e em comentário indicar o que representa. É no entanto redundante documentar o óbvio, como seria o caso neste primeiro exemplo.

3.2.3 DEFINIÇÃO DO COMPORTAMENTO: OS CONSTRUTORES

Em geral, o primeiro código que se define quando se constrói uma classe, é o de uns métodos particulares em JAVA e C++ designados por **construtores**, que não são métodos de instância, e que são por isso métodos especiais dado que apenas são utilizados numa construção JAVA muito particular que tem por objectivo **criar instâncias** de uma dada classe.

Em JAVA, desde que uma classe tenha já sido definida, por exemplo uma classe designada por `Ponto`, é de imediato possível criar instâncias dessa classe se usarmos uma das seguintes construções sintácticas:

```
Ponto p1 = new Ponto();
```

ou de forma equivalente,

```
Ponto p1;
p1 = new Ponto();
```

Em ambos os casos começamos por declarar uma variável `p1` como sendo do tipo `Ponto`, ou seja, capaz de referenciar objectos que são instâncias de tal classe. Em seguida, ainda que de formas diferentes, a essa variável é associado o resultado da execução da expressão `new Ponto()`. Esta expressão, que traduzida se torna clara, corresponde à criação de **uma nova instância** da classe `Ponto` pela execução do método `Ponto()`, método especial com nome igual ao da classe.

A nova instância é referenciada através da variável `p1`. O método especial `Ponto()` que em conjunto com `new` é capaz de construir instâncias da classe `Ponto` designa-se, por tal razão, um **construtor**.

Em JAVA, para cada classe de nome, por exemplo, **`C`**, é **automaticamente** associado um construtor **`C()`**, capaz de criar instâncias de tal classe quando usado em conjunto com a palavra reservada **`new`** em expressões como **`new C()`**.

Adicionalmente, JAVA permite ao programador criar tantos construtores quantos os que achar necessários para criar instâncias de dada classe, impondo apenas uma regra óbvia: **todos os construtores possuem o mesmo nome da classe, podendo diferir, no entanto, quanto ao número e tipo dos seus parâmetros**.

No nosso exemplo, caso nenhum outro construtor fosse por nós definido na classe `Ponto`, a expressão usando o operador `new`, como em:

```
Ponto p1 = new Ponto();
```

associaria à variável `p1` uma instância de `Ponto` que teria nas suas variáveis de instância `x` e `y` o valor inicial 0, dado estas serem do tipo `int` e 0 ser o valor predefinido para variáveis deste tipo. Como facilmente se compreende, estes são valores iniciais adequados para um objecto do tipo `Ponto`. Porém, vamos desde já habituar-nos a redefinir o construtor por omissão, realizando a sua definição de forma explícita para que não haja qualquer ambiguidade.

Portanto, e em função da análise realizada, vamos dar uma definição nossa para o construtor `Ponto()`, que será executado em vez do referido construtor por omissão, ou seja, que se vai **sobrepor** ao construtor por omissão, e que vai atribuir explicitamente o valor `0` a cada variável de instância do ponto criado. O código do mesmo seria, nesta fase, simplesmente:

```
Ponto() { x = 0; y = 0; }
```

Os **construtores** de uma classe são todos métodos especiais que são declarados na classe, tendo por identificador o exacto nome da classe, e tendo por argumentos valores de qualquer tipo de dados, e cujo objectivo é criar instâncias de tal classe que sejam de imediato manipuláveis. Os construtores servem apenas para criar novas instâncias de uma dada classe, pelo que não faz sentido que tenham que especificar um qualquer tipo de resultado.

Por outro lado, é sempre possível, e muitas vezes bastante útil, definir mais de um construtor de instâncias de uma dada classe, construtores que, em geral, apenas diferem nos valores iniciais atribuídos às variáveis de instância, ou quanto à forma de atribuição de tais valores a tais variáveis (como veremos mais tarde).

Se pretendermos no exemplo em questão criar instâncias da classe `Ponto` atribuindo de imediato às suas variáveis de instância valores iniciais dados como parâmetro, então, poderemos criar um construtor adicional que receba as coordenadas como parâmetro e as atribua às variáveis de instância da instância criada, como:

```
Ponto(int cx, int cy) { x = cx; y = cy; }
```

A nossa classe `Ponto` tem neste momento apenas construtores e estrutura, ou seja, não tem definido qualquer comportamento – dado pelo conjunto de operações ou métodos, quer de consulta quer de modificação –, pelo que, se criássemos neste momento instâncias da classe, nada poderíamos realizar com as mesmas. Os métodos de instância são, portanto, métodos essenciais para que possamos utilizar as instâncias.

```
class Ponto {
  // Construtores
  Ponto() {
    x = 0; y = 0;
```

```
    }
    Ponto(int cx, int cy) {
      x = cx; y = cy;
    }
    // Variáveis de Instância
    int x;
    int y;
    ...
}
```

3.2.4 DEFINIÇÃO DO COMPORTAMENTO: OS MÉTODOS DE INSTÂNCIA

Vamos agora iniciar o processo de programar em JAVA os **métodos de instância** que irão ser os responsáveis pelo comportamento das instâncias de `Ponto` quando recebem as mensagens a que deverão ser capazes de responder.

Dado que a estrutura sintáctica geral para a completa definição em JAVA de um método de instância é relativamente complexa, vamos enveredar por um processo de apresentação da mesma em função das necessidades, isto é, as mais complexas particularidades definicionais serão apresentadas sempre que, perante um exemplo, as mesmas tenham que ser introduzidas.

Vamos também, desde já, introduzir uma regra crucial para uma correcta programação em PPO, regra apresentada no capítulo 1, e que propõe o seguinte:

- "Qualquer que seja a linguagem de PPO usada, a única forma de se poder garantir, mesmo usando os mecanismos típicos da PPO, que estamos a programar entidades independentes do contexto e, assim, reutilizáveis, é garantir que nenhuma cápsula faz acesso directo às variáveis de instância de outra cápsula, mas que apenas invoca por mensagens os métodos de acesso a tais valores, métodos que para tal devem ser programados nas cápsulas".

Esta regra, extraordinariamente importante, impõe duplas responsabilidades aos que têm que implementar as classes que vão gerar instâncias. Por um lado, devem saber que, na definição dos métodos de instância de um dado objecto, não devem escrever código que corresponda a aceder directamente às variáveis de instância de um qualquer outro objecto, pois tal gera código dependente da implementação interna de tal objecto, logo dependente do contexto. Por outro lado, para que do exterior se possa ter acesso aos valores das variáveis de instância de uma forma correcta, então, é necessário que as **interfaces** forneçam os métodos adequados a tal acesso (**métodos de consulta** das variáveis).

Note-se ainda que estas regras correspondem a princípios de programação que não são verificados pelos compiladores, pelo que compete aos programadores apenas o seu cumprimento. De facto, mesmo as mais evoluídas linguagens de PPO possibilitam, ou seja, não detectam nem em tempo de compilação nem noutro qualquer, situações de completa infracção desta regra tão importante.

Por exemplo, em JAVA, e usando o que apenas foi até agora definido para a classe `Ponto`, uma classe "cliente" da classe `Ponto`, ou seja, uma classe que esteja a usar esta classe, poderia ter um método que, necessitando de ter o valor actual da variável de instância `x` da instância referenciada pela variável `p1`, contivesse o código seguinte:

```
int cx = p1.x;
```

Este código passaria como correcto no compilador de JAVA. De facto, o operador ponto (`.`) funciona em JAVA, tal como em várias outras linguagens, como um **selector**, que possibilita aceder por nome a uma variável, ou método, do objecto referenciado pela variável escrita do seu lado esquerdo. Poderíamos até, usando tal construção, alterar directamente o valor da variável de instância `y` de `p1` escrevendo, por exemplo:

```
p1.y = -9;
```

Porém, do ponto de vista dos princípios fundamentais da PPO, este código está formalmente incorrecto e nem sequer deveria ser aceite como correcto na fase de compilação, dado conter uma expressão baseada no acesso

directo à representação interna de outro objecto. Tal como anteriormente foi demonstrado, o que acontece ao código fonte desta classe, que é "cliente" da classe `Ponto`, caso a classe `Ponto` seja redefinida, por exemplo, alterando o nome da variável de instância ou, pior ainda, a representação do valor inteiro, é que se torna inutilizável dado usar uma representação que passa a estar errada. Por outro lado, e do ponto de vista da instância `p1`, qualquer classe cliente poderá deste modo modificar o seu estado interno a qualquer momento e sem qualquer controlo desta.

É, no entanto, comum ler-se em livros sobre programação por objectos usando a linguagem JAVA ou outras, publicados por algumas editoras internacionais de nome credível nesta área, a informação de que, caso se pretenda ter acesso a uma variável *x* de uma instância referenciada, por exemplo, como *c*, se deve escrever ***c.x***, frase que está sintacticamente correcta.

Obviamente que, dentro dos princípios de programação que vimos defendendo para que, de facto, possamos atingir certos objectivos no desenvolvimento de aplicações de alguma escala, em particular com o auxílio de certas construções oferecidas pelo paradigma da PPO, tal tipo de codificação é, como se provou, completamente desaconselhável.

Naturalmente que, quando não são estes os objectivos a perseguir na utilização de JAVA e das suas construções de PPO, mas antes, por exemplo, máxima eficiência do código, custe o que tal possa custar em correcção de erros, reutilização e generalidade, então, tais construções podem ser usadas, sendo certo que sacrificam tais objectivos, que, no nosso caso, se consideram fundamentais.

Exactamente por estas razões, no livro *The Java Language Specification*, dos autores da linguagem JAVA, a primeira regra por estes apresentada, relativamente aos nomes a atribuir aos métodos, sugere que todos os métodos que fazem acesso ao valor de uma variável de instância de nome genérico **x** devem ser designados por **`getX`** (ex.: `getNome()`, `getConta()`, etc.). Por outro lado, todos os métodos que alteram o valor de uma variável de instância **x** devem ser designados por **`setX`** (ex.: `setNome()`, `setConta()`, etc.), ou seja, desta forma desaconselhando indirectamente a manipulação do identificador e do valor da variável usando um selector.

Os métodos do tipo `getX()`, porque realizam a consulta do valor de dada variável e devolvem esse valor como resultado, designam-se por **interrogadores** ou **selectores**. Tipicamente devolvem um resultado que é do mesmo tipo da variável consultada e não possuem parâmetros de entrada. Os métodos do tipo `setX()`, dado terem por objectivo alterar o valor da variável de instância, designam-se por **modificadores** e, em geral, têm parâmetros de entrada mas não devolvem qualquer resultado. Naturalmente que não é obrigatório usar exactamente **set** e **get**, podendo usar-se identificadores que reflictam no entanto a semântica (ex.: `mudaX()`, `alteraX()`, etc.).

A sintaxe geral para a definição de um método de instância consiste em JAVA na definição do seu **cabeçalho** (*header*) e do seu **corpo** (*body*). A estrutura mais básica de declaração do cabeçalho de um método é a seguinte:

<tipo de resultado> <identificador> ***(*** *tipo nome_parâmetro, ...* ***)***

O *<identificador>* de um método é uma sequência de letras e dígitos, devendo o primeiro carácter ser uma letra e, por razões de estilo, minúscula.

A lista de parâmetros formais *<pares tipo-nome: parâmetros>* é constituída por zero ou mais pares *tipo nome_parâmetro*, que definem os tipos e identificadores dos parâmetros formais do método. Os tipos, de momento, tanto podem ser identificadores de tipos simples, como *arrays* ou nomes de classes. Esta lista, que pode ser vazia, é delimitada por parêntesis.

O *<tipo de resultado>* de um método, de momento, poderá ser um identificador de um dos tipos simples da linguagem (ex.: `int`, `boolean`, `real`, `float`, `char`, etc.), um *array* de um dado tipo, simples ou referenciado, ex.: `int[]` ou `String[]`, o nome de uma qualquer classe (ex.: `String`, `Date`, `Point`, etc.) e, ainda, a palavra reservada **`void`** caso o método não devolva qualquer resultado.

Apresentam-se, em seguida, alguns exemplos deste tipo de cabeçalhos, usualmente designados também como **assinaturas** dos métodos (se excluirmos o tipo do resultado):

```
int getConta()
boolean maiorQue(int valor)
```

```
void setValor(int valor)
boolean valorEntre(real min, real max)
String toString()
nt[] notasAcimaDe(int NotaRef)
Triangulo maiorTtri(Triangulo t2)
```

A **assinatura** (*signature*) de um método é constituída pelo nome do método e pelos seus parâmetros (em número, tipo e ordem). A assinatura de um método é um elemento caracterizador do mesmo e, como veremos mais tarde, um elemento importante na resolução de conflitos que podem surgir na definição de classes, dado que a assinatura de um método, tal como o nome indica, é uma chave que deve ser única para que o interpretador de JAVA possa aceder ao código do mesmo para o executar. Assim, não devem existir conflitos de assinaturas em métodos da mesma classe.

O **corpo de um método** é definido como sendo um bloco de instruções, tendo por delimitadores as usuais chavetas **{ }**. Caso o método devolva um resultado, este será o valor resultante do cálculo do valor da expressão que está associada à primeira instrução **return** que for executada, instrução essa que, em boa programação, aparecerá uma só vez no código do método, em geral no final do mesmo (salvo raras excepções).

Retomando o exemplo da classe `Ponto`, podemos agora escrever o método de instância que deve devolver como resultado o valor actual da variável de instância `x`, método que deve poder ser invocado do exterior da classe, logo, sendo "público":

O código de tal método, usando a regra atrás referida para os nomes, será o seguinte:

```
// interrogador – selector
int getx() { return x; }
```

Como se pode verificar, o método limita-se a devolver o valor da variável de instância `x`. Assim, e mais uma vez admitindo que a variável `p1` referencia uma instância de `Ponto`, qualquer objecto externo – por exemplo, criado no contexto de um programa principal contendo o método `main()` onde se criariam diversos pontos para com eles testarmos várias das operações que estamos a programar –, passa a poder ter acesso ao valor da variável de instância de `p1`, escrevendo o código que corresponde ao envio da mensagem que activa o método `getx()`, como por exemplo:

```
int cx = p1.getx();
System.out.println(" X de p1 = " + cx);
```

ou de forma equivalente e mais simples:

```
System.out.println(" X de p1 = " + p1.getx());
```

código que permitiria consultar o valor actual da coordenada em `x` de `p1`, e, através da instrução básica de JAVA para *output* em ecrã, apresentar tal valor ao utilizador. O mesmo faríamos para a coordenada em `y`.

Vamos, em seguida, desenvolver outros métodos que, conforme os requisitos originais, se pretendem ver implementados na classe `Ponto`. Assim, pretende-se programar um método modificador que incremente x e y com os valores dados como parâmetro. Tal método terá como resultado `void`, e será designado por `incCoord`, tendo os dois parâmetros de entrada que vão incrementar os valores de x e de y:

```
// modificador – incremento das Coordenadas
void incCoord(int deltaX, int deltaY) {
   x = x + deltaX; y = y + deltaY ;
}
```

Não devolvendo este método qualquer resultado, a sua invocação no código de um objecto exterior ou de um programa principal teria a seguinte forma:

```
p1.incCoord(dx, dy);
```

ou seja, o objecto exterior ao invocar tal método altera o estado interno de `p1`, mas, embora o ponto receptor reaja a tal invocação mudando o seu estado interno, o objecto exterior (invocador) não necessita de receber qualquer

resultado (nem poderia porque o método tem `void` por resultado). Note-se mais uma vez, pelo exemplo, a similaridade entre a forma sintáctica da mensagem e a assinatura do método, ou seja, o envio da mensagem resulta na invocação do método sintacticamente correspondente.

Vamos em seguida desenvolver o código do método modificador que permite que as coordenadas sejam decrementadas. O código do método será, neste caso o seguinte:

```
// modificador – decremento das Coordenadas
void decCoord(int deltaX, int deltaY) {
    x -= deltaX; y -= deltaY;
}
```

3.2.5 A REFERÊNCIA this

No desenvolvimento do código dos métodos de instância de uma dada classe, há por vezes a necessidade de referenciarmos ou uma variável de instância própria, interna ao próprio objecto que estamos a programar, ou, até, invocar um método interno de tal objecto. Porém, se ainda estamos a programar tal objecto, ele ainda não tem identificador para ser referenciado por nome. Assim, como referenciá-lo?

Em JAVA, sempre que dentro do código de um dado objecto pretendermos ter acesso a uma variável ou método do mesmo objecto, podemos fazê-lo usando a palavra reservada `this`, que é uma **referência especial**.

Um exemplo muito comum da necessidade de utilização de `this`, e que até agora propositadamente evitamos, ocorre quando os identificadores dos parâmetros de entrada coincidem com os nomes das variáveis de instância (o que é normal por serem mnemónicos e portanto mais fáceis de perceber, quer na API quer no código do objecto).

Por exemplo, se num construtor tal acontecesse, teríamos então,

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

em que `this.x` distingue claramente que se trata da variável de instância `x`, de modo a resolver a potencial ambiguidade. Assim se fará sempre que tal seja necessário, por razões de clareza do código.

Quanto aos métodos, a questão que se coloca não será tanto a da ambiguidade, mas antes a necessidade de, na codificação de um método, se pretender invocar um outro método já definido. Vejamos um exemplo simples utilizando o método `decCoord(int, int)` atrás codificado. Imaginemos que nos era pedido para programarmos um método que apenas decrementasse a coordenada em X. A primeira solução óbvia seria a seguinte:

```
// modificador – decrementa Coordenada X
void decCoordX(int deltaX) {
    x -= deltaX;
}
```

Outra possível solução seria, reutilizando o método já antes definido:

```
// modificador – decrementa Coordenada X
void decCoordX(int deltaX) {
   this.decCoord(deltaX, 0);     // invoca decCoord() local
}
```

usando `this.decCoord()`, em que `this` é uma referência especial para o próprio objecto (contexto) que estamos a definir, lendo-se: "invocar o método local `decCoord()`" ou, até, "invocar o método `decCoord()` deste objecto". O método é invocado com o segundo parâmetro igual a 0 para que a coordenada em Y não seja alterada.

É boa prática escrever `this.m()` quando estamos a invocar um método interno. No entanto, se tivéssemos apenas escrito `decCoord(deltaX, 0)`, o compilador adicionaria a referência `this` de forma automática. Nas variáveis de instância, `this` apenas deve ser usada quando houver colisões com identificadores de parâmetros, já que as variáveis de instância são, por definição, locais ao objecto.

Vamos, agora, continuar com o desenvolvimento do código dos métodos de instância restantes. O código do método que determina se um ponto tem ambas as coordenadas positivas deverá devolver como resultado um `boolean`, não tendo parâmetros de entrada:

```
/* determina se um ponto tem ambas as
   coordenadas positivas */
boolean coordPos() {
    return x > 0 && y > 0 ;
}
```

Aproveitamos para introduzir os comentários multilinha de JAVA que são todas as frases escritas delimitadas por **`/*`** …**`*/`** e que o compilador despreza.

Após estes pequenos primeiros exemplos de definição de métodos de instância, é já possível chamar a atenção para o facto de que, quando temos que definir o código de tais métodos, e no sentido de tornar tal processo mais fácil, nos deveremos sempre colocar, de um ponto de vista conceptual, numa posição semelhante à da própria instância que recebe a correspondente mensagem. Isto é, será sempre mais simples codificar tais métodos se pudermos atribuir algum grau de antropomorfismo aos objectos que estamos a programar, ou assumindo nós próprios a sua posição de "prestação de um serviço", em qualquer dos casos procurando sempre a melhor resposta à seguinte questão que é fundamental: "com a minha estrutura interna e com o auxílio de tudo o que o ambiente me permite utilizar, como devo programar uma resposta correcta ou apenas modificar o meu estado interno, em função dos requisitos apresentados, à mensagem **m** que me foi enviada?".

Esta noção de ter que programar a **prestação de um serviço** (seja ele um resultado de facto ou apenas uma alteração interna) em resposta à recepção de um pedido formal solicitado sob a forma de uma mensagem, é crucial para a compreensão e para a boa utilização do paradigma da PPO, dado ser esta a sua essência.

Por outro lado, e até por tudo o que já vimos anteriormente, sendo o modelo computacional básico da PPO o envio de uma mensagem a um objecto receptor, a frase fundamental da escrita de código em PPO, e naturalmente em JAVA, será, sendo `obj` um objecto e `m()` uma mensagem, a seguinte frase de estrutura básica:

obj.m();

que, a partir desta forma mais simples em que a mensagem não tem argumentos e não há resultado do método invocado, se generaliza para todas as outras formas, que são:

```
obj.m(p1, p2, …, pn);          // modificador com parâmetros
res = obj.m();                 // interrogador simples
res = obj.m(p1, p2, …, pn);    // interrogador com parâmetros
```

As mensagens podem por vezes ser escritas como uma sequência, **obj.m1().m2()**, devendo ser interpretadas da esquerda para a direita, isto é, a mensagem **m2()** será enviada ao objecto que é o resultado obtido após o envio de **m1()** a **obj**.

Note-se que, dentro desta metodologia que aqui se está a procurar definir, nunca iremos criar métodos que, em simultâneo, modifiquem o estado interno de um objecto e devolvam um resultado para o exterior. Isto é, cada método ou é um **modificador** ou é um **interrogador**, e bastará verificar o *pattern* da sua utilização (ver as frases acima) para se saber perfeitamente o que significa tal linha de código.

O método seguinte visa determinar se um ponto dista igualmente do eixo dos X e do eixo dos Y, isto é, se, em valor absoluto, as suas coordenadas `x` e `y` são iguais. Ora, a função que calcula o valor absoluto de um inteiro (ou de outro tipo numérico qualquer) encontra-se em JAVA codificada numa classe chamada `Math`.

Vamos em seguida ver como é que em JAVA podemos ter acesso a classes que estão predefinidas e saber como podemos realizar a sua importação para o contexto das classes que estamos a definir, por forma a podermos utilizar tudo o que nesta tenha sido definido.

3.2.6 *PACKAGES* E IMPORTAÇÃO DE CLASSES

Em JAVA, um ficheiro que contenha código fonte tem a extensão *.java* e um nome igual ao da classe pública definida em tal ficheiro. O ficheiro onde colocaríamos a nossa classe `Ponto` teria pois a designação *Ponto.java* e conteria apenas a definição da classe `Ponto`. Após a compilação de uma classe, é gerado um ficheiro de *byte-*

codes, designado por **ficheiro de classe**, que terá o mesmo nome da classe compilada mas extensão *.class*. A compilação do ficheiro *Ponto.java* daria, pois, origem ao ficheiro *Ponto.class*, ficheiro este que, só por si, não é executável pelo interpretador de JAVA pois não possui um método **`main()`** para início de execução. Por isso, em geral, constrói-se uma pequena classe contendo tal método `main()` no qual se usa a classe definida, `Ponto`, para criar instâncias e com estas testar os métodos desenvolvidos, tal como se tal constituísse um programa principal de teste da classe (ver exemplo adiante).

Por outro lado, em JAVA, as classes são em geral agrupadas em *packages*, desta forma sendo possível manter as classes em compartimentos ou pacotes distintos, em geral agrupadas em função da sua funcionalidade. Em especial, tal facto facilita a procura das classes, dado que os *packages* têm nomes bastante representativos da funcionalidade das classes que os constituem.

Por exemplo, o *package* de JAVA que contém todas as classes relacionadas com as operações de *input/output* designa-se por *package java.io*. O *package* de JAVA que contém um conjunto de classes que implementam estruturas e tipos de dados de grande utilidade geral designa-se por *package java.util*; o *package java.lang* reúne todas as classes fundamentais à execução de programas JAVA. Em JAVA, existem actualmente definidos cerca de 130 diferentes *packages*. Ao longo deste livro necessitaremos de, no máximo, utilizar classes de seis destes *packages*.

Qualquer classe ou membro de uma classe pode ser referenciado de forma absoluta usando como prefixo o nome do *package*, seguido do identificador da entidade a que se pretende aceder como, por exemplo, em:

```
comp = java.Math.min(lista.size(), MIN);
java.lang.System.out.println("abc");
java.util.Scanner;
```

Porém, sempre que na definição de uma classe necessitarmos de usar classes pertencentes a um dado *package*, poderemos sempre usar uma **cláusula de importação**, que torna os elementos importados disponíveis na definição da classe que os importa, podendo, a partir daí, as classes serem designadas pelos seus nomes abreviados.

As **cláusulas de importação**, obrigatoriamente escritas antes do início da definição da classe, permitem realizar **importações qualificadas** (específicas), em qualquer número, tal como por exemplo em:

```
import java.lang.String;
import java.zip.ZipInputStream;
import java.io.BufferedReader;
```

bem como **importações globais** de todo o *package*, tal como em:

```
import java.util.*;
```

A cláusula de importação `import java.lang.*;` é automaticamente assumida pela JVM, pelo que está sempre implícita em qualquer programa JAVA. Não há, pois, a necessidade de a escrever quando se pretende importar todo o *package*.

Se ao definirmos uma dada classe pretendermos que esta vá fazer parte de um dado *package* já existente, então a primeira declaração a fazer no ficheiro fonte é a indicação do *package* a que a mesma vai pertencer, conforme em:

```
package java.minhasClasses;
package jogos.classes;
```

No entanto, há que ter o cuidado adicional de acrescentar à variável de ambiente de nome CLASSPATH (ver secção do capítulo 2 sobre a instalação do JDK) os identificadores dos directórios onde tais classes devem ser procuradas pelo interpretador.

Finalmente, todas as classes que não possuam qualquer referência ao *package* de que vão fazer parte são colocadas pelo compilador num *package* particular que se designa por "*package* por omissão" (*default*) e que é automaticamente associado ao directório actual de trabalho, desde que na variável CLASSPATH tal caminho seja referenciado através do símbolo "**.**".

Continuando agora com a definição da classe `Ponto`, resta-nos codificar o método que determina se um ponto é simétrico, ou seja, se as suas coordenadas são iguais em valor absoluto. Ora, o método que determina o valor absoluto de um qualquer valor numérico encontra-se implementado na classe `java.lang.Math`, e pode ser invocado usando `Math.abs()` (não é pois um método de instância normal, tratando-se de um método especial a explicar mais tarde), lendo-se, "o método `abs()` da classe `Math`". O código do nosso método `simetrico()` será então o seguinte:

```
boolean simetrico() { return Math.abs(x) == Math.abs(y); }
```

Note-se, portanto, que este método especial `abs()` foi invocado utilizando um prefixo, que é o identificador da classe a que pertence, e não obedece ao padrão normal de PPO que consiste no envio de uma mensagem a um objecto receptor. Trata-se, pois, seguramente, de uma excepção que será explicada posteriormente mas que foi necessária aqui, pois foi necessário recorrer a um "serviço" definido na classe `Math`, como muitos outros há em JAVA em muitas outras classes.

No exemplo, este serviço prestado pela classe `Math` consistiu em permitir-nos invocar uma das suas muitas funções matemáticas, `abs(int x)`, (cf. *f(x)*), que nos devolve um resultado. Não há mensagem, não há receptor. Não há PPO. Há apenas serviços auxiliares que se encontram disponíveis numa determinada classe e que podem ser invocados.

Estas classes não servem para criar instâncias pois não possuem tal capacidade, acabando por se comportar como módulos que implementam facilidades através dos seus métodos.

3.2.7 DEFINIÇÃO ACTUAL DA CLASSE Ponto

```
class Ponto {
   // Construtores
   Ponto() { x = 0; y = 0; }
   Ponto(int cx, int cy) {
      x = cx; y = cy;
   }
   // Variáveis de Instância
   int x;
   int y;
   // Métodos de Instância
   // interrogador – devolve X
   int getx() { return x;    }
   // interrogador – devolve Y
   int gety() { return y;    }
   // modificador – incremento das Coordenadas
   void incCoord(int deltaX, int deltaY) {
      x = x + deltaX; y = y + deltaY;
   }
   // modificador – decremento das Coordenadas
   void decCoord(int deltaX, int deltaY) {
    x -= deltaX; y -= deltaY;
   }
   /* determina se um ponto tem ambas as
     coordenadas positivas */
   boolean coordPos() {
      return x > 0 && y > 0 ;
   }
   /* determina se um ponto é simétrico, ou seja
      se dista igualmente do eixo dos X e dos Y */
   boolean simetrico() {
      return Math.abs(x) == Math.abs(y);
   }
}
```

3.2.8 REGRAS DE ACESSIBILIDADE A UMA CLASSE

JAVA fornece mecanismos de **controlo de acesso** (ou visibilidade) quer para os *packages*, que contêm conjuntos de classes, quer para as classes individuais, quer para cada um dos **membros** destas, como sejam as variáveis e os métodos que nestas são definidos, quer ainda para outras construções a introduzir posteriormente, tais como as Classes Abstractas e as Interfaces. Estes mecanismos de controlo de acesso especificam quem tem acesso às entidades definidas.

São quatro os **tipos de acesso** (ou de visibilidade) que podem ser especificados em JAVA para estas entidades, dos quais três usando palavras reservadas que se designam por **modificadores de acesso**, especificamente: `public`, `protected` e `private`. O quarto tipo de acesso é definido *por omissão*, ou seja, caso nenhum modificador seja declarado, conforme no exemplo da classe `Ponto`, tal como até aqui a definimos, e designa-se por acesso de nível *package*.

Em qualquer dos casos, o **modificador de acesso** é, em geral, especificado antes de qualquer outra especificação da entidade, seja ela classe, variável ou método, ou seja, no início da sua declaração.

Na Tabela 3.1 sintetizam-se as regras de acesso ou visibilidade associadas a cada modificador de acesso para as **classes**.

Modificador	Acessível a partir do código de
`public`	Qualquer classe
`protected`	(*)
`private`	(*)
nenhum	Classes dentro do seu *package*

Tabela 3.1 – Modificadores de acesso para classes

Classes do tipo `private` e `protected` são, em JAVA, classes especiais (**inner classes*) que são usadas na implementação de classes predefinidas complexas de JAVA, algumas das quais serão referidas mais adiante neste livro. Estas classes são definidas no interior de outras classes, implementando funcionalidades especiais (ver capítulo 11).

A classe `Ponto` anteriormente definida, dado não ter sido declarada usando um modificador particular, estará acessível apenas dentro do respectivo *package*. Porém, não declarámos a que *package* é que esta vai pertencer. Em tal situação, JAVA associa tal classe a um *package* particular sem identificador, o que se torna conveniente numa fase de desenvolvimento e teste da classe, dado que, em tal caso, o código pode ser interpretado a partir do directório corrente.

No entanto, se pretendermos associar uma dada classe a um *package*, então temos que usar uma declaração de *package*, devendo esta ser a primeira linha de código do programa. Por exemplo, se pretendêssemos declarar que a classe `Ponto` depois de compilada deverá fazer parte do *package* `java.classes1`, cujo directório deverá fazer parte da CLASSPATH, então, escreveríamos no ínicio do ficheiro de definição da classe `Ponto`:

```
package java.classes1;
class Ponto {
```

Se, adicionalmente, pretendêssemos, tal como deve ser feito em geral, tornar a classe pública, então, acrescentaríamos o respectivo modificador de acesso antes da palavra reservada `class`, tal como em:

```
package java.classes1;
public class Contador {
```

3.2.9 REGRAS DE ACESSO A VARIÁVEIS E MÉTODOS DE INSTÂNCIA

A acessibilidade a variáveis e métodos de instância de uma dada classe para cada tipo de modificador de acesso é, como se pode verificar pela tabela a seguir, exactamente igual. No entanto, as regras para a aplicação de tais modificadores são completamente diferentes para cada caso.

Modificador	Acessível a partir do código de
`public`	Qualquer classe
`protected`	Própria classe, qualquer classe do mesmo *package* e qualquer subclasse
`private`	Própria classe
nenhum	Própria classe e classes dentro do mesmo *package*

Tabela 3.2 – Modificadores de acesso para variáveis e métodos

As variáveis de instância de uma dada classe podem ser declaradas de tal forma que nenhum código, a não ser o código dos métodos de instância da classe onde a variável é declarada, se lhes pode referir, ou seja, podem ser tornadas variáveis de acesso restrito ao código existente dentro da classe.

Para tal, como se pode ver pela tabela acima, bastará que na declaração da variável se use o modificador `private`, como as formas sintácticas do exemplo seguinte:

```
private double raio;
private String nome;
private int x;
```

Tal implica também, que exemplos anteriormente referidos de má programação são imediatamente eliminados, pois expressões, tais como `cx = p1.x;` usadas exteriormente à classe onde `x` é declarada, passam agora a dar mesmo um erro de compilação. Note-se que em C++, por exemplo, as variáveis de instância são, por omissão, privadas.

Ao garantir o total encapsulamento da sua estrutura de dados interna à classe (*data hiding*), o programador terá agora que garantir, através de adequados métodos de consulta e de modificação, que os "clientes" de tal classe possuem uma API que lhes permita consultar e alterar o estado interno dos objectos criados.

Quanto aos métodos de instância, em geral passa-se exactamente o contrário. De facto, eles são escritos para que do exterior possam estar acessíveis aos "clientes" de tal classe, ou seja, a todas as classes que pretendam usar a classe criando instâncias e realizar com estas computações invocando os seus métodos através de mensagens. Portanto, e em geral, métodos de instância são de tipo de acesso `public`. A API (*Application Programming/Programmer´s Interface*) de uma classe de JAVA é o conjunto dos seus métodos de instância com acesso `public`. A estes devemos adicionar os construtores, que devem igualmente ser `public` para que as outras classes lhes tenham acesso, e assim possam invocá-los para criar instâncias desta.

Os métodos de instância definidos como `private`, sendo apenas acessíveis ao código da própria classe, acabam por poder ser vistos como métodos auxiliares, que por vezes tornam mais simples a codificação de métodos de instância mais complexos, realizando cálculos intermédios, suboperações, etc.

Finalmente, as variáveis declaradas dentro dos métodos são **variáveis locais**, sendo acessíveis apenas a estes e deixando de ter existência quando tais métodos terminam a sua execução. Variáveis declaradas dentro de blocos { .} existem apenas dentro destes.

As regras de acessibilidade agora introduzidas levam-nos a ter que reescrever de forma mais correcta a classe `Ponto` que desenvolvemos. A classe deve ser uma classe pública, tal como pretendemos que sejam públicos os seus construtores. As variáveis de instância devem ser, portanto, declaradas como `private`, e os respectivos métodos selectores e modificadores definidos como `public` também.

Estas vão ser, em geral, as formas de acesso mais comuns aos membros das nossas classes. Construtores `public`, variáveis de instância `private` e métodos `public`.

3.2.10 DEFINIÇÃO FINAL DA CLASSE Ponto

```
public class Ponto {
   // Construtores
   public Ponto() { x = 0; y = 0; }
   public Ponto(int cx, int cy) { x = cx; y = cy; }
   //
   // Variáveis de Instância
   private int x;
   private int y;

   // Métodos de Instância

   // interrogador - devolve X
   public int getx() { return x;  }

   // interrogador - devolve Y
   public int gety() { return y;  }

   // modificador - incremento das Coordenadas
   public void incCoord(int deltaX, int deltaY) {
      x = x + deltaX; y = y + deltaY ;
   }

   // modificador - decremento das Coordenadas
   public void decCoord(int deltaX, int deltaY) {
      x -= deltaX; y -= deltaY;
   }

   /* determina se um ponto tem ambas as
      coordenadas positivas */
   public boolean coordPos() {
      return x > 0 && y > 0 ;
   }

   /* determina se um ponto é simétrico, ou seja
     se dista igualmente do eixo dos X e dos Y */
   public boolean simetrico() {
      return Math.abs(x) == Math.abs(y) ;
   }
}
```

Neste exemplo, não houve a necessidade de usar métodos do tipo `private`. No entanto, este tipo de métodos, apenas acessíveis dentro da própria classe, devem ser vistos como métodos puramente auxiliares à definição da classe, logo, sem qualquer interesse para o exterior. Por tal motivo não fazem parte da API da classe.

3.2.11 COMPILAÇÃO E TESTE DA CLASSE Ponto

A compilação da classe `Ponto`, depois de o código fonte ter sido introduzido num ficheiro de texto tipo WordPad ou NotePad, e gravado como *Ponto.java*, conistirá da invocação do compilador de JAVA na linha de comandos, usando a seguinte forma:

javac *[-opções]* Ponto ↵

A classe `Ponto` não é uma classe por si só executável (tal como qualquer outra deste tipo), pois não possui um método `main()`. Por isso, vamos agora estudar qual a forma usual de, em JAVA, se desenvolver código executável para testar uma dada classe acabada de desenvolver, neste nosso exemplo, a classe `Ponto`.

Para tal, em JAVA, é criada uma classe muito especial que irá servir para testar a classe desenvolvida. Esta classe particular de JAVA vai assumir o papel do que noutras linguagens se designa por **programa principal**. Porém, e por questões de simplicidade, o código da classe de teste não apresentará qualquer interacção com o utilizador para leitura de dados, mas apenas entre a classe de teste e a classe a testar através da invocação dos métodos desta através de parâmetros constantes.

Ou seja, pretendemos desenvolver uma classe de teste da classe `Ponto` que nos permita criar instâncias de `Ponto`, enviar a estas instâncias mensagens que correspondem à invocação e execução dos métodos da sua API, consultando e modificando o estado interno de tais instâncias e podendo observar no monitor os diversos resultados produzidos pelo envio de tais mensagens (introduziremos adiante o ambiente BlueJ que muito nos facilitará esta e outras tarefas).

Esta classe, criada apenas com o objectivo de testar uma dada classe desenvolvida, tem algumas características particulares que devemos agora descrever e analisar.

Em primeiro lugar, e tal como a classe em desenvolvimento e teste, esta classe não deve estar associada a qualquer *package* particular, mas antes ao *package* por omissão, que é associado ao directório corrente. Sendo, em geral, de acesso público, esta classe tem a seguinte estrutura de definição genérica:

```
public class TestePonto {
// Classe de teste da Classe Ponto.
// Aqui se escrevem variáveis globais e
// métodos auxiliares.
   public static void main(String args[]) {
/* Aqui se escreve todo o código de teste, ou seja,
    de criação de instâncias, de envio de mensagens,
    de apresentação de resultados, etc. */
   }
}
```

Tal método tem que ter o nome **`main`** e o atributo **`static`**, cujo significado será mais à frente explicado, dado que o interpretador de JAVA, ao ser invocado sobre uma qualquer classe, inicia a execução exactamente pelo método declarado como `static main()`.

Ainda que de momento não possa ser completamente definido o significado do atributo `static`, esta é a estrutura a construir para a criação de uma classe JAVA que tenha por objectivo servir como classe de teste da camada computacional das classes definidas pelo utilizador, usando sempre apenas as API disponibilizadas por tais classes.

O método `main()` de cada classe de teste, vai conter, fundamentalmente, código não interactivo de criação de instâncias, de envio de mensagens às instâncias criadas e código de apresentação de resultados das consultas às instâncias, código de *output* que é escrito usando as instruções básicas `System.out.print()`, ou `println()`, ou `printf()` para escrita no ecrã..

```
public class TestePonto {
   // Classe de teste da Classe Ponto.
   public static void main(String args[]) {
      // Criação de Instâncias
      Ponto pt1, pt2, pt3;
      pt1 = new Ponto();
      pt2 = new Ponto(2, -1);
      pt3 = new Ponto(0, 12);

      // Utilização das Instâncias
      int cx1, cx2, cx3;     // variáveis auxiliares
      int cy1, cy2, cy3;     // variáveis auxiliares
      cx1 = pt1.getx();
      cx2 = pt2.getx();

      // saída de resultados para verificação
      System.out.println("cx1 = " + cx1);
```

```
        System.out.println("cx2 = " + cx2);

        // alterações às instâncias e novos resultados

        pt1.incCoord(4,4); pt2.incCoord(12, -3);
        cx1 = pt1.getx(); cx2 = pt2.getx();
        cx3 = cx1 + cx2;
        System.out.println("cx1 + cx2 = " + cx3);

        pt3.decCoord(10, 20); pt2.decCoord(5, -10);
        cy1 = pt2.gety(); cy2 = pt3.gety();
        cy3 = cy1 + cy2;
        System.out.println("cy1 + cy2 = " + cy3);
    }
}
```

3.2.12 MÉTODOS COMPLEMENTARES USUAIS

Vamos agora complementar a definição da classe `Ponto` com métodos que, não tendo sido pedidos na lista de requisitos, são métodos de implementação quase obrigatória, e que são os métodos que permitem determinar a igualdade de dois pontos, criar uma cópia não partilhada de um ponto e criar uma representação textual (sob a forma de caracteres ASCII/UTF) de um ponto.

O método `igual()` deverá permitir comparar o objecto parâmetro com o receptor e devolver um valor boleano, `true` ou `false`, conforme estes sejam ou não iguais em valor (que não em identidade, que corresponde ao seu endereço). O método `igual()`, por razões óbvias, deve obedecer à seguinte assinatura e resultado:

```
boolean igual(Ponto p)
```

ou seja, receber como parâmetro uma instância da classe `Ponto`, verificar se o parâmetro não é uma referência nula, e, caso não seja, implementar a igualdade de objectos desta classe `Ponto`, sob a forma de uma relação de equivalência (ou seja, sendo reflexiva, simétrica, transitiva e consistente).

Teríamos o seguinte código para comparação de instâncias da classe `Ponto`:

```
public boolean igual(Ponto p) {
   if (p != null)
        return (x == p.getx()) && (y == p.gety());
   else
        return false;
}
```

ou, ainda:

```
public boolean igual(Ponto p) {
   return (p == null) ? false : x == p.getx() &&
                                y == p.gety();
```

usando o operador condicional ternário **?:**.

Em primeiro lugar, a condição garante que apenas iremos comparar os valores da instância parâmetro com o receptor se o objecto parâmetro não tiver o valor `null`, ou seja, se `p != null`. Satisfeita esta condição, então poderemos comparar os valores das duas instâncias de `Ponto`, parâmetro e receptor, comparando os valores das respectivas coordenadas, o que corresponde à comparação de dois inteiros usando o respectivo operador de igualdade ==.

O método `copia()` deverá permitir criar e devolver uma cópia do objecto receptor da mensagem `copia()`, devendo ser original e cópia garantidamente objectos diferentes, ou seja, devendo-se garantir que, o original e a cópia **não são o mesmo objecto**, **são instâncias da mesma classe** e **têm o mesmo valor**.

O método `copia()` deve ter como assinatura `Ponto copia()`, vai ser usado em expressões como `Ponto p2 = p1.copia();` e tem o código seguinte:

```
// Cria um ponto que é uma cópia do receptor
public Ponto copia() {
  return new Ponto(x, y);
}
```

Utilizou-se o construtor passando-lhe os valores das variáveis de instância do receptor, garantindo assim que, qualquer que seja o receptor da mensagem `copia()`, uma nova instância de `Ponto` é criada contendo os mesmos valores do receptor, mas sendo distinta deste pois não partilha o seu endereço (*deep copy*).

Note-se que em JAVA a passagem de parâmetros para os métodos se faz de duas formas distintas: **por valor** (ou cópia) para os tipos simples e **por referência** (endereço) para os tipos referenciados (objectos ou *arrays*).

De notar que o resultado devolvido por um método de JAVA obedece igualmente a tal mecanismo, pelo que há que ter sempre cuidado com o que passamos para o exterior como resultado de um método. Se passarmos para o exterior como resultado uma variável de instância que é um objecto, o que estamos de facto a passar é um apontador para esse objecto, dando-se assim acesso para a sua manipulação a partir do exterior da classe. Logo, se assim procedermos, é como estarmos a enviar de "Tróia" para fora "cavalos de Tróia" para "entrarem em Tróia", sendo nós próprios a violar o encapsulamento. Que fazer em casos destes? Simples: criar uma cópia da variável de instância e **devolver como resultado a cópia realizada** (de facto, o seu endereço) **e nunca a variável de instância**.

Um outro método, que é muito usual associar-se a qualquer classe como método de instância, é um método que dá como resultado uma representação textual da instância, para, eventualmente, ser posteriormente apresentada em ecrã ou gravada num ficheiro de texto de forma formatada.

Este método, a que vamos dar o nome de `paraString()`, vai converter para caracteres cada uma das variáveis de instância, dar-lhes um formato legível e construir uma instância da classe `String` como resultado. Vejamos um exemplo com a classe `Ponto`:

```
public String paraString() {
   String s = new String();
   s = "Ponto(" + x + ", " + y + ")";
   return s;
}
```

ou, de forma mais simples mas equivalente,

```
public String paraString() {
     return new String("Pt(" + x + ", " + y + ")");
}
```

A classe `String` é também uma classe do *package* `java.lang` que fornece um conjunto de métodos para a manipulação de *strings* de caracteres, para além de possuir (herdados da linguagem C e de outras) um conjunto de operadores usuais, tais como o operador de concatenação de *strings* + usado acima. Este operador permite mesmo concatenar a uma *string* valores de tipos primitivos depois de os converter automaticamente para *strings* , como em `"Ponto(" + x` .

Objectos do tipo `String` podem ser criados não só usando `new String()` mas também, excepcionalmente, através de atribuição directa de um valor, como em:

```
String nome1, nome2, linhas;
nome1 = "Pedro Nuno";
nome2 = new String("Rita Isabel");
String linhas = nome1 + "\n" + nome2;
```

O teste de igualdade entre duas *strings* é realizado, naturalmente, através da comparação dos seus caracteres e não dos seus endereços (cf. ==), daí que nunca se deva confundir os dois casos. O método que verifica se duas

instâncias da classe `String` são iguais designa-se por `equals()`, recebe uma *string* como parâmetro e devolve um boleano:

```
boolean ok, iguais;
ok = nome1.equals(nome2);
iguais = nome1.equals(nome2) && nome2.equals(nome3);
```

As *strings* de JAVA possuem no entanto um grande inconveniente que em certas circunstâncias pode mesmo conduzir a grandes perdas de *performance* quando se programam aplicações de maior escala. As *strings* de JAVA, depois de criadas, são **imutáveis** (são entidades constantes em memória). Por isso, quando por exemplo escrevemos a frase `nome1 + " da Silva Costa"`, o que acontece é que uma terceira *string* vai ser criada em memória para guardar os caracteres resultantes da concatenação destas duas, e assim, sucessivamente, a cada concatenação ou atribuição do tipo `nome1 = nome1 + "\n"`.

Imagine-se, portanto, o esforço de alocação de memória que frases aparentemente simples como a que antes escrevemos no método `paraString()` implicam, e o consequente peso na eficiência dos programas. Havendo em todo o caso que usar o tipo `String` em múltiplas situações, apresentam-se na tabela seguinte alguns dos métodos mais úteis.

Método	Semântica
`char charAt(int i)`	Carácter no índice i
`int compareTo(String s)`	-1 menor, 0 igual, 1 maior que a *string* parâmetro
`boolean endsWith(String s)`	s é prefixo da receptora?
`boolean equals(String s)`	Igualdade de *strings*
`String String.valueOf(simples)`	Converte para *string*
`int length()`	Comprimento
`String substring(int i, int f)`	*Substring* entre i e f
`String toUpperCase()`	Converte para maiúsculas
`String toLowerCase()`	Converte para minúsculas
`String trim()`	Remove espaços (início e fim)

Tabela 3.3 – Alguns métodos da classe `String`

Salvo as raras excepções em que tal perda de *performance* seja negligenciável, ao longo deste livro utilizaremos as classes `StringBuilder` e `StringBuffer`, que criam instâncias de *strings* que são dinâmicas e disponibilizam métodos mais do que suficientes para a manipulação destas, sempre que tivermos que criar *strings* resultantes da junção de várias *substrings* (como no método `paraString()` e outros).

3.2.13 SOBRECARGA DE MÉTODOS: OVERLOADING

A **sobrecarga semântica** (*overloading*) de operadores, ou seja, a possibilidade de o mesmo operador possuir significados distintos em função das entidades sobre quem opera num dado momento (ex.: o que significa `x1 + x2` na maioria das normais linguagens de programação?) é uma daquelas idiossincrasias das linguagens, que são por vezes contestadas mas que até dão algum jeito ter disponíveis.

Em JAVA e na maioria das linguagens de PPO, existe também a possibilidade de se usar **sobrecarga semântica** de métodos (*method overloading*), que consiste no facto de se poderem definir na mesma classe dois ou mais métodos com o mesmo nome desde que tenham assinaturas diferentes. Por assinaturas diferentes dever-se-á entender uma lista de parâmetros diferente, seja quanto aos seus tipos de dados ou quanto à ordem dos mesmos.

Para além dos métodos de instância, os construtores estão sempre em sobrecarga dado serem obrigados a ter um nome igual ao da própria classe. Tal sobrecarga, como vimos, é no entanto muito útil pois são em geral múltiplas as formas como pretendemos criar instâncias de uma dada classe.

Relativamente aos métodos de instância, dado que a activação de cada método é realizada em função da estrutura da mensagem recebida, o nome, número, tipo e ordem dos parâmetros da mensagem serão a "chave" para a determinação do método a executar, pelo que o seu identificador é apenas um *token* de tal chave.

Assim, na classe `Ponto` anterior, para além dos dois métodos modificadores que incrementavam ambas as coordenadas, cf.

```
public void incCoord(int deltaX, int deltaY)
public void decCoord(int deltaX, int deltaY)
```

poderíamos acrescentar, ainda que sem vantagens especiais neste caso, os métodos que incrementam e decrementam X e Y de apenas 1 unidade, por exemplo

```
public void incCoord()
public void decCoord()
```

ou um `paraString()` em que o início do texto fosse dado como parâmetro:

```
public String paraString(String txtHeader)
```

Enfim, muitos outros exemplos poderiam ser encontrados, sendo certo que a sobrecarga de métodos é tanto mais útil quanto mais complexos e/ou interessantes forem as classes a desenvolver.

3.2.14 MÉTODOS COM NÚMERO VARIÁVEL DE PARÂMETROS

Muitas das vezes, a sobrecarga de métodos é realizada, apenas porque pretendemos ter métodos com o mesmo nome, com parâmetros do mesmo tipo, mas tendo um número diferente destes.

Consideremos que se pretende implementar um método que, dados dois, três, quatro ou mais números inteiros, e um factor multiplicativo que é um valor real, calcule o somatório das multiplicações. A primeira solução poderia ser criar métodos baseados em sobrecarga, que tratassem cada um dos casos, como em:

```
public double imposto(double taxa, int val1, int val2) {
   return val1*taxa + val2*taxa;
}
public double imposto(double taxa, int val1, int val2,
                      int val3) {
   return val1*taxa + val2*taxa + val3*taxa;
}
```

Esta solução é, porém, pouco versátil, pois obriga-nos a contar previamente quantos números vamos somar, para invocarmos o método adequado. Assim, o usual nestas circunstâncias seria criarmos um *array* de inteiros, preenchê-lo, e depois passá-lo como parâmetro para um único método que, recebendo o *array*, calculasse o somatório dos resultados dos produtos, como em:

```
public double imposto(double taxa, int[] vals) {
   double impTotal = 0.0;
   for(int i = 0; i < vals.length; i++)
      impTotal += vals[i]*taxa;
   return impTotal;
}
```

Em JAVA5, foram introduzidos métodos especiais que aceitam um **número variável de argumentos** (*varargs*). Neste métodos, que podem ter parâmetros normais, o parâmetro que pode ser substituído por um número indeterminado de argumentos é especificado escrevendo-se o seu tipo e nome, tal como em:

```
public imposto(double taxa, int... vals) {
```

Como nunca sabemos qual o efectivo número de valores que foram dados como argumentos, então teremos agora o problema de saber como percorrer todos os valores dados como argumentos do método. Ora, o ciclo *foreach* foi adaptado para contemplar este tipo de situação, permitindo percorrer automaticamente os argumentos.

No nosso exemplo teríamos então como código final:

```
public imposto(double taxa, int... vals) {
   double impTotal = 0.0;
   for(int val : vals) impTotal += val;
```

```
    return impTotal;
  }
```

A utilização deste método, apresentará as seguintes formas exemplo:

```
double calc1 = 0.0, calc2 = 0.0;
calc1 = imposto(0.45, 12, 34, 44, 25);
calc2 = imposto(0.21, 56, 34, 11, 23, 45, 2, 45, 67);
```

Os argumentos podem ser também de tipo referenciado, em cujo caso o seu tipo deverá ser o identificador da classe (ou compatível) dos argumentos. Por exemplo, poderíamos ter como argumentos instâncias de `Ponto`, num método com *varargs* para calcular o ponto com maior coordenada em X, como se mostra no código seguinte:

```
public Ponto maiorX(Ponto... pts) {
   Ponto max = new Ponto(-9999, -9999);
   for(Ponto p : pts)
      if (p.getx() > max.getx()) max = p;
   return max.copia();
}
```

O método poderia, então, ser agora utilizado da forma seguinte:

```
Ponto maxP1 = maiorX(new Ponto(-5,3), new Ponto(0,0));
Ponto maxP2 = maiorX(p1, p2, p3, p4, p5);
```

Completámos assim o primeiro ciclo de conhecimentos necessários à criação de classes em JAVA, ficando a conhecer alguns dos seus mais importantes membros, como as variáveis e os métodos de instância e como podemos impor, através dos modificadores de acesso, regras para a sua utilização.

Em seguida, vamos ver como é que as classes podem ser usadas na criação de outras classes, ou seja, como podemos criar classes reutilizando o código já desenvolvido em classes anteriormente definidas.

3.3 COMPOSIÇÃO NA DEFINIÇÃO DE CLASSES

A classe `Ponto` cuja definição nos tem servido de exemplo para a introdução de um conjunto de conceitos importantes, é, em si, uma classe muito simples, dado que a definição das variáveis de instância foi realizada recorrendo apenas a um tipo simples, no exemplo, o tipo `int`.

Possuindo JAVA, como sabemos já, um enorme conjunto de classes predefinidas às quais se podem juntar as nossas próprias classes, poderá agora perguntar-se de que forma umas e outras podem ser usadas na definição de uma nova classe.

Em JAVA, tal como em qualquer outra linguagem de PPO, existe um mecanismo básico e simples de uma classe poder usar na sua definição classes já definidas, o que corresponde a um primeiro mecanismo de reutilização, mecanismo que se designa por **composição** ou **agregação**. Este mecanismo de composição consiste na possibilidade de as variáveis de instância definidas numa classe poderem ser definidas como sendo de um tipo correspondente a classes já existentes, sendo depois manipuladas usando todos os métodos definidos nas suas classes tipo. Apenas teremos que conhecer as API e enviar as mensagens que activam os métodos compatíveis com os tipos de tais variáveis.

Naturalmente que, sendo este mecanismo bastante favorável, porque vem facilitar a **reutilização**, deve desde já também referir-se que uma utilização optimizada do mesmo implica um bom conhecimento de todas as classes existentes, não tanto das suas implementações, mas, seguramente, das suas API.

Até agora, o único relacionamento que vislumbramos entre classes foi um simples relacionamento do tipo **A invocou método de B**. Na classe `Ponto`, tivemos a necessidade de invocar métodos da classe `java.lang.Math`, pelo que, podemos afirmar que a nossa classe `Ponto` **usa** `Math`.

No entanto, o **mecanismo de composição** estabelece um relacionamento entre classes de natureza mais forte, que é uma **relação de inclusão**. Se a classe **A contém** (***has***) objectos da classe **B**, dado que algumas das suas variáveis de instância são instâncias do tipo **B**, então a classe **A** diz-se que **é composta por B**. De **B** dir-se-á que **é parte (da definição) de A** (*part-of*). Por exemplo, `Motor`, `Asa`, `Passageiro`, `Piloto`, etc., todas podem ser classes que poderão fazer parte da composição de uma classe `Aviao`.

Um exemplo simples desta composição (ou agregação) usando a classe `Ponto`, seria a criação de uma classe `Circulo`, cujas instâncias fossem definidas como tendo um centro dado por uma instância da classe `Ponto` e um raio dado por um `double`. Poderíamos ter a seguinte definição estrutural de tal classe:

```
public class Circulo {
   // construtores
. . . . . .
   // variáveis de instância
   private Ponto centro;
   private double raio;
.................
}
```

Vamos, agora, criar uma classe que defina segmentos de recta para tal contando com a nossa anteriormente definida classe `Ponto` para caracterizar a estrutura do segmento de recta, usando dois pontos delimitadores do segmento. Note-se, como primeira diferença relativamente à classe `Ponto`, que agora temos variáveis de instância que não são de tipos simples mas de tipos referenciados (objectos), pelo que as questões de inicialização dos seus valores são diferentes.

Vamos considerar, por hipótese, que criávamos a classe `Segmento` sem definir qualquer construtor. Será que podíamos criar instâncias desta classe? E com que valores seriam então as variáveis de instância destas inicializadas?

```
public class Segmento {
   // Variáveis de Instância
   private Ponto p1;
   private Ponto p2;
   ...
}
```

A resposta à primeira questão é sim, tal como vimos já anteriormente. Em JAVA, se uma classe for definida sem ter definido qualquer construtor, é-lhe acrescentado de imediato o construtor por omissão:

```
public NomeClasse() {…}
```

A questão seguinte é saber-se quais os valores iniciais que este construtor atribui às variáveis de instância. Vimos anteriormente que às variáveis de tipos simples lhes são atribuídos os valores predefinidos desses tipos (0 para `int`, 0.0 para `double`, `false` para `boolean`, etc.).

Em JAVA, as **variáveis de instância** de tipos referenciados são por omissão inicializadas com o valor **`null`** (Figura 3.3).

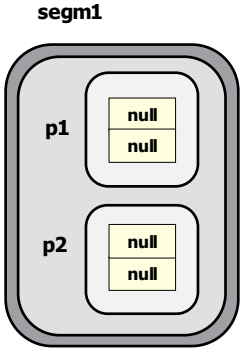

Figura 3.3 – Inicialização de variáveis de instância-objecto por omissão

No entanto, será sempre melhor prática que sejamos nós a definir construtores que atribuam valores iniciais não `null` às variáveis de instância das classes que criamos.

Teremos, então, para a classe `Segmento` os seguintes construtores:

```
public class Segmento {
   // Construtores
   public Segmento() {
      p1 = new Ponto(); p2 = new Ponto();
   }
   public Segmento(Ponto pa, Ponto pb) {
      p1 = pa; p2 = pb;
   }
   // Variáveis de Instância
   private Ponto p1;
   private Ponto p2;
   ...................
}
```

Para completarmos a classe `Segmento` vamos agora definir, para além dos métodos interrogadores e modificadores, e os métodos para igualdade, `copia()` e `paraString()`, métodos que realizem as seguintes operações:

- Dar como resultado o comprimento do segmento;
- Dar como resultado o declive do segmento;

O comprimento de um segmento de recta é dado pela expressão seguinte:

$$\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

da qual poderá resultar um número real, pelo que o método deverá devolver um resultado do tipo `double` e invocar a função matemática `sqrt()` da classe `Math` (conforme nos revelaria a consulta à API desta classe) para calcular a raiz quadrada da expressão que se apresenta acima e que é a soma do quadrado dos Δx com o quadrado dos Δy:

```
public double compSeg() {
   int dx = p1.getx() - p2.getx();
   int dy = p1.gety() - p2.gety();
   return Math.sqrt(dx*dx + dy*dy);
}
```

O método seguinte deverá calcular o declive m = (y2-y1)/(x2-x1) do segmento, pelo que o resultado deverá ser igualmente um valor real.

```
// determina o declive da recta
public double declive() {
   double dx = (double)(p1.getx() - p2.getx());
   double dy = (double)(p1.gety() - p2.gety());
   return dy/dx;
}
```

De notar que, sendo o resultado esperado um valor real e sendo as duas coordenadas valores inteiros, a sua diferença é um valor inteiro. A divisão (/) de dois inteiros dá como resultado um inteiro e não um real. Assim, devemos fazer o *casting* dos valores para `double (real)` antecipadamente à operação de divisão, para que se trate de uma divisão de dois números reais com resultado real.

Criámos, portanto, uma classe designada `Segmento` composta por duas variáveis de instância da classe `Ponto` e, respeitando a API definida em `Ponto`, trabalhámos com as respectivas variáveis como se de tipos "primitivos" se tratassem. A base do mecanismo de **composição** de classes é exactamente que aquilo que num dado momento parece estar a ser composto e, portanto, não era ainda primitivo, possa de seguida ser "material básico de construção" para o que houver a construir em seguida.

Criada a classe `Segmento` a partir de `Ponto`, podemos agora usá-la para definir novas classes contendo instâncias de `Segmento` como suas variáveis de instância.

Vamos, então, desenvolver um pequeno project, que, embora seja ainda do tipo monoclasse, se apresenta com maior número de requisitos e que tem por objectivo alicerçar em sede de projecto o conjunto de conhecimentos que foram até agora transmitidos ao longo dos pequenos exemplos apresentados.

3.4 EXEMPLO: PORTA-MOEDAS MULTIBANCO (PMMB)

Vamos agora desenvolver uma classe um pouco mais complexa, necessitando já de incorporar os conhecimentos até aqui adquiridos, aos quais juntaremos mais alguns.

Trata-se de desenvolver uma classe cujas instâncias são os pouco utilizados mas interessantes *Porta Moedas MultiBanco*, possuindo as características seguintes:

- Um PMMB tem um código alfanumérico único;
- Um PMMB tem gravado o nome do titular;
- Um PMMB ao ser criado tem saldo 0.0 €;
- A qualquer momento é possível carregar o PMMB com X €;
- Um pagamento apenas pode ser realizado se a importância a pagar não ultrapassar o valor do saldo;
- A qualquer momento deverá ser possível saber o saldo do PMMB;
- Deverá registar-se e poder consultar-se o número total de movimentos activos realizados, ou seja, carregamentos e pagamentos;
- Devem ser operações também disponíveis: igualdade de cartões (por número apenas), cópia e representação como `String`.

Vamos começar por especificar a estrutura interna do PMMB, definindo um tipo para cada uma das variáveis de instância: o código, que é até alfanumérico, será, como é usual, do tipo `String`; o nome do titular é igualmente uma `String`; o saldo do PMMB será um `double` já que o valor é um número real (ex.: €). O número de movimentos será um valor do tipo `int`.

Podemos, então, desde já, definir os construtores e as variáveis de instância:

```
public class PMMB {
   // Variáveis de Instância
   private String codigo;
   private String titular;
   private double saldo;
   private int numMovs;  // total de movimentos
   // Construtores
   public PMMB() {
      codigo = ""; titular = "";
      saldo = 0.0; numMovs = 0;
   }
   public PMMB(String codigo, String tit,
       double saldo, int numMovs)
      this.codigo = codigo; titular = tit;
      this.saldo = saldo; this.numMovs = numMovs;
   }
}
```

Vamos, em seguida, definir os métodos que vão ser tornados públicos pela classe `PMMB`. Em primeiro lugar, deveremos programar um conjunto de métodos de acesso às variáveis de instância PMMB, métodos que serão os seguintes:

```
public String getCodigo() { return codigo; }
public String getTitular() { return titular; }
public int getSaldo() { return saldo; }
public int getNumMovs() { return numMovs; }
```

Vamos agora escrever o código dos métodos modificadores do estado, sabendo-se que os carregamentos e os pagamentos devem não só alterar o saldo da conta mas também incrementar o número de movimentos da mesma:

```
public void mudaTitular(String novoTit) {
   titular = novoTit;
}

public void carregaPM(double valor) {
   saldo = saldo + valor;
   numMovs++;
}
```

3.4.1 MÉTODOS PARCIAIS E PRÉ-CONDIÇÕES

Deixámos propositadamente para o fim o método `pagamento`, dado apresentar uma característica particular que, por ser muito comum em Informática, merece uma análise especial, por forma a ser de futuro metodologicamente resolvida. De facto, a operação de pagamento de uma quantia é uma **operação condicionada**, ou seja, é uma operação que nem sempre pode ser realizada com sucesso, sendo por vezes impossível a sua realização. Qualquer que seja o PMMB, não deverá nunca ser possível realizar um pagamento de uma importância maior que o saldo actual do PMMB. Assim, apenas deveremos invocar a operação de pagamento numa situação em que a operação possa de facto ser realizada com sucesso. Para tal, deveremos realizar previamente um teste à condição que determina se a operação pode ou não ser invocada.

Em geral, sempre que uma dada operação não é **total**, isto é, é **parcial**, ou seja, não deve ser executada em certas condições, a sua codificação sem qualquer apoio metodológico torna o seu código muito pouco claro. Por um lado, porque se torna imperioso introduzir no próprio código o teste de tais condições de erro bem como o tratamento das mesmas, caso ocorram. Por outro lado, porque, em caso de erro, se torna necessário notificar o invocador de que a operação não pôde ser executada.

Uma "terrível", ainda que muito usual, solução é a seguinte:

```
public void pagamento(double valor) {
   if (saldo >= valor)
      saldo = saldo - valor;
   else
      System.out.println("Saldo Insuficiente!");
}
```

A solução parece pacífica, está bem estruturada, etc., mas vai contra vários dos princípios fundamentais da Engenharia de *Software*. Em primeiro lugar, o princípio da separação da camada computacional da camada de *input/output*. Se estas duas camadas se misturarem, quando uma muda a outra terá que mudar.

Por exemplo, o código do método anterior não funciona se a aplicação for transferida para uma interface gráfica. Por outro lado, e pela mesma ordem de razões, o princípio do encapsulamento, que atrás referimos como sendo a base de toda a PPO.

Um outro exemplo de uma "má codificação" deste método poderia ser o código seguinte (ou variantes idênticas só com um `return`), que também é bastante típico:

```
// má codificação de um método parcial
public boolean pagamento(double valor) {
   if (saldo >= valor) {
      saldo = saldo - valor;
      return true;
   }
   else
      return false;
}
```

Em primeiro lugar, o tipo do resultado, que havíamos definido como devendo ser `void`, já que se trata de um modificador do estado, passou a ter que ser `boolean`, dado que se pretende que, externamente ao método se possa

saber se a operação foi ou não bem sucedida. Claro que, do ponto de vista da clareza da assinatura do método em termos metodológicos, torna-se muito pouco claro porque é que um modificador do estado devolve um boleano. Por outro lado, do ponto de vista do código invocador deste método, algumas preocupações adicionais terão que ser tomadas. De facto, dado que o método devolve um boleano, não o poderemos invocar usando a que seria a expressão normal caso o resultado fosse `void`:

```
meuPMMB.pagamento(10.50);
```

mas antes uma série de expressões, tais como:

```
boolean ok = meuPMMB.pagamento(10.50);  // teste
if(ok)
   meuPMMB.pagamento(10.50);            // compra !
else
   System.out.println("Pagamento OFF !");
```

Claro que este código, dado possuir instruções de saída, apenas poderia ser código da classe de teste e não código de outro qualquer método, o que faz levantar a questão de se saber como trataria a situação de erro um método de instância de uma outra classe que tivesse invocado o método `pagamento`. Provavelmente, teria que alterar igualmente o seu tipo de resultado para poder comunicar a quem por sua vez o invocou, que não cumpriu totalmente os seus objectivos, isto é, que o serviço que lhe foi solicitado falhou.

A questão metodológica que está em causa é saber se é produzido melhor código quando, como no exemplo anterior, se invoca a operação em qualquer estado e, em seguida, se pergunta se tudo correu bem, e, caso não tenha corrido, se informa quem a invocou de que afinal não o deveria ter feito (cf. métodos terapêuticos), ou se, pelo contrário, se deve testar antecipadamente se a operação pode ser invocada e, só se tal for possível, esta é de facto invocada garantidamente com sucesso (cf. métodos profilácticos).

Nesta segunda abordagem, começa-se por definir a condição prévia à realização de tal operação, designada por **pré-condição**, que é implementada num método que deve ser `public` e que devolve como resultado um `boolean` após realizar o teste ao estado interno do objecto.

```
public boolean prePaga(double valor) {
   return saldo >= valor ;
}
```

A seguir programa-se o código do método total, isto é sem limitações, dado que se sabe que, quando for invocado, é porque a pré-condição se verifica (é `true`), e que quem o vai usar já deverá ter tido tal preocupação (como um contrato entre cliente e servidor).

```
public void pagamento(double valor) {
   saldo = saldo - valor;
   numMovs++;
}
```

Finalmente, no código invocador, haverá que perceber que a **pré-condição** deve ser testada antes de invocar o método, o que, admitindo que a variável `conta` é uma instância de `ContaBanc`, se traduz na seguinte estruturação:

```
if (meuPMMB.prePaga(val))
   meuPMMB.pagamento(val);   // realiza a operação
else
   System.out.println("Levantam. impossível !");
```

Se à pré-condição tivéssemos dado o nome sugestivo de `podePagar?(val)`, então poderíamos dizer que à melhoria da clareza do código do método invocado se tinha adicionado clareza no código do invocador. Não sendo rígida a atribuição de nomes às pré-condições, tal clareza é facilmente alcançável. Importante sim, é que estas pré-condições sejam declaradas como métodos públicos, e documentadas como sendo funções de teste para a correcta invocação dos respectivos métodos.

Até ao estudo do mecanismo de **excepções** de JAVA, esta será a metodologia de programação a adoptar para métodos parciais, isto é, métodos que apresentam restrições de funcionamento.

Vamos de seguida apresentar o código completo da classe `PMMB`, como sempre usando linhas de comentário para separar os vários membros da classe, designadamente, os construtores, variáveis de instância e métodos de instância. São adicionados à classe os três métodos complementares que devem sempre ser implementados numa classe, mesmo que tal não seja explicitamente pedido, designadamente, os métodos `igual()`, `copia()` e `paraString()`.

```
public class PMMB {
   // Construtores
   public PMMB() {
      codigo = ""; titular = "";
      saldo = 0.0; numMovs = 0;
   }
   public PMMB(String código, String tit,
               double sld, int nMovs) {
      this.codigo = codigo; titular = tit;
      saldo = sld; numMovs = nMovs;
   }
   // Variáveis de Instância
   private String codigo; private String titular;
   private double saldo;
   private int numMovs;     // total de movimentos
   // Métodos de Instância
   public String getCodigo() { return codigo; }
   public String getTitular() { return titular; }
   public double getSaldo() { return saldo; }
   public int getNumMovs() { return numMovs; }
   //
   public void mudaTitular(String novoTit) {
      titular = novoTit;
   }
   public void carregaPM(double valor) {
      saldo = saldo + valor; numMovs++;
   }
   public boolean prePaga(double valor) {
      return saldo >= valor ;
   }
   public void pagamento(double valor) {
      saldo = saldo - valor; numMovs++;
   }
   //
   public String paraString() {
      StringBuilder s = new StringBuilder();
      s.append("------- PMMB Nº: ");
      s.append(codigo);
      s.append("\nTitular: "); s.append(titular);
      s.append("\nSaldo Actual: "); s.append(saldo);
      s.append("\nMovimentos; "); s.append(numMovs);
      s.append("\n--------------------------------");
      return s.toString();
   }
   //
   public boolean igual(PMMB pm) {
      return codigo.equals(pm.getCodigo());
   }
   //
   public PMMB copia() {
      return new PMMB(codigo, titular, saldo, numMovs);
   }
}
```

3.5 MÉTODOS E VARIÁVEIS DE CLASSE

A definição de **classe** que foi apresentada neste capítulo, e foi até utilizada em exemplos concretos, está perfeitamente correcta, mas, no entanto, está incompleta. A razão é muito simples. Como dissemos atrás, quer as classes quer as instâncias que são criadas a partir das classes têm uma coisa em comum: **são objectos**.

As classes são objectos particulares dado que guardam a estrutura e o comportamento que vai ser comum a todas as instâncias a partir de si criadas. Porém, classes não deixam por isso de ser **objectos**.

Os objectos foram anteriormente definidos como sendo entidades que possuem uma estrutura de dados privada e um conjunto de métodos que representam o seu comportamento. Assim sendo, para que a definição de classe esteja coerente com o facto de classes também serem objectos, então uma classe deverá poder ter a sua própria estrutura de dados e os seus próprios métodos, para além de possuir uma definição das variáveis e métodos das suas instâncias.

Às variáveis que representam a estrutura interna de uma dada classe designaremos por **variáveis de classe**. Aos métodos que implementam o comportamento de uma classe designaremos por **métodos de classe**. Os métodos de classe são activados através das mensagens correspondentes **que deverão ser enviadas à classe**. Se uma classe possui variáveis de classe, tal como para as instâncias, o acesso a tais variáveis deverá apenas ser realizado através de métodos de classe de acesso a tais variáveis, mantendo-se o princípio do encapsulamento.

As variáveis de classe são, em certas circunstâncias, muito interessantes, dado que permitem guardar na classe informações que podem dizer respeito à globalidade das instâncias criadas e que não faria sentido colocar em qualquer outro local, ou seja, variáveis de classe são variáveis acessíveis à classe e a todas as suas instâncias.

Por exemplo, imaginemos que no conjunto de requisitos para a definição de uma classe `ContaBancaria` (não `Banco`, o que seria diferente) nos era solicitado o seguinte:

- Deverá ser possível possuir a cada momento o número total de contas criadas;
- Deverá ser possível possuir a cada momento o somatório dos saldos das contas existentes.

Uma variável que guarde o número total de contas já criadas não é certamente uma variável de instância, dado que nenhuma conta necessita de saber o total de contas já criadas. Admitindo, porém, que só poderíamos usar variáveis de instância para se guardar tal valor, que é de âmbito mais global do que o âmbito de uma instância, dado que diz respeito a todas as instâncias, então, tal variável de instância, por exemplo, `totcontas`, apareceria em todas as instâncias de `ContaBancaria` e se, por exemplo, já existissem 120 contas, em todas as instâncias deveria aparecer com o valor 120, o que seria uma perfeita redundância. Pior ainda, logo que uma nova conta fosse criada, 120 mensagens deveriam ser de imediato enviadas às 120 instâncias, dando a indicação de que agora passaria a ser 121.

O mesmo raciocínio se aplica à variável que deverá a cada momento conter o total dos saldos. Se um depósito ou levantamento fosse feito numa conta, uma mensagem teria que ser enviada a todas as outras para actualizar o saldo actual total: não teria sentido.

Por exemplo no caso da classe PMMB, poderíamos necessitar de uma variável que assegurasse que os códigos dos cartões emitidos fossem garantidamente sequenciais e que, portanto, nos obrigasse a saber a cada momento quantas instâncias de PMMB foram a cada momento já criadas, etc. Não colocaríamos tal valor em cada cartão, certamente.

Assim, as **variáveis de classe** tornam-se muito úteis para que nelas se possam guardar informações que dizem respeito à classe e/ou a todas as suas instâncias, tais como se exemplificou com os dois valores acima introduzidos relativamente a `ContaBancaria`. Tais valores assumem um carácter de "informação global" dentro do pequeno contexto classe e suas instâncias, sendo acessíveis e consultáveis através dos métodos de classe, em função dos modificadores de acessibilidade destes. As variáveis de classe estão exclusivamente associadas à classe, tendo existência real e podendo ser usadas mesmo se a classe não criou instâncias.

As variáveis de classe servem também, por vezes, para conter valores constantes, ou seja, imutáveis a partir da sua inicialização, funcionando pois como constantes de referência, por exemplo, para cálculos a realizar pelas instâncias da classe, cálculos que desta forma são tornados rigorosos e, mais do que isso, normalizados, dado serem garantidamente realizados por todas usando os mesmos valores. Por exemplo, se definirmos uma classe `Circulo` com um método de instância `area()` e pretendermos que o valor de **pi** para o cálculo da área seja sempre igual

para todos os círculos criados, então o melhor será criarmos uma "variável" de classe que define o valor a ser usado por todos os círculos no cálculo da área, em vez de termos um valor de **pi** em cada instância, correndo-se o risco de serem diferentes.

Os **métodos de classe** servirão, fundamentalmente, tal como os de instância, para garantir o acesso e a manipulação dos valores associados às variáveis de classe. Aos métodos de classe aplicam-se as mesmas regras anteriormente definidas para os métodos de instância. Os métodos de classe são sempre acessíveis às instâncias da classe, porém, métodos de classe não têm acesso a qualquer dos métodos de instância.

Quer os métodos quer as variáveis de classe distinguem-se dos de instância pelo facto de que, nas suas declarações, imediatamente a seguir ao modificador de acesso, aparece declarado o modificador **`static`**, que indica que, no contexto classe-instâncias, existe apenas uma cópia destas entidades, residindo tal cópia na classe.

Uma **Classe** passa assim a ser definitivamente definida como contendo:

- A definição de um conjunto de variáveis de classe;
- A definição de um conjunto de métodos de classe;
- A definição de um conjunto de variáveis de instância;
- A definição de um conjunto de métodos de instância.

A Figura 3.4 mostra a estrutura em geral usada para a especificação em abstracto de uma classe, ainda que a mesma não seja completamente rígida. Por vezes, as variáveis de instância são definidas antes dos construtores dado estes lhes atribuirem valores iniciais e, ao codificá-los, poder ser importante ter acesso visual aos nomes das variáveis.

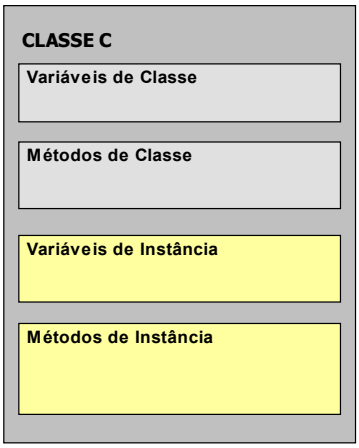

Figura 3. 4 – Estrutura completa de definição de uma classe

Vamos em seguida apresentar a forma de declaração JAVA das variáveis e dos métodos de classe, voltando ao exemplo concreto da classe `PMMB`, admitindo que pretendíamos adicionalmente contar o número de instâncias criadas e o saldo total dos cartões.

Assim, imediatamente a seguir ao cabeçalho da declaração de classe, devemos declarar as variáveis e métodos da classe que permitam realizar estes requisitos.

Vamos necessitar de duas variáveis e dos respectivos métodos de consulta e modificação das mesmas, pelo menos. Ambas as declarações são perfeitamente idênticas às declarações já estudadas para os métodos e variáveis de instância, excepto num único ponto: usam a palavra reservada **`static`**. Haverá assim uma única cópia destas entidades que é acessível a todas as instâncias da classe que forem criadas, o que corresponde excatamente ao que pretendêmos. Vejamos o novo código da classe `PMMB`:

```
public class PMMB {

  // Variáveis de Classe
  public static int numPMMB = 0;
  public static double saldoTotal = 0.0;
```

```
    // Métodos de Classe
    public static int getNumPMMB() {
       return numPMMB;
    }
    public static double getSaldoTotal() {
       return saldoTotal;
    }
    public static void incNumPMMB() {
       numPMMB++;
    }
    public static void actSaldoTotal(double valor) {
       saldoTotal += valor;
    }
    // Construtores
    public PMMB() {
       codigo = ""; titular = "";
       saldo = 0.0; numMovs = 0;
       this.actSaldoTotal(0);
       this.incNumPMMB();
    }
    public PMMB(String codigo, String tit,
            double sld, int nMovs) {
       this.codigo = codigo; titular = tit;
       saldo = sld; numMovs = nMovs;
       this.actSaldoTotal(saldo);
       this.incNumPMMB();
    }
    // Variáveis de Instância
    private String codigo;
    private String titular;
    private double saldo;
    private int numMovs;  // total de movimentos
    // Métodos de Instância
    . . . .
    public void carregaPM(double valor) {
       saldo = saldo + valor;
       numMovs++; actSaldoTotal(valor);
    }
    // pré-condição
    public boolean prePaga(double valor) {
       return saldo >= valor ;
    }

    public void pagamento(double valor) {
       saldo = saldo - valor;
       numMovs++; actSaldoTotal(-valor);
    }
    public String paraString() { . . . }
    . . . .
}
```

Note-se que métodos de classe devem ter sempre como objecto receptor uma classe. Tal apenas é opcional se forem usados dentro da própria classe. O acesso a variáveis de classe pode ser feito também usando o selector ponto (`.`), mas também aqui se aconselha o uso de métodos de consulta, tal como para as instâncias. A sintaxe é óbvia a partir do momento em que definimos que uma classe tem um identificador iniciado por uma letra maiúscula e uma variável por minúscula.

Assim, frases como as seguintes não provocam qualquer tipo de dúvida semântica.

```
pt1.incCoord(10, 5)     // método de instância
Ponto.x                 // variável de classe
ystem.out               // variável de classe
PMMB.incNumPMMB()       // método de classe
```

```
meuPMMB.getSaldo()      // método de instância
```

Dos quatro métodos de classe definidos, dois são simples interrogadores que dão como resultado os valores internos das variáveis de classe, devendo portanto ser usados em expressões que tenham como receptor o nome da classe, tais como:

```
int pMoedas = PMMB.getNumPMMB();
double saldos = PMMB.getSaldoTotal();
```

Outros são modificadores do estado das variáveis de classe, sendo pois usados em expressões da forma seguinte:

```
PMMB.incNumPMMB();       // ou apenas incNumPMMB();
PMMB.actSaldoTotal(valor);
```

Note-se ainda que, num construtor, a referência `this` tanto pode estar associada a um método de classe como a um método de instância. Num método de instância também pode, mas por questões de clareza e estilo não é aconselhável usar `this` associado a métodos de classe (conforme em `actSaldoTotal(-valor);`).

Sempre que pretendermos definir **constantes de classe**, isto é, valores imutáveis que são armazenados na classe e utilizáveis por qualquer instância, então, devemos adicionar o modificador **`final`** à declaração da "pseudovariável", deste modo dando a indicação de que se trata de uma constante e, em geral, por uma questão de estilo, usando apenas letras maiúsculas (e, por vezes, o _) no seu identificador, tal como em:

```
public static final double PI = 3.1415926535897932;
public static final int MAX_INT = 99999;
```

Possuindo tais identificadores valores por definição inalteráveis, e dado possuírem um estilo de declaração diferente das classes, das variáveis e dos métodos, e ainda dado que o compilador de JAVA ao reconhecê-los como **valores imutáveis** realiza a substituição imediata dos seus identificadores pelos respectivos valores, há neste caso toda a vantagem em explicitamente se identificar tais constantes, daí até a sua designação como *named constants.*

Assim, sendo por definição imutáveis, as constantes de classe podem ser usadas directamente sob a forma de expressões via selector ponto (**.**) tal como em:

```
double area = Circulo.PI * Math.pow(raio, 2);
int x = ClasseX.MAX_INT * 10;
int minInt = Integer.MIN_VALUE;  // constante da classe Integer
```

3.6 IMPORTAÇÃO ESTÁTICA

A partir de JAVA5, a linguagem passou a possuir também um mecanismo de **importação estática** que permite ao programador importar de uma dada classe um qualquer dos seus membros **`static`**, ou seja, variáveis ou métodos de classe, e, a partir desse momento, poder usá-los sem ter que usar como prefixo o nome da sua classe. As instruções de **`import static`** devem ser escritas no início dos ficheiros de definição das classes, tal como as cláusulas de importação até aqui estudadas.

Por exemplo, na classe `Ponto`, poderíamos então ter escrito:

```
import static java.lang.Math.*;
public class Ponto {
```

e, agora, ao longo do código dos métodos de instância de `Ponto`, em vez de escrevermos `Math.abs(x)`, `Math.pow(dx, 2)`, `Math.sqrt(dy)`, `Math.PI*5`, bastaria escrever `abs(x)`, `pow(dx, 2)`, `sqrt(dy)` e `PI*5`.

Se num programa principal escrevermos:

```
import static java.lang.System.out;
import static javax.swing.WindowConstants.*;
```

as nossas instruções de *output* podem passar a ter a forma mais simples:

```
out.println("xxxx");
```

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

3.7 CLASSES NÃO INSTANCIÁVEIS

A definição de uma classe é na programação por objectos o processo central de concepção das entidades computacionais que pretendemos construir. A classe permite que a partir dela sejam criadas tantas instâncias idênticas em estrutura e comportamento quantas as necessárias. Este é, portanto, o desiderato fundamental da criação e da utilização de classes. As classes são também tipos pois as variáveis são declaradas como sendo referências para objectos de dado tipo, ou seja, como sendo instâncias de dada classe.

Porém, em algumas circunstâncias especiais, pode fazer algum sentido a definição de classes que não vão poder criar quaisquer instâncias. Antes de nos debruçarmos sobre a utilidade de tais classes, que não vão poder criar quaisquer instâncias, vejamos em que condições uma classe não pode criar as suas próprias instâncias.

Os componentes de uma classe que têm directamente a ver com a criação correcta de instâncias dessa classe, são não só os construtores mas também as variáveis de instância e os métodos de instância. Ora se uma dada classe for definida como tendo apenas variáveis e métodos de classe, o que é perfeitamente possível em JAVA, então essa classe não especifica a estrutura e o comportamento de qualquer instância, pelo que estas não poderão ser criadas, nem tal faria qualquer sentido.

Assim sendo, isto é, não podendo tal classe criar instâncias, qual o seu interesse ou vantagem num ambiente de PPO? De facto, classes contendo apenas variáveis e métodos de classe podem ser auxiliares preciosos em PPO, já que, mesmo que não permitam criar instâncias suas, podem representar "centros de serviços", dado disponibilizarem métodos de classe que têm funcionalidades de grande utilidade, sem que haja necessidade de criar instâncias para que tal funcionalidade possa ser tornada acessível.

Esta noção de uma classe como um "centro de serviços" único, sem instâncias ou réplicas, pode ser ilustrada recorrendo, por exemplo, à classe de JAVA designada por `Math`, classe `final` que pertence ao *package* `java.lang`, classe que coloca ao dispor dos seus "clientes" um conjunto de "serviços matemáticos", como sejam cálculos trigonométricos, conversões entre graus e radianos, arredondamentos, cálculo de logaritmos, raízes quadradas e quadrados, entre outros.

Todos estes "serviços" são oferecidos por esta classe através de um conjunto de métodos de classe de tipo `public static`, invocáveis portanto a partir de qualquer outra classe, e até importáveis. Sendo métodos de classe, o utilizador de tais métodos apenas terá que ter em consideração o facto de que as respectivas mensagens deverão ser enviadas a uma classe usando o seu identificador, excepto se for realizada a sua importação estática usando a instrução **`import static`**.

Esta possibilidade de concentrar um conjunto de "serviços" numa única classe não deve ser erradamente confundida com o facto de que, em certas circunstâncias, e em função de certos requisitos de projecto, certas classes surgem como devendo ter, no máximo, uma única instância (multiplicidade 1). Ter, no máximo, uma instância de uma dada classe nada tem a ver com a criação de uma classe para a qual não faz sentido criar instâncias, dado que a razão da sua existência é oferecer um conjunto de "serviços" que não fazem sentido ser replicados.

JAVA possui algumas classes deste tipo, de grande utilidade no desenvolvimento de aplicações, sendo no entanto pouco comum que, em projectos concretos, surja a necessidade de se desenvolverem classes com estas características.

A classe `Math` que já anteriormente referimos é, naturalmente, uma delas. Uma outra classe deste tipo é a classe `Arrays` que implementa um grande conjunto de métodos (algoritmos) sobre *arrays* de tipos primitivos, dado que os *arrays*, ainda que sendo de tipo referenciado, não são objectos e, portanto, não têm a si associados quaisquer métodos. Esta classe complementa-os no que diz respeito ao comportamento.

Outras classes deste tipo são as classes `Beans`, `Arrays`, `Proxy`, `Collections`, etc. A classe `Collections`, por exemplo, complementa as colecções de objectos (que iremos estudar no capítulo 8), fornecendo métodos de classe para ordenação de listas, para determinação do máximo elemento de uma colecção, etc.

É também comum que a **classe de teste** que se desenvolve para testar um dado conjunto de classes, possua vários métodos de classe que são invocados a partir de `main()` e que ajudam a estruturar o código, facilitando a realização de vários testes. São métodos de classe já que, como é compreensível, não há qualquer objectivo em criar instâncias desta classe de teste, mas apenas estrutrar o código do método `main()`.

Vejamos um exemplo de um programa em que é lida uma sequência de números inteiros para um *array* e, usando métodos auxiliares de classe, determinamos o seu máximo e se faz a sua ordenação por ordem crescente.

```
import static java.lang.System.out;
import java.util.*;
public class ProgMainAuxs {
  public static final int DIM = 100; // dimensão máxima
  public static Scanner input = new Scanner(System.in);
  // método auxiliar para leitura do array
  public static int[] leArray(int num) {
      int[] nums = new int[num];
      int n = 0;
      for(int i = 0; i< num; i++) nums[i] = input.nextInt();
      return nums;
  }
  // método auxiliar que determina o máximo do array parâmetro
  public static int max(int[] nums, int total) {
      int max = Integer.MIN_VALUE;  // menor valor int
      for(int i = 0; i<total; i++)
         if (nums[i] > max) max = nums[i];
      return max;
  }
  // O programa principal, desenvolvido no método main()
  // vai agora invocar os métodos static auxiliares.
  public static void main(String[] args) {
     int[] arrayNum = new int[DIM];
     out.print("Total de números a ler: ");
     int dim = input.nextInt();
     arrayNum = leArray(dim);
     int maximo = max(arrayNum, dim);
     out.println("Máximo = " + maximo);
     Arrays.sort(arrayNum);
     out.println("Array Ordenado --");
     for(int i = 0; i<dim; i++) out.println(arrayNum[i]);
     int soma = 0;
     for(int n : arrayNum) soma += n;
     out.println("Somatório = " + soma);
  }
}
```

Neste programa usámos um método auxiliar para leitura do *array*, outro para calcular o seu máximo e usámos o método `sort(int[] a)` da classe `Arrays` para ordenar o *array* de inteiros. Desta forma, estruturámos melhor o código do método `main()`, que se torna mais simples e mais legível. Criou-se uma constante de classe `DIM` que define a dimensão máxima dos *arrays* a usar, e uma variável `input`, que é global a todos os métodos, e que se associa a uma instância de `Scanner` para leitura de valores (via `System.in`).

Estas classes são úteis na medida em que "prestam serviços". Porém, deve ter-se em atenção que programar usando métodos e variáveis de classe são excepções por utilidade. Não é programação por objectos, é, de facto, programação imperativa. Porém, os nossos programas principais, nos quais vamos criar instâncias de classes e realizar operações com elas, têm sempre esta configuração, ou seja, são uma classe de alto nível onde se programa um método `main()` e, eventualmente, alguns auxiliares, para criarmos os nossos objectos. Neste exemplo, não foram objectos mas *arrays*, que são de tipo referenciado como se pode comprovar pela instrução `return nums;` do método `leArray()`.

Outras classes não instanciáveis mas essas não possuindo sequer código, quer de métodos de classe quer de métodos de instância, serão estudadas posteriormente, classes que apesar de não conterem código, serão de grande utilidade, como veremos.

3.8 CLASSES WRAPPER: INTEGER, DOUBLE E OUTRAS

No exemplo anterior, no cálculo do máximo do *array* utilizou-se uma constante definida na classe `Integer`, escrevendo `Integer.MIN_VALUE`, que corresponde ao menor valor inteiro possível. As classes `Integer`, `Double`, `Float`, `Short`, `Long`, `Byte`, etc., são classes de JAVA que foram desenvolvidas para compatibilizar os tipos primitivos com o nível dos objectos, sendo designadas por *wrapper classes* porque as suas instâncias são "embrulhos", sob a forma de objectos, para valores de tipos primitivos.

As classes *wrapper* numéricas criam objectos que guardam valores de tipo `int`, `double`, `short`, `long`, `float` e `byte`, mas tais objectos têm a propriedade particular de serem **imutáveis**, pois não existem métodos de instância disponíveis para modificar os seus valores. Para que os seus valores sejam alterados deverão ser convertidos num valor de tipo simples, modificados e depois convertidos de novo num outro objecto.

Cada uma destas classes possui duas constantes de classe que definem o maior e o menor valor de cada tipo um dos tipos numérico primitivos (`MAX_VALUE` e `MIN_VALUE`). Cada uma destas classes possui também diversos construtores e métodos de classe para:

- Converter valores de tipos primitivos em objectos (ex.: `new Integer(int i)`, `new Double(double d)`, etc.);
- Converter um objecto de uma das classes *wrapper* num valor de tipo numérico (ex.: `Integer.intValue()`, `Double.doubleValue()`, etc.);
- Converter um objecto de uma dada classe num valor de um tipo primitivo compatível (ex.: `Integer.doubleValue()`, `Double.intValue()`, etc.);
- Criação de valores de um tipo numérico a partir da sua representação sob a forma de uma *string* (ex.: `int i = Integer.parseInt(String s)`);
- Criação de um objecto a partir de uma *string* que representa o seu valor numérico (ex.: `Double d = Double.valueOf(String s)`).

Estes métodos encontram-se definidos em cada uma das classes *wrapper*, todos com a mesma semântica e apenas variando o sufixo ou prefixo do seu identificador.

Vejamos alguns exemplos comuns de utilização usando a classe `Integer`, exemplos que são extensíveis a todas as outras classes (substituídos os nomes de alguns métodos):

```
Integer intg1 = new Integer(120);        // int -> Integer
int val1 = intg1.intValue();             // Integer -> int
val1 = val1 + 2 ; intg1 = new Integer(val1);
String strVal = "123";
int val2 = Integer.parseInt(strVal);     // String -> int
Integer intg2 = Integer.valueOf(strVal); // String -> Integer
double dbval = intg2.doubleValue();      // Integer -> double
int max = Integer.MIN_VALUE;
double maiorDouble = Double.MAX_VALUE;
```

3.9 SÍNTESE DO CAPÍTULO

Classes são um conceito fundamental em todo o paradigma da PPO. Poder-se-ia mesmo dizer que este é um "paradigma de concepção por classes" e "de computação por instâncias", dado serem de facto as classes as principais entidades que temos que conceber, por forma a termos as desejadas implementações finais construídas

através das múltiplas instâncias que aquelas criam e das mensagens que estas trocam dinamicamente entre si. Como quer classes quer instâncias são objectos, generaliza-se e designa-se o paradigma por **programação por objectos**.

As classes cumprem em PPO o duplo papel de conterem as definições comuns de estrutura e de comportamento de todas as suas instâncias, para além de serem, elas próprias, objectos, possuindo a sua própria estrutura e comportamento, que são definidos em variáveis e métodos de classe. O mecanismo de composição ou agregação permite que classes possam reutilizar classes já existentes, tornando a sua definição mais simples e favorecendo a reutilização de código.

Todas as instâncias são criadas a partir de classes existentes, sendo cada instância uma instância de uma e uma só classe. A classe, ou tipo, de uma dada instância pode ser mesmo determinada durante a execução do programa, através do envio da mensagem `getClass()`, o que por vezes é muito útil. Se a um qualquer objecto se enviar a sequência de mensagens `getClass().getName()` obteremos uma `String` que é o nome completo da sua respectiva classe (p.ex. `java.lang.StringBuilder`), ou, se usarmos `getSimpleName()`, o seu nome simples, `StringBuilder`. Estes comentários servem apenas para realçar o facto de que para além da parte estática, as entidades criadas por um programa JAVA possuem existência em tempo de execução, são elas próprias dinâmicas e podemos utilizar essa informação no código dos nossos programas para controlo da execução.

Toda a computação em PPO se baseia na criação de instâncias a partir das classes, no envio de mensagens que activam métodos de instância, na eventual recepção dos resultados de tais métodos, métodos que se definem quer usando tipos simples e instruções e operadores básicos, quer usando objectos e mensagens.

A acessibilidade aos métodos e variáveis de instância pode ser definida de forma explícita recorrendo aos modificadores de acessibilidade. Assim, cada membro de uma dada classe poderá possuir diferentes tipos de declarações de acessibilidade tais como `protected`, `public`, `private e package`. Na definição das classes, em particular no que diz respeito ao acesso do exterior às variáveis de instância, a obediência aos princípios do encapsulamento e da abstracção permite garantir o desenvolvimento de código modular e reutilizável.

Veremos em capítulos posteriores outras formas de relacionamento entre classes, e mecanismos que trarão ainda maior modularidade e extensibilidade à PPO em JAVA.