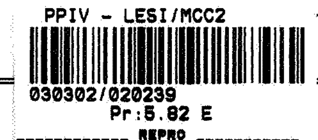
239



PARADIGMAS DE PROGRAMAÇÃO IV

2º ANO DA LESI
2º ANO DA LMCC

PROGRAMAÇÃO ORIENTADA AOS OBJECTOS USANDO A LINGUAGEM JAVA 2

NOTAS TEÓRICAS

Prof. F. Mário Martins

fmm@di.uminho.pt

Departamento de Informática
Universidade do Minho

2003/2004

PARADIGMAS DA PROGRAMAÇÃO IV

Departamento de Informática / Escola de Engenharia
Universidade do Minho

Ano Lectivo 2003/2004 (2º semestre)

Escolaridade

2 T + 2 TP + 0 P (3,5 créditos)

Cursos a que é leccionada:

Curso	Ano	Código
Engenharia de Sistemas e Informática	*2º*	530405
Matemática e Ciências da Computação	*2º*	7004N5

Responsável:

Prof. F. Mário Martins

Equipa Docente:

Docente	T	TP
Prof. F. Mário Martins	*2 (2 x 1h) LESI* *2 (2 x 1h) LMCC*	*2 (1 x 2h) MCC*
Prof. António Ramires Fernandes		*2 (1 x 2h) LESI* *2 (1 x 2h) LMCC*
Prof. António Nestor Ribeiro		*4 (2 x 2h) LESI*
Engº Sérgio Almeida		*4 (2 x 2h) LESI* *2 (1 x 2h) LMCC*

ESTRUTURA DE FUNCIONAMENTO:

Aulas teóricas onde são introduzidos os conceitos fundamentais deste paradigma tão importante e tão particular, em articulação com as aulas teórico-práticas que têm por objectivo consolidar tais noções à custa da exemplificação da sua utilização (ou até das consequências da sua falta de utilização) em programas escritos utilizando **JAVA** como linguagem de programação.

Estrutura e Objectivos Pedagógicos:

As aulas teóricas terão por objectivo a exposição da matéria fundamental que permita caracterizar o *paradigma da programação orientada aos objectos*.

Esta disciplina pretende realizar a transição entre a *programação em pequena escala* até aqui realizada nas disciplinas anteriores, onde se estudaram estruturas de dados mas onde as mesmas raramente foram aplicadas em aplicações de envergadura real, introduzindo-se agora todos os problemas inerentes ao desenvolvimento de aplicações de carácter e envergadura reais. Surgem assim, conceptual e pragmaticamente questões tais como a modularidade conceptual e de implementação, questões relacionadas com a correcção de erros e eficiência, e, ainda, com a própria facilidade de manter e estender as aplicações desenvolvidas. Procra mostrar-se como as metodologias orientadas aos objectos permitem satisfazer um elevado e importante número de princípios fundamentais da Engenharia de Software.

Pretende-se atingir uma boa compreensão da evolução histórica do paradigma, surgido como muitos outros nos anos 70, mas que só mais recentemente atingiu uma enorme projecção fora das universidades, das suas principais características identificadoras e dos conceitos formais que o consubstanciam e distinguem de outros.

A título de exemplo, e procurando introduzir certas novas noções a partir de outras noções introduzidas em anteriores disciplinas, o conceito de *objecto*, unidade modular fundamental ao paradigma, é associada às noções de módulo, abstracção de dados, protecção e encapsulamento tão importantes para a independência do software.

Não se pretende a este nível a apresentação formal de conceitos tais como objecto, classe, polimorfismo, hierarquia, herança, agregação, etc. Porém, sendo estes os conceitos fundamentais à compreensão não só da tecnologia mas também dos métodos de análise e concepção de software baseada no paradigma, em termos

pragmáticos, os mesmos deverão ser profundamente apresentados (segundo diferentes perspectivas da sua implementação até) e no final da disciplina perfeitamente assimilados.

Apresentar-se-ão igualmente alguns dos principais problemas típicos que se colocam na concepção de aplicações orientadas aos objectos, em particular os que têm a ver com decisões de classificação de novas classes desenvolvidas. Aqui procura-se uma clara distinção entre os dois principais tipos de herança (lógica ou física, simples ou múltipla) e ainda a distinção clara entre os mecanismos de herança e de agregação.

Ao longo da disciplina procurar-se-á sempre, ao nível das aulas teóricas, apresentar comparações entre as mais diversas maneiras como a actual tecnologia baseada no paradigma dos objectos implementa tais conceitos fundamentais. Apresentam-se assim as implementações realizadas em algumas linguagens de objectos mais usadas, em particular comparando a linguagem usada na disciplina, JAVA, com C++.

Numa primeira fase do curso, a preocupação centrar-se-á na construção da camada computacional de aplicações orientadas aos objectos em isolamento da camada interactiva. Na parte final do curso, se possível, abordar-se-á o modelo de interacção do JAVA.

As aulas teórico-práticas, todas realizadas em laboratórios, consistirão, desde o seu início, da apresentação sintética e estudo, sob a forma de resolução de pequenos problemas adequados, da linguagem JAVA, em particular das regras a cumprir para realizar PPO em JAVA, suas Classes e respectivos métodos.

Numa fase posterior, e admitindo um razoável conhecimento por parte dos alunos das principais classes e métodos, bem como de alguns conceitos introduzidos nas aulas teóricas, as aulas teórico-práticas procurarão exercitar questões relacionadas com a concepção de pequenas aplicações, nas quais o domínio dos mecanismos de herança e de agregação, de classificação, e de metodologia de concepção se torna importante.

As aulas teórico-práticas terminarão com a resolução de um ou dois exercícios onde o correcto uso de certas classes existentes em JAVA permitirá a construção final da desejada aplicação interactiva. A extensiva utilização do *mecanismo de excepções* de JAVA permitirá chamar a atenção dos alunos para os tão importantes aspectos da segurança e robustez do software, e, ao usar JAVA, tirar todo o potencial do muito claro e limpo mecanismo de tratamento de excepções.

São palavras e ideias chave da engenharia do software que importa reforçar como bem abordadas por este paradigma: a *modularidade* e o *encapsulamento* (pelo uso de objectos, classes e mensagens), a *flexibilidade* (via polimorfismo), a *classificação* (via hierarquia), a *reutilização* (via mecanismo de herança e agregação), a *extensibilidade* e a *generalidade* (via polimorfismo natural), entre outras.

Deve ainda ser realçada a *verticalidade* da utilização destes conceitos no projecto de software, dado que os mesmos podem ser aplicados desde a análise à implementação de sistemas seguindo o paradigma OO.

Síntese de Objectivos:

A disciplina de Paradigmas da Programação IV tem por objectivo completar a formação dos alunos na área da programação, pela introdução de outros modelos de programação existentes com grande capacidade na resolução de classes particulares de problemas. É objectivo da disciplina a apresentação do *Paradigma da Programação Orientada aos Objectos*, das suas bases formais, das suas capacidades específicas e das áreas da sua particular aplicação.

É particularmente importante que esta disciplina e este paradigma estabeleçam, ao nível do curso, a distinção entre a *programação em pequena escala* e os problemas da *programação em grande escala*, designadamente a adopção de técnicas de concepção e desenvolvimento modulares e escaláveis, em particular explorando todas as que o paradigma da programação por objectos oferece, neste caso particular usando as características da linguagem JAVA.

Compreendidas as potencialidades do paradigma para a resolução de certas classes de problemas, pretende-se, do ponto de vista prático, que os alunos se tornem auto-suficientes na escrita de aplicações em JAVA, e que adquiram o conhecimento genérico suficiente para que, posteriormente, e por si, possam, se tal for necessário, desenvolver as suas capacidades de utilização da linguagem e dos conceitos em disciplinas mais avançadas do curso, tais como Desenvolvimento de Sistemas de Informação, Técnicas Avançadas de Orientação aos Objectos, Bases de Dados Orientadas aos Objectos, Sistemas Multimédia, Sistemas Operativos, Criptografia, e, de forma geral, em quase todos os projectos de investigação.

Sistema de Avaliação:

A avaliação tem uma componente teórica e uma componente prática, ambas obrigatórias. Tal significa que um aluno que não obtenha a classificação mínima fixada para cada componente não será aprovado.

A **componente prática** consistirá da realização de **2 trabalhos práticos**, sob a forma de trabalho de grupo. A não realização dos trabalhos implica de imediato a reprovação do aluno à disciplina, passando a ser **não admitido** a exame, e considerado para efeitos estatísticos como **não avaliado**.

Para a componente prática a nota mínima deverá ser de **10 valores**, caso contrário o aluno é igualmente **não admitido**, ainda que seja estatisticamente considerado como **avaliado**.

Nas aulas teórico-práticas da disciplina são propostos e acompanhados pequenos exercícios, agrupados em fichas semanais teórico-práticas. Tais trabalhos pretendem servir de guião à componente teórico-prática da cadeira, e fios condutores do estudo dos alunos, tendo ainda como objectivo serem auxiliares à resolução do trabalho final. Tais trabalhos servirão também de base para muitas das questões que serão colocadas aos alunos nos exames que dizem respeito à avaliação da sua formação teórica.

A **nota teórica** será obtida através da realização de **1 teste individual escrito**, sendo a nota mínima necessária para a realização da componente teórica **9,5 valores**. Caso o aluno obtenha uma classificação entre 9 e 9,4, esta nota teórica, devidamente pesada, é considerada para efeitos de média com a nota prática, sendo-lhe no entanto descontado 1 valor à sua média final, devendo esta ser obviamente positiva após tal ajuste (>= 9,5).

A nota teórica deverá ser obtida em exame, desde que o aluno possua já nota prática. O exame poderá ser realizado numa das duas chamadas da época normal de Junho/Julho, ou na respectiva época de recurso (ou ainda na de Novembro para os alunos para tal habilitados).

A **nota final** da disciplina será encontrada, após satisfação das regras anteriores, e salvo o caso especial de nota teórica entre 9 e 9,4 anteriormente referido, pela aplicação da seguinte fórmula simples:

Nota Final = (Nota Teórica x 0, 55) + (Nota Prática x 0, 45)

O trabalho prático entregue pelos alunos de um dado grupo terá uma classificação que poderá, quando tal se justifique, ser individualizada. A não presença, injustificada, de um dos elementos de um dado grupo na apresentação e discussão do respectivo trabalho implica a sua não avaliação e consequente reprovação.

A classificação final do trabalho prático entregue, deverá ser calculada em função da seguinte escala de critérios e valores:

Escalão	Nota
Sem qualidade	6
Pouca qualidade	8/9
Qualidade mínima	10
Qualidade média	13
Bom trabalho	15
Muito bom trabalho	17
Trabalho excelente	18-20

A avaliação dos trabalhos terá em consideração diversas componentes, tais como:

- *pontualidade na entrega do trabalho;*
- *qualidade e complexidade das decisões de projecto;*
- *qualidade, em apresentação e síntese, do relatório apresentado;*
- *qualidade da documentação sobre o código fonte apresentado;*
- *qualidade do código fonte apresentado;*
- *qualidade da execução (com ou sem erros, satisfaz ou não requisitos, etc.);*
- *qualidade da apresentação ao utilizador (interface), caso tal se aplique;*
- *facilidade de utilização do programa sem ler manuais;*
- *nível da prestação oral dos elementos do grupo;*
- *agradabilidade geral do trabalho em todas as componentes (factor subjectivo);*

PARADIGMAS DE PROGRAMAÇÃO IV

CONTEÚDO PROGRAMÁTICO DETALHADO

PROGRAMAÇÃO POR OBJECTOS EM JAVA2

1. MATÉRIA TEÓRICA DE PPO:

1.1.- Introdução à Programação por Objectos.

- Origem do paradigma. Via Simulação. Via Computação.
- Conceitos básicos fundamentais.
- Modelos: de processos versus de objectos.
- A procura da modularidade no software.
- Independência do contexto como condição fundamental.
- Encapsulamento versus independência e modularidade.
- Modularização pelos dados: a solução em PPO.

1.2.- Noção de "Objecto" em PPO.

- Noção de "objecto" em PPO. Estrutura e Comportamento.
- Encapsulamento e protecção nos objectos.
- Interacção entre objectos. Mensagens vs. Métodos.
- Introdução ao Polimorfismo.
- Tipos de objectos: instâncias e classes.

1.3.- Classes, Hierarquia de Classes e Herança.

- Definição de Classe em PPO.
- Relação Classe-Instâncias. Introdução.
- Mecanismo de instanciação.
- Classes e sua Hierarquia. Superclassificação.
- Relações entre Classes. A herança.
- Herança lógica versus herança de implementação.
- Herança como mecanismo de reutilização e de programação incremental.
- Herança simples e múltipla.
- Algoritmo de procura de métodos.
- Herança versus Agregação.

1.4.- Classes e Herança.

- Criação de Classes.
- Classes "run-time" versus Classes para "compile-time".
- Tipos estáticos e dinâmicos das variáveis.
- Polimorfismo, "static" e "dynamic binding".
- Classes e Metaclasses.

1.5.- Classes Abstractas.

- Definição de Classe Abstracta. Importância das Classes Abstractas.
- Classes Abstractas vistas como Tipos Abstractos de Dados.
- Classes Abstractas como mecanismo de abstracção.
- Classes Abstractas como mecanismo de reutilização e de extensibilidade.
- Polimorfismo. Estudo dos diferentes tipos.

1.6.- Concepção de aplicações em PPO.

- Subclassificação e herança versus agregação.
- Subclasses como especializações.
- Subclasses para implementação.
- Algumas regras de concepção em PPO.

1.9.- Actuais principais aplicações do paradigma dos objectos.

- Linguagens e Tecnologia de interacção.
- Tecnologia de Sistemas Operativos e "plataformas".
- Metodologias OO de Análise e Concepção de Software (UML, Rational Rose, etc.

2. PROGRAMAÇÃO POR OBJECTOS EM JAVA: ESTUDO DA LINGUAGEM JAVA4 (JAVA 1.41)

2.1.- Programação por Objectos em JAVA.
Características do ambiente de desenvolvimento JDK.
A JVM ("Java Virtual Machine"). Byte-code.
Estrutura dos programas.
Bibliotecas. Packages.

2.2.- Tipos básicos (não objectos) e operadores.
Numéricos. Booleanos. Declarações.
Arrays Java e suas inconveniências vs. a classe Vector. A classe ARRAY como classe "servidora" de operações sobre "arrays".

2.3.- Estruturas de controlo.
Condicionais simples e compostas.
Estruturas Iterativas.

2.4.- Definição de Classes e Instâncias em JAVA.
Construtores. Métodos e variáveis de instância e de classe.
Tipos de qualificadores de visibilidade e acesso das variáveis e constantes.

2.5.- Hierarquia de Classes em JAVA.
Classe Object. Classes versus Packages.
Herança simples.
Redefinição e sobreposição de métodos e variáveis.
Classes e subclasses. Exemplos clássicos.
Compatibilidades entre instâncias de classes e subclasses.
O mecanismo de "dynamic type checking".

2.6.- Classes Abstractas em JAVA.
Declaração.
Polimorfismo e sua utilização. Regras da linguagem.
"Static-checking" vs. "Run-time checking" em JAVA.
Exemplos com classes abstractas.
"Casting".

2.7.- O mecanismo de Excepções da linguagem JAVA.
Cláusulas try, catch, finally, throws e throw.
Regras de utilização.

2.8.- *Interfaces* JAVA como especificações de Tipos de Dados.
Classes como subclasses e classes como subtipos. Análise aprofundada.
Herança múltipla de Interfaces em JAVA.
Regras para a implementação de Interfaces em Classes.

2.9.- Estudo das *Streams* de JAVA.
Streams de caracteres versus streams de bytes.

Streams de input e streams de output. As classes abstractas Writer e Reader.
Subclasses de Writer e Reader. Mecanismo de "aninhamento" de streams.
As ObjectStreams como mecanismo de persistência de dados. Serializable.
Exemplos de eficiência no uso de streams na gravação e leitura de 300.000
Fichas com a informação típica de um Aluno.
Comparação da eficiência das diversas soluções.

2.10.- Os Packages JAVA como mecanismos de Meta-Modularidade.
Estudo dos packages principais de JAVA.
Revisão das classes e interfaces fundamentais de JAVA, tais como Object, Number, Math, Array, Class, Collections (cf. Vector, Stack, Hashtable, HashMap, LinkedList), Number, StringTokenizer, Stream, Exception e outras, e das interfaces List, Map, Iterator, Enumeration, Cloneable e Serializable.
Importância das "inner classes" e de outras construções de JAVA como suporte efectivo a implementações genéricas de estruturas de dados.

BIBLIOGRAFIA

SOBRE O PARADIGMA DA PROGRAMAÇÃO POR OBJECTOS USANDO A LINGUAGEM JAVA2

Programação Orientada aos Objectos em JAVA2
F. Mário Martins, Editora FCA, Série Tecnologias de Informação,
ISBN-972-722-196-3, 1ª edição, Setembro de 2000, 4ª Edição, Fevereiro de 2004;

% Um livro que apresenta as características fundamentais do paradigma da PPO, e como tais características podem e devem ser implementadas usando, por exemplo, JAVA, para a criação metodológica de aplicações %

Paradigmas da Programação IV
Programação Orientada aos Objectos em JAVA
F. Mário Martins, Notas Pedagógicas, 2002.

% O conjunto dos apontamentos teóricos da disciplina, tal como apresentados nas aulas teóricas, aos quais se anexam diversos exemplos concretos de pequenos projectos de preparação. %

Object Oriented Design with Applications
G. Booch, The Benjamim Cummings Pub. Company, USA, 1991.

% Embora seja o livro onde o autor apresenta a sua metodologia para concepção de aplicações orientadas aos objectos, possui definições profundas dos principais conceitos existentes em PPO. Muito bom livro. No mínimo merece consulta. %

An Introduction to Object Oriented Programming
T. Budd, Addison-Wesley, 2nd Edition, 1997

% Livro de síntese dos principais conceitos do paradigma da PPO, e que apresenta um estudo comparativo sobre como tais conceitos são implementados nas mais relevantes linguagens de PPO, designadamente, JAVA, C++, ObjectPascal, Objective C, Eiffel, etc. %

SOBRE A LINGUAGEM JAVA
(não necessariamente bons para PPO)

Dominando o JAVA
P. Naughton, McGraw-Hill, 1996

% Livro em português do Brasil, que tem algum interesse apenas na fase inicial da aprendizagem de JAVA dado ser muito simples e muito limitado. Apresenta alguns erros graves e código fonte não correspondente aos resultados depois apresentados. Encontrar estes erros poderá ser um exercício interessante. Para os primeiros passos em JAVA, em particular para a sintaxe básica e familiarização com algumas classes vale a leitura dado não existirem muitas alternativas em simplicidade. %

Core Java
G. Cornell, C. Horstmann
Prentice-Hall, 1996.

% Livro bastante rigoroso e completo. Não acessível numa primeira fase do curso. Os autores usam muitas vezes classes não existentes no JDK1.1. É no entanto um verdadeiro livro de programação em JAVA, apresentando aplicações muito interessantes, podendo-se aprender bastante com o código apresentado.
Por exemplo, a programação com Streams e Applets, ainda que com problemas de compatibilidade com o JDK1.4, é ilustrada com base em aplicações com interesse %

Java in a Nutshell
D. Flanagan
O'Reilly & Associates, 3rd Edition, 1999

% A 3ª edição do livro anterior. Compatível com JDK1.4. Apresenta ainda algumas novidades (cf. classes anónimas, etc.) %

Teach Yourself Java in 21 Days
L. Lemay, C. Perkins
Sams.net, 1996

% Espera-se que quem o ler não fique a detestar Java em 21 dias !
Trata-se de um exemplo pela negativa. Não ensina PPO e apenas, de forma muito incompleta, JAVA.%

Data Structures in JAVA
T. Standish
Addison Wesley, 1998

% Livro bastante bom que ensina a implementar em Java as principais estruturas de dados usadas em programação, introduzindo ainda medidas simples de eficiência e complexidade .%

Data Structures & Problem Solving using JAVA
M. Weiss
Addison Wesley, 1998

% Livro interessante e complementar ao anterior. %

Notas das Aulas Teórico Práticas de PPIV (on-line na página de PPIV)
Mário Martins, UM/DI, 2002

% Todos os exercícios resolvidos nas aulas teórico-práticas por mim leccionadas %

Manuais ON-LINE, APIs, Fontes, etc. de JAVA 1.4.1/1.4.2
Página da disciplina: http://www.di.uminho.pt
Ver ainda: http://www.sun.com e
http://www.javasoft.com e http://www.bluej.com

INTRODUÇÃO AO PARADIGMA DA PPO. CONCEITOS FUNDAMENTAIS.

(PPO I)

☞ **Origens do paradigma da PPO.**

☞ **Conceitos Gerais em PPO.**

☞ O que é a PROGRAMAÇÃO POR OBJECTOS ?

✏ Os computadores são programados para simular sistemas físicos e abstractos complexos.

✏ Para desenhar, construir e comunicar estes sistemas programados a outros, ferramentas adequadas são, cada vez mais, necessárias.

✏ Um factor importante de adequação é, sem dúvida, a *proximidade* concedida por uma dada ferramenta ou modelo, entre ***o que*** se pretende modelar e ***como*** tal é modelado.

✏ A PPO é antes de mais um novo PARADIGMA ou MODELO de programação.

✏ Como todos os outros, propõe métodos e técnicas para auxiliar o programador a :

➡ dominar a COMPLEXIDADE dos PROBLEMAS ;

➡ escrever "boas" SOLUÇÕES (ie. PROGRAMAS).

✏ Apresenta porém características que a distinguem de qualquer outro paradigma convencional :

➡ **Objectos, Classes, Herança e Polimorfismo.**

➡ **Modularidade e Reutilização.**

➡ **Extensibilidade.**

➡ **Desenho e Programação Incremental.**

➡ **Desenho naturalmente "Bottom-Up".**

➡ **Proximidade entre modelo e modelado.**

✏ Assim, *satisfaz* e *suporta* os principais princípios que orientam a Engenharia de Software.

☞ Origens e Conceitos Fundamentais.

- A PPO tem por origem os trabalhos na área da SIMULAÇÃO, em particular, o desenvolvimento de MODELOS e MECANISMOS com vista à adequada representação dos FENÓMENOS FÍSICOS (ou seja do MUNDO REAL).

- Um programa é, nesta abordagem, um **modelo físico** que simula a estrutura e o comportamento de parte do mundo real ou imaginário.

- ENTIDADES do MUNDO REAL são, neste contexto, consideradas modeláveis, desde que sobre estas se possua a seguinte informação :

  - ➡ **Identidade** (e autonomia).
  - ➡ Propr. Estáticas (Atributos) : **Estrutura**.
  - ➡ Propr. Dinâmicas (Acções) : **Comportamento**.
  - ➡ Relação com outras entidades : **Interacção**.

- ENTIDADES individuais devem classificar-se, segundo um processo de abstracção, que delas reune as propriedades comuns e fundamentais em CONCEITOS.

- A linguagem SIMULA-67 [Dahl, Oslo, 67], foi a primeira a introduzir noções, que caracterizam completamente o modelo da ENTIDADE REAL. Tal representação da ENTIDADE REAL foi então designada por **OBJECTO**, sendo estes agrupados em **CLASSES** caracterizando as suas propriedades comuns.

```
CLASS Veiculo (peso, cargamax);
   real peso, cargamax
begin
      integer LIC_NUMERO
      real CARGA

      boolean PROCEDURE exc_peso;
          exc_peso := peso + CARGA > cargamax;
end;
```

- Em SIMULA-67, OBJECTOS representavam-se como "records" contendo campos ATRIBUTO (estrutura) e campos FUNÇÃO (comportamento). CLASSES são em SIMULA-67 associadas a TIPOS de OBJECTOS.

- Mais tarde, SMALLTALK [A. Kay, Xerox P.A., 70], [Goldberg, Xerox P.A., '80], reutiliza os conceitos anteriormente introduzidos, estendendo-os e utilizando-os na construção de um ambiente interactivo de programação verdadeiramente revolucionário.

- **Conceitos Fundamentais em PPO** :

  - OBJECTO.
  - CLASSE.
  - HERANÇA.

BASES DO PARADIGMA DA PPO

(PPO II)

☞ Modelos : Processos vs. Objectos.

☞ O que é um OBJECTO ?

☞ Estrutura e Comportamento.

☞ Interacção entre Objectos.

☞ Modelos : Processos vs. Objectos.

✏ Na programação convencional, seja ela *lógica*, *funcional* ou *imperativa*, **dados** e **operações** são vistos como entidades distintas e desligadas :

PASCAL :

```
type
     ponto =    record
                     cx : real;
                     cy : real
                end;
var
  p1, p2 : ponto;
  .....

procedure mkponto (x, y : real; VAR p : ponto);

procedure somap (p1, p2 : ponto; VAR p3 : ponto);
```

LISP :

```
(def x (list 1 2 3))

(def insere lambda (e lista)
      (cons e lista))
```

✏ Segundo estes modelos **programar** é :

Aplicar operações a dados transformando-os no sentido da obtenção de uma solução.

✏ Este modelo, originário dos primórdios da computação, é ainda visível aos mais diversos níveis :

MÁQUINA : Instruções + Dados.

LING. PROG. : Expressões + Variáveis.

PROGRAMA : Subrotinas + Argumentos.

LING. COM. : Comandos + Ficheiros.

✏ Segundo este modelo OPERADOR-OPERANDO :

➡ Procedimentos ACTIVOS, são aplicados a,

➡ Estruturas de Dados PASSIVAS,

➡ Sem ligação lógica entre si.

✏ Embora aparentemente independentes dos dados, os OPERADORES colocam enormes restrições aos TIPOS de OPERANDOS a que se aplicam.

- A compatibilidade entre OPERADORES e OPERANDOS é assim uma preocupação constante para o programador.

  procedure *desenhat*(t : TRIÂNGULO);

  procedure *desenhaq*(q : QUADRADO);

- As relações entre OPERANDOS e OPERADORES são do tipo ENTRADA-SAÍDA.

MODELO OPERADOR-OPERANDO

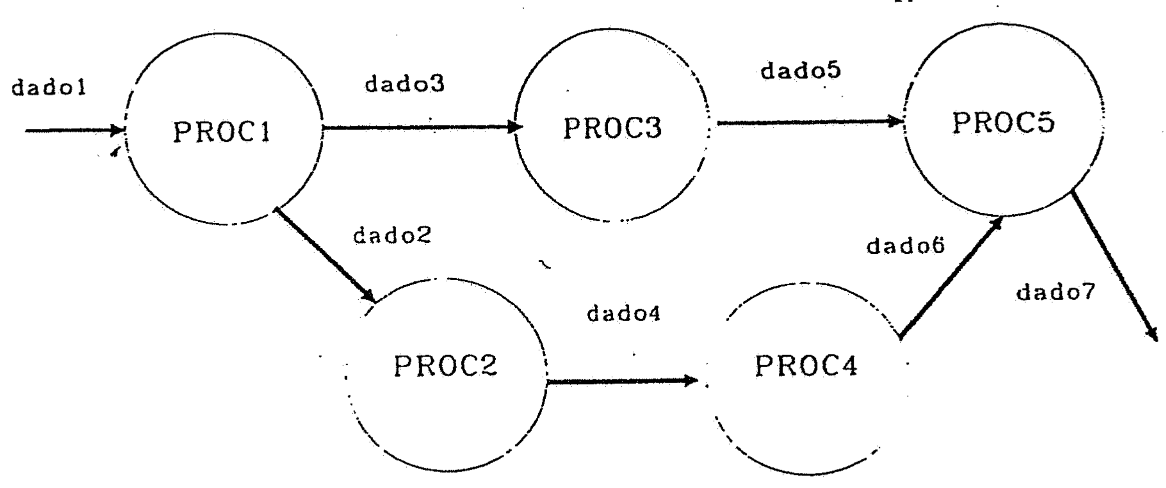


Fig. 1 : Modelo OPERADOR-OPERANDO

✏ Porém, devendo ser

PROGRAMAS = *MODELOS DE ENTIDADES REAIS*

este paradigma torna irreconhecíveis as ENTIDADES do mundo real e as suas TRANSFORMAÇÕES, dado basear-se na sua separação.

✏ EM RESUMO USAMOS :

- **MAUS MODELOS.**
- **FRACAS METODOLOGIAS.**

✏ EVOLUÇÃO :

➡ MODULARIZAR pelos DADOS.

- + Estabilidade de Dados ⇒ *Continuidade.*
- + Independência ⇒ *Reutilização.*
- + Encapsulamento ⇒ *Protecção*

➡ REPENSAR FERRAMENTAS.

➡ REPENSAR METODOLOGIAS.

✏ IDEIAS FUNDAMENTAIS :

➡ **TOP-DOWN** não satisfaz (*função de topo* ?).

➡ **TIPO ABSTRACTO DE DADOS** (a teoria).

➡ **MÓDULO = DADOS + OPERAÇÕES**.

- MÓDULOS → **BOTTOM-UP**
- MÓDULOS → **REUTILIZAÇÃO**.

➡ **OBJECTOS**.

ENTIDADES COMPUTACIONAIS GENÉRICAS

- **INFORMAÇÕES;**
- **TRANSFORMADORES DE INFORMAÇÕES;**

CARACTERIZAÇÃO :

- **FORMA ☞ SINTAXE;**
- **CONTEÚDO ☞ SEMÂNTICA**

MODELOS COMPUTACIONAIS ☚ PARADIGMAS

1.- IMPERATIVO

INFORMAÇÃO **TRANSFORMAÇÃO**
VALORES EM VARIÁVEIS **INSTRUÇÕES**

- PROGRAMA = SEQUÊNCIA DE INSTRUÇÕES
- COMPUTAÇÃO = SEQUÊNCIA DE ALTERAÇÕES DOS VALORES DAS VARIÁVEIS

2.- FUNCIONAL

INFORMAÇÃO **TRANSFORMAÇÃO**
VALORES identificados **FUNÇÕES**

- PROGRAMA = SEQUÊNCIA DE FUNÇÕES
- COMPUTAÇÃO = INVOCAÇÃO FUNCIONAL/RECURSIVIDADE/IF ..THEN

3.- LÓGICO

P :- Q, R, S.
Q :- X, Y.
Z.

INFORMAÇÃO — **FACTOS e REGRAS**

TRANSFORMAÇÃO — **VIA REGRAS**

- PROGRAMA = CONJUNTO DE PREDICADOS
- COMPUTAÇÃO = UNIFICAÇÃO + DEDUÇÃO + BACKTRACKING

4.- RELACIONAL

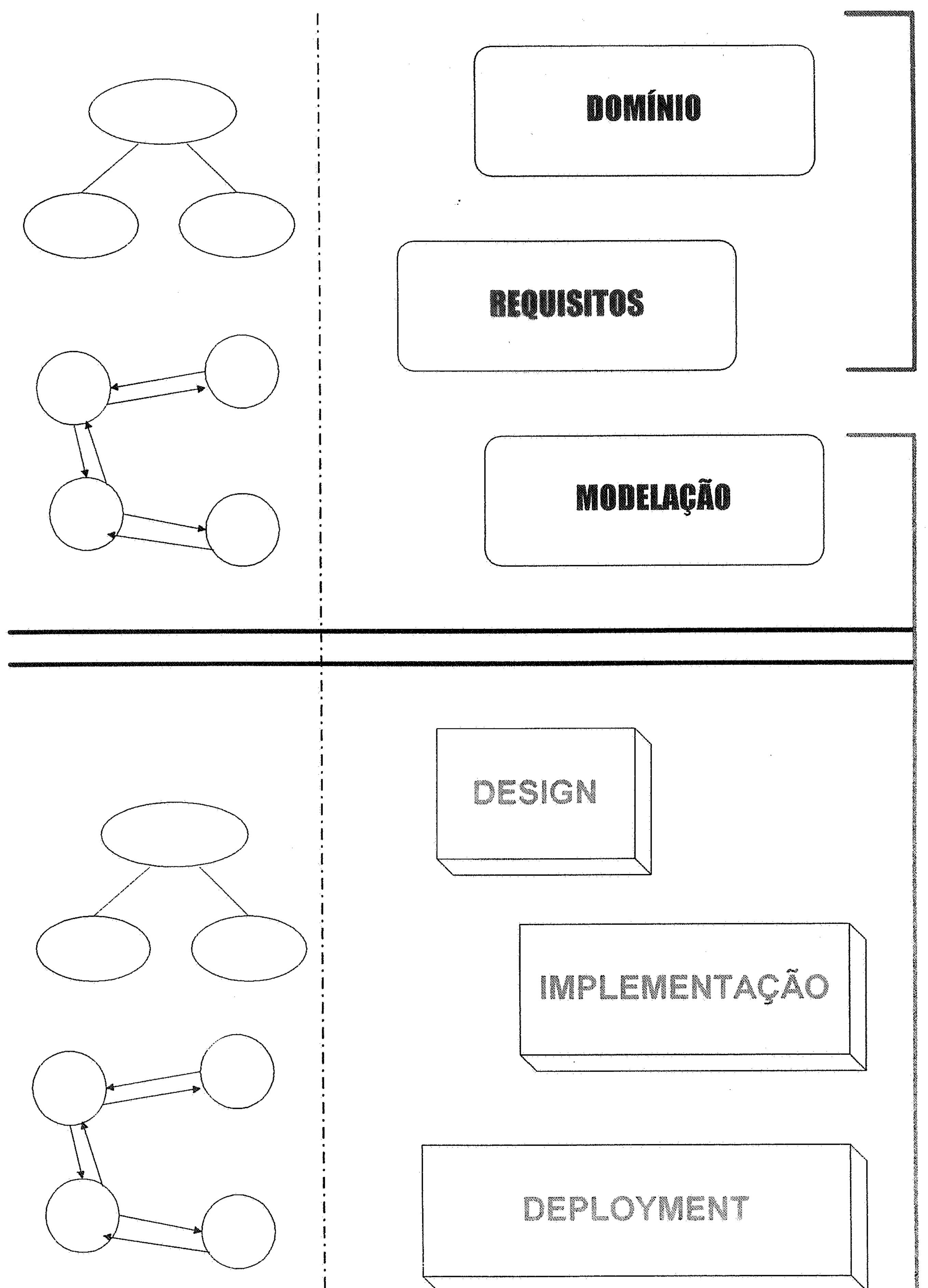
Num	Nome	Curso	Media

INFORMAÇÃO — **TABELAS**

TRANSFORMAÇÃO — **OPERAÇÕES COM TABELAS**

- PROGRAMA = CONJUNTO DE TRANSFORMAÇÕES EM TABELAS
- COMPUTAÇÃO = RELAÇÕES SOBRE TABELAS VIA LINGUAGEM DE MANIPULAÇÃO E CONSULTA (SQL)

A ENGENHARIA DO SOFTWARE



DESENVOLVIMENTO DE SOFTWARE
EM LARGA ESCALA

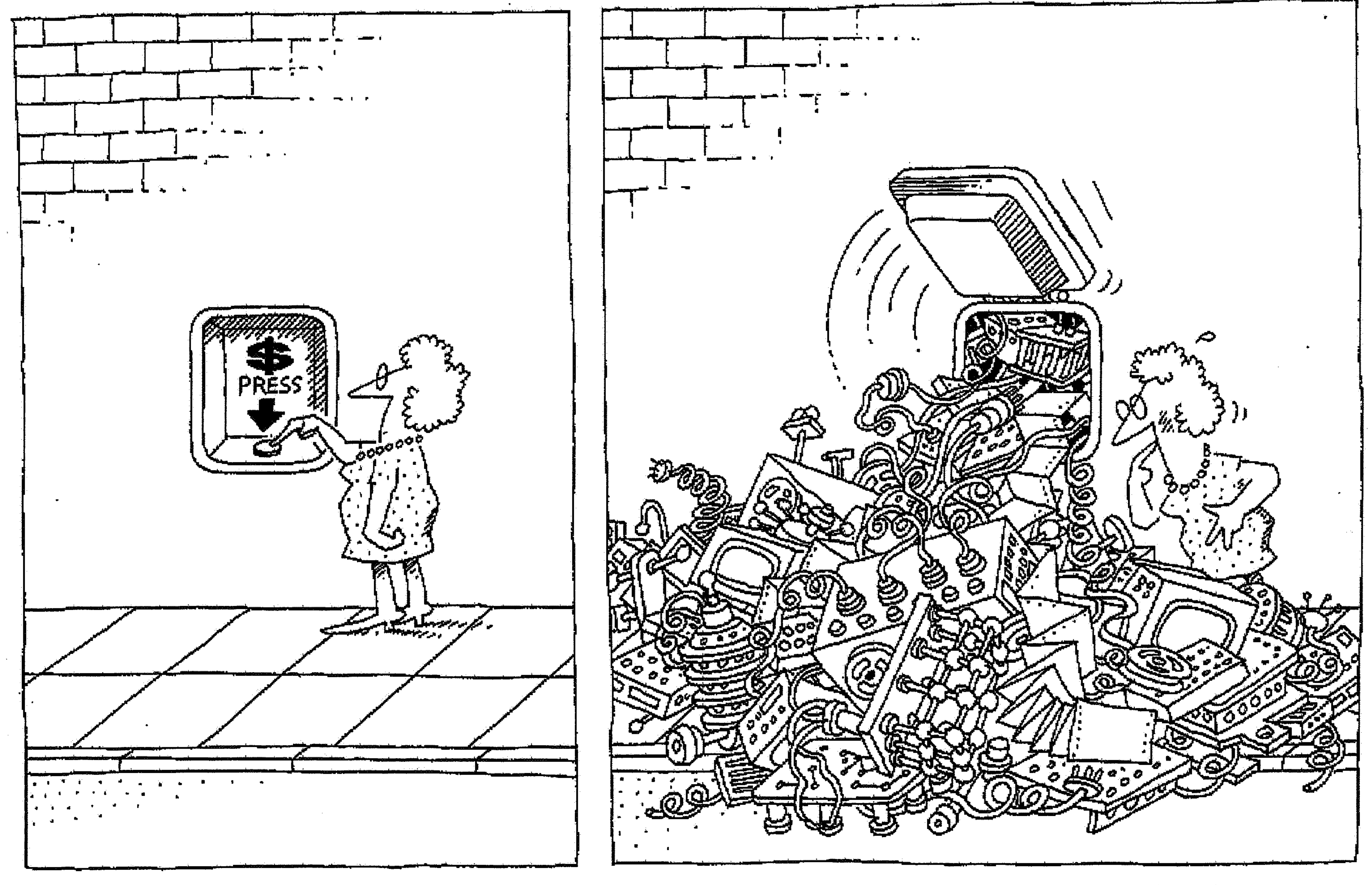
CONCEITOS FUNDAMENTAIS

- ☐ "DATA HIDING"
- ☐ "IMPLEMENTATION HIDING"
- ☐ ABSTRACÇÃO DE DADOS
- ☐ ENCAPSULAMENTO
- ☐ INDEPENDÊNCIA CONTEXTUAL

METODOLOGIA DE

DESENHO E PROGRAMAÇÃO

ORIENTADAS AOS OBJECTOS



The task of the software development team is to engineer the illusion of simplicity.

1.1 Origens do Paradigma

No seu livro *The Structure of Scientific Revolutions*, publicado em 1970, Thomas Kuhn, historiador da ciência, usa o termo *paradigma* associando-o a um conjunto de teorias e métodos que representam uma forma particular de se tentar organizar o conhecimento numa dada área. Revolução científica é, em tal contexto, entendida como uma substituição do paradigma estabelecido.

Robert Floyd, um dos grandes cientistas da computação, recebeu em 1979 o ACM Turing Award, tendo então proferido uma palestra intitulada "Os Paradigmas de Programação", na qual *paradigmas de programação* são definidos como modelos, ou exemplos, e abordagens organizacionais, que permitem conceptualizar o que se deve entender por computações, bem como se devem estruturar e organizar as tarefas que se pretendem ver realizadas por um computador.

Contrariamente ao que muitos pensam, o paradigma da *programação por objectos* ou *programação orientada aos objectos* (PPO), as duas traduções mais comuns da expressão original em língua inglesa *"object-oriented programming"*, não é um paradigma de programação nascido nos anos 90, sendo de facto muito mais antigo.

Este paradigma, que serve de base às *tecnologias de objectos* hoje em dia usadas em praticamente todas as áreas da Informática, tem na sua génese o conjunto de desenvolvimentos realizados ao longo de muitos anos em áreas do conhecimento bastante distintas, designadamente, a Simulação e a Engenharia de Software.

Pela via da Simulação, a maioria dos conceitos fundamentais ao paradigma da PPO surgem nos anos 60. Pela via da Engenharia do Software, surgem nos anos 70 as primeiras implementações de relevo tendo por base tal paradigma, ainda que, por várias razões, apenas nos anos 90 tenham de facto tido impacto tecnológico.

No entanto, curiosamente, alguns dos conceitos fundamentais do paradigma dos objectos, remontam à filosofia grega, em particular a Sócrates e ao seu discípulo Platão, nos séculos V e IV A.C., a propósito da forma ideal de realizar a catalogação ou classificação do conhecimento, já então visto como podendo ser dividido em *ideias individuais* e em *classes de ideias*. Estas duas correntes deram também origem a linguagens de PPO baseadas em modelos diferentes, tal como se irá referir mais adiante.

A VIA DA SIMULAÇÃO

De facto, a primeira linguagem a ser desenvolvida tendo por base a maior parte dos conceitos que iremos posteriormente estudar, e que são os conceitos principais que caracterizam e tornam diferente este paradigma de todos os outros, surgiu no final dos anos 60, foi desenvolvida por Dahl em Oslo e designava-se SIMULA-67.

Como o próprio nome indicia, a linguagem destinava-se a ser não propriamente uma linguagem de programação, mas antes uma linguagem na qual pudessem ser desenvolvidos modelos do mundo real e sobre estes executadas simulações.

O problema a resolver nesta área era o de encontrar formas simples de representar (modelizar) em computador, para mais fácil estudo, entidades do mundo real cujo *comportamento* se pretende analisar, sendo reconhecido que tais entidades possuem atributos definitórios próprios, isto é, valores associados a propriedades que representam a sua *estrutura* interna.

As entidades do mundo real são, nesse contexto, consideradas modeláveis, desde que sobre as mesmas se possuíssem as seguintes informações:

- *Identidade* (única);
- *Estrutura* (via atributos, ou seja as propriedades estáticas);
- *Comportamento* (as acções e reacções respectivas);
- *Interacção* (forma de relacionamento com outras entidades).

A estas entidades abstractas, modelos de entidades reais através de um processo de abstracção (simplificação), caracterizadas pela informação anteriormente indicada, a linguagem SIMULA-67 deu o nome de *objectos*. Dada a necessidade de modelar por vezes inúmeros objectos, diferindo apenas na sua identidade mas que em tudo o resto possuem características iguais, por exemplo, um dado número de pontos no plano 2D (logo todos com coordenadas X e Y) e com iguais comportamentos (por exemplo todos podem ser incrementados ou decrementados em X e Y), SIMULA-67 introduz a noção de *classe* como entidade geradora e agrupadora de todos os objectos que obedecem a um dado padrão comum, de estrutura e comportamento. Seguindo o exemplo, teríamos então que definir uma *Classe Ponto2D*, a partir da qual todos os pontos individuais poderiam ser criados, só desta forma podendo haver a garantia de que todos possuiriam a mesma estrutura e comportamento,

ainda que, naturalmente, possuindo valores diferentes nos seus atributos, para além de terem diferentes identificadores.

Os objectos assim definidos em SIMULA-67 eram considerados, do ponto de vista da simulação, como *entidades reactivas*, ou seja, entidades capazes de responder a solicitações exteriores realizando operações sobre o seu estado interno e enviando uma resposta à entidade solicitadora.

Sendo programas em computador simulações digitais quer de modelos conceptuais quer de modelos físicos, os resultados obtidos na área da Simulação usando tal abordagem foram sempre de grande interesse para a Engenharia de Software.

A VIA DA ENGENHARIA DO *SOFTWARE*

Nos anos 60 e 70, a Engenharia do Software havia adoptado um conjunto de princípios relativos ao processo de desenvolvimento de aplicações e construção de linguagens, genericamente designados como análise e programação *estruturadas e procedimentais*.

A ideia geral consistia em considerar-se que quer nas tarefas de análise quer nas tarefas de programação, os objectos da análise ou do desenvolvimento deveriam ser inicialmente entidades pouco definidas quer estrutural quer funcionalmente, muito abstractas, por forma a poder controlar-se a sua intrínseca complexidade. Em seguida, em passos sucessivos, cada um destes níveis de abstracção funcional, ou seja de processos (na análise) ou instruções (na programação), eram individual e separadamente refinados até se atingir o limite do seu detalhe.

O processo era designado por *Análise Estruturada* ou *Programação Estruturada*, conforme a fase do desenvolvimento e o produto final pretendido, tratando-se de facto de um processo em árvore de desenvolvimento de cima (nível mais abstracto) para baixo (nível mais concreto), por isso por muitos designado por metodologia "top-down".

Porém, uma das características importantes desta metodologia é que as entidades computacionais que eram objecto do refinamento sucessivo, ou seja, do processo "top-down", eram os processos e as instruções. Ou seja, quer na análise quer na

programação, não só os processos e instruções estavam completamente separados dos dados, como as grandes preocupações metodológicas se centravam apenas no correcto tratamento dos processos e instruções. Os dados eram em tal metodologia considerados como entidades de 2ª classe adaptados às necessidades funcionais.

Este modelo *estruturado funcional* e "top-down", em que as acções representavam as entidades computacionais de 1ª classe e os dados as entidades computacionais de 2ª, quer pelas suas virtudes quer pelas suas ineficiências, é um modelo muito importante dado que esteve na base de todos os grandes desenvolvimentos na área da Engenharia de Software dos últimos 25 anos, e também porque foi com base no mesmo que a Informática foi mundialmente sendo realizada nas organizações mais diversas.

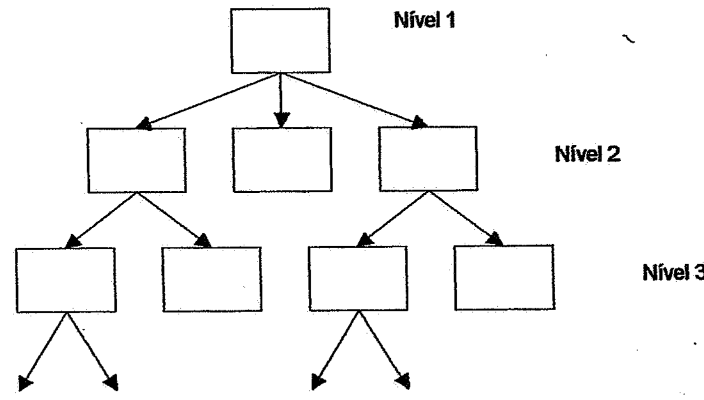


Fig. 1.1 – Refinamento progressivo dos processos

Como prova fundamental desta tendência orientada às instruções e aos algoritmos, Niklaus Wirth, que em 1971 havia concebido a linguagem PASCAL usando ideias fundamentais apresentadas na linguagem ALGOL de 1968, entre elas a necessidade de os dados serem rigorosamente associados a tipos verificáveis pelo compilador, usa em 1975 a linguagem PASCAL para apresentar, numa célebre obra prima da programação associada ao paradigma então em vigor, a fórmula em tal momento fundamental: *Algoritmos + Estruturas de Dados = Programas*. Note-se que os *Algoritmos* vêm primeiro e em segundo lugar surgem as *Estruturas de Dados*. Note-se ainda que a semântica do + nunca foi então claramente definida, ou seja, como deveria ser realizada estruturadamente a junção entre algoritmos e dados.

O mais importante de realçar aqui é o facto de que instruções e dados são vistas como entidades importantes, mas com tratamentos separados.

Note-se ainda que, em 1972, tendo por base o princípio da clara separação entre os dados e os procedimentos que os manipulam, Codd desenvolve o modelo teórico das Bases de Dados Relacionais, cuja posterior implementação nos anos 80 por várias companhias fez desaparecer os anteriores modelos hierárquicos e em rede para as bases de dados. Independentemente de todos os posteriores avanços nos modelos computacionais, a estruturação dos dados em bases de dados relacionais é ainda hoje, apesar do aparecimento de algumas bases de dados por objectos, a única forma generalizada de armazenamento de dados.

Estes princípios serviam igualmente de guião ao desenvolvimento das principais linguagens então largamente utilizadas, designadamente, o Basic, o Fortran e o Cobol. Estas linguagens eram caracterizadas por possuirem um conjunto de instruções com poucas formas de estruturação do controlo da execução, o que causava muita dificuldade de leitura e compreensão do código dos programas. Além disso, os programas eram muito inseguros por não existir qualquer *verificação estática* de tipos ("static type checking"). De facto, ao contrário do que se verifica actualmente com linguagens mais modernas como PASCAL, MODULA, C, C++, JAVA, etc., muitas variáveis não eram declaradas, isto é, associadas a tipos de dados. Não o sendo, o compilador não poderia realizar uma verificação estática de tipos, isto é, verificar em tempo de compilação – e não em tempo de execução – se todas as variáveis estavam a ser bem usadas no sentido de lhes serem associados os valores compatíveis com o seu tipo. Em algumas linguagens, o tipo simples associado a uma variável dependia da primeira letra do identificador da mesma.

De notar que muitas destas insuficiências, agora consideradas básicas, não podiam ter sido à época resolvidas, dado o escasso poder computacional e capacidade de memória da maioria dos processadores então existentes. De notar também, por outro lado, que é ao constatar certas insuficiências que são criadas exigências que conduzem ao desenvolvimento da tecnologia de base, sendo todo este processo um ciclo.

As primeiras tentativas de melhoria destas linguagens de programação surgiram naturalmente à medida que os projectos de *software* se iam tornando de maior

dimensão, onde passou a ser inaceitável, até pelos custos envolvidos, que todo o projecto de *software* tivesse que começar sempre "do zero", ou seja, que nada do que havia sido feito em projectos anteriores pudesse ser reaproveitado. Surge assim a noção muito importante de *reutilização de software*, ou seja, de garantir que qualquer programador deveria desenvolver código posteriormente reutilizável noutros projectos. Porém, nos anos 70, a única solução prática existente para garantir tal possibilidade, dada a inexistência de construções nas linguagens de programação que auxiliassem a tal objectivo, consistia em instruir e solicitar aos programadores que documentassem de forma clara o seu código, usando o que as linguagens de então aceitavam para documentação dos programas. Sentindo que o seu poder poderia estar em jogo nas organizações, os programadores adoptaram a prática contrária, ou seja, ou não documentavam programas ou os documentavam de forma dúbia, por forma a garantirem que as organizações, do ponto de vista informático, deles continuavam dependentes.

O constante aumento da complexidade dos problemas a resolver entretanto sentida por todos os envolvidos em Informática, conduziu à necessidade de criação de mecanismos de abstracção capazes de facilitar as tarefas quer de análise quer de programação, ainda que sob um ponto de vista *procedimental*, ou seja, procurando sempre responder à questão de como especificar a funcionalidade das aplicações de forma a que a mesma possa ser tão clara que possa ser reutilizada.

As linguagens de programação, ou seja as tecnologias de base da programação, acabaram por mais tarde dar resposta a todo este conjunto de exigências, não apenas desenvolvendo estruturas de controlo de execução dos programas muito mais compreensíveis – estruturadas - tais como as estruturas *for, repeat, while e case*, como também diferentes *mecanismos de abstracção de controlo* (ou de instruções) tais como as *subrotinas* ("subroutines"), as *funções* ("functions") e também os *procedimentos* ("procedures").

Estes primeiros *mecanismos de abstracção de controlo* implementam uma forma simples de *reutilização de software*, ao permitirem que um bloco de instruções seja escrito independentemente do programa principal, tendo a si associado um identificador único, pelo qual tal conjunto passa a ser referido (ou invocado). Em geral, este bloco de instruções é parametrizado, pois aceita parâmetros de entrada sobre os valores dos quais as instruções são executadas, podendo de tal execução serem produzidos resultados que são comunicados para o exterior.

A estrutura de um programa deixa assim de ser monolítica, passando cada vez mais a ser composta por um programa principal e por um conjunto de procedimentos e funções por este utilizáveis através de mecanismos simples de invocação.

A abstracção de controlo associada aos procedimentos, resulta do facto de que, tal como se pretende ilustrar na Fig. 1.2, para que os mesmos sejam utilizados não é necessário que se conheça o seu interior (o seu código), mas apenas a sua forma correcta de invocação e o resultado por este eventualmente devolvido. Ou seja, os procedimentos são vistos como *black boxes*, cujo interior é desconhecido, o que não impede a sua utilização desde que se compreendam as relações das entradas com as saídas, ou seja, a transformação realizada.

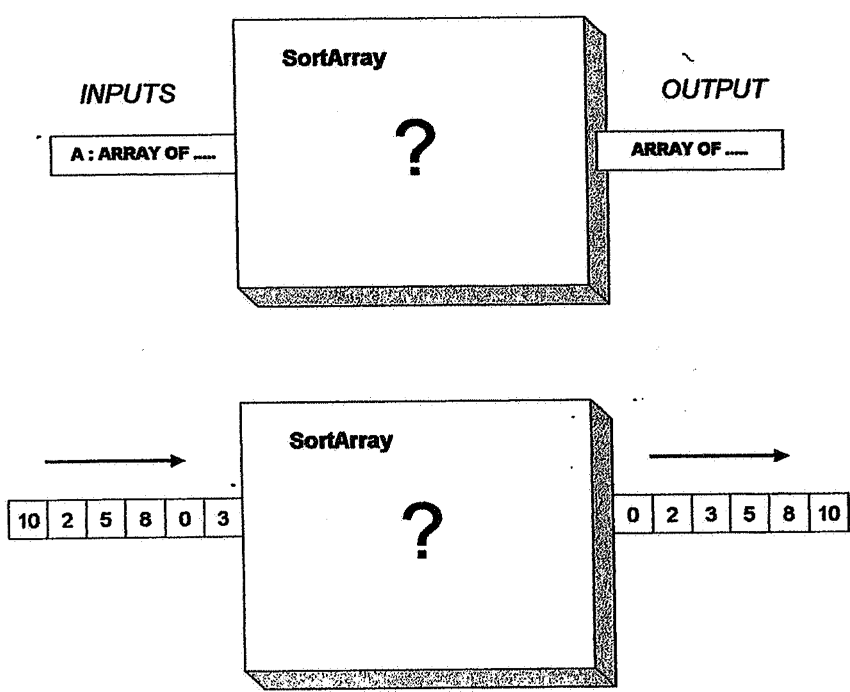


Fig. 1.2 – Abstracção de Controlo ou Procedimental

No entanto, estes simples mecanismos de abstracção de controlo que em muito vieram auxiliar à estruturação dos programas, possuiam ainda algumas restrições

que diminuiam a sua efectiva utilização. Por um lado, e porque não eram unidades de programação que pudessem ser compiladas separadamente do seu programa principal, ou seja, não sendo unidades de código independentes, a sua reutilização apenas podia ser realizada através de mecanismos de edição tipo "copy & paste". Por outro lado, por serem abstracções muito dependentes dos tipos de dados quer de entrada quer de saída, a sua utilização genérica era, desde logo, muito comprometida. Por exemplo, uma simples função Maior, que dados dois valores de entrada de um dado tipo dá como resultado o maior dos valores parâmetro, teria que ter tantas diferentes versões quantos os possíveis tipos de dados simples dos seus parâmetros de entrada. Assim, a solução seria criar uma função de nome diferente para cada caso, cf. MaiorInt, MaiorReal ou MaiorFloat, o que, como se compreende, se tornava pouco eficaz. Qualquer outra solução mais inteligente teria que ser completamente programada.

Admitindo adicionalmente que a linguagem de programação possa aceitar novos tipos de dados definidos pelo programador com base nos seus tipos primitivos, ou seja, admita alguma extensibilidade de tipos de dados, cf. PASCAL, MODULA, C e outras, nas quais, por exemplo, um tipo de dados Ponto2D é definido pelo programador como sendo representado como um registo com dois campos *x* e *y* de tipo inteiro, mais complexo se torna definir procedimentos (ou funções) genéricos.

Por outro lado, ao ser definido pelo programador um novo tipo de dados, havia que ter em atenção que, a menos que o programador definisse igualmente um conjunto de procedimentos e funções que implementassem as operações que com tal tipo de dados se pretendiam realizar, por exemplo operações básicas tais como a igualdade de valores do tipo, a comparação entre valores do tipo e outras, muito pouco se poderia fazer com os valores de tal tipo. Por exemplo, se o programador necessitasse de desenvolver uma *Stack* de inteiros, não só deveria criar uma representação da *Stack*, por exemplo usando um *array* de inteiros, como também programar os procedimentos de criação, "pop", "push", etc.. Se tal não fosse feito, a *Stack* teria uma representação mas não poderia ser manipulável.

De facto, e de um ponto de vista matemático, definir um tipo de dados com sendo apenas um conjunto de valores que variáveis de tal tipo podem assumir é redutor, dado que associa um tipo a um *conjunto matemático*. Definir um tipo de dados como tal conjunto de valores bem como todas as operações que sobre o mesmo se podem realizar, associa um tipo de dados a uma *álgebra*, noção matemática muito diferente da noção de conjunto. Esta distinção definicional trouxe às Ciências da

Computação conceitos novos, sendo a noção de *Abstract Data Type* ou *Tipo Abstracto de Dados (TAD)* uma das mais importantes, até porque, como veremos adiante, se liga directamente com algumas características da PPO.

Cronologicamente, o passo seguinte das linguagens ainda no sentido de melhorar os mecanismos de abstracção de controlo, foi o aparecimento em algumas delas, por exemplo em UCSD PASCAL e MODULA, de unidades de computação designadas por *módulos*. Módulos continham declarações de dados e declarações de diversos procedimentos e funções que podiam ser invocadas do exterior, mas possuíam a importante propriedade de poderem ser compilados isoladamente e posteriormente poderem ser "ligados" a qualquer programa que deles necessitasse. A abstracção continuava a ser de controlo ou de funcionalidade, mas os módulos facilitavam largamente a tão desejável *reutilização de código*.

Com este mecanismo, a maior parte das linguagens passou a ser fornecida com um enorme conjunto de módulos guardados em bibliotecas, que auxiliavam muito os programadores, já que, não sendo necessário conhecer o seu código, ofereciam uma funcionalidade por vezes de grande utilidade.

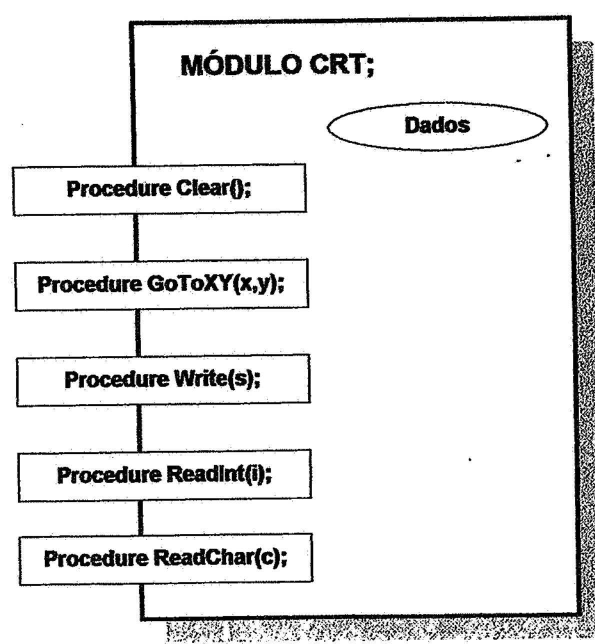


Fig. 1.3 – Módulo como Abstracção Procedimental

Por exemplo, um módulo designado CRT poderia oferecer todo um conjunto de procedimentos e funções para auxiliar o programador a escrever código simples de interface com o utilizador, e gerir as entradas e saídas de dados para o ecrã, apenas invocando os procedimentos disponibilizados pelo módulo e sem ter que saber como tal era de facto realizado. "Device drivers", módulos de gestão de ficheiros, módulos de tratamento de "strings", etc., passaram a ser as novas "black boxes" da programação, desta vez porém com a evidente vantagem de serem facilmente reutilizáveis por serem unidades de compilação em separado.

No entanto, e para além dos anteriormente referidos problemas relacionados com a falta de generalidade dos procedimentos e funções por serem "tipados", tal como mais uma vez pode ser visto no módulo CRT relativamente aos procedimentos de leitura, que devem ser especializados ao ponto de cada um ler um tipo de dados, o outro problema que conduziu à falência relativa deste modelo, que já permitia uma *programação modular, mas ainda orientada às instruções*, é que a utilização dos dados continuava a ser descurada, por ser considerado um problema de segunda ordem de importância.

Em resultado, os dados definidos num dado módulo poderiam ser globais a vários outros módulos, isto é, acessíveis a vários outros módulos como a figura seguinte procura ilustrar,ou seja, os procedimentos de um dado módulo poderiam ter acesso às estruturas de dados de outros módulos utilizados pelo programa.

Porém, admitindo tais acessos, um grave problema surge então. Se o objectivo dos módulos é encontrarmos unidades de programação que sejam independentes e reutilizáveis em qualquer contexto, isto é em diferentes programas, então tais módulos deveriam ser construídos em completa independência de todos os outros, ou seja, sendo completamente autónomos e, portanto, não devendo construir a sua funcionalidade à custa do acesso directo às variáveis de outros módulos, apenas realizando computações que utilizam unicamente o seu estado interno.

De facto, se um módulo A depende das variáveis de um módulo B e, por sua vez, B depende de outros módulos, então onde quer que se necessite de A, o módulo B e todos os de que B depende e assim sucessivamente de forma transitiva, teriam que existir para que A pudesse ser compilado e usado. A complexidade de controlar todas estas dependências seria para o programador uma tarefa impensável.

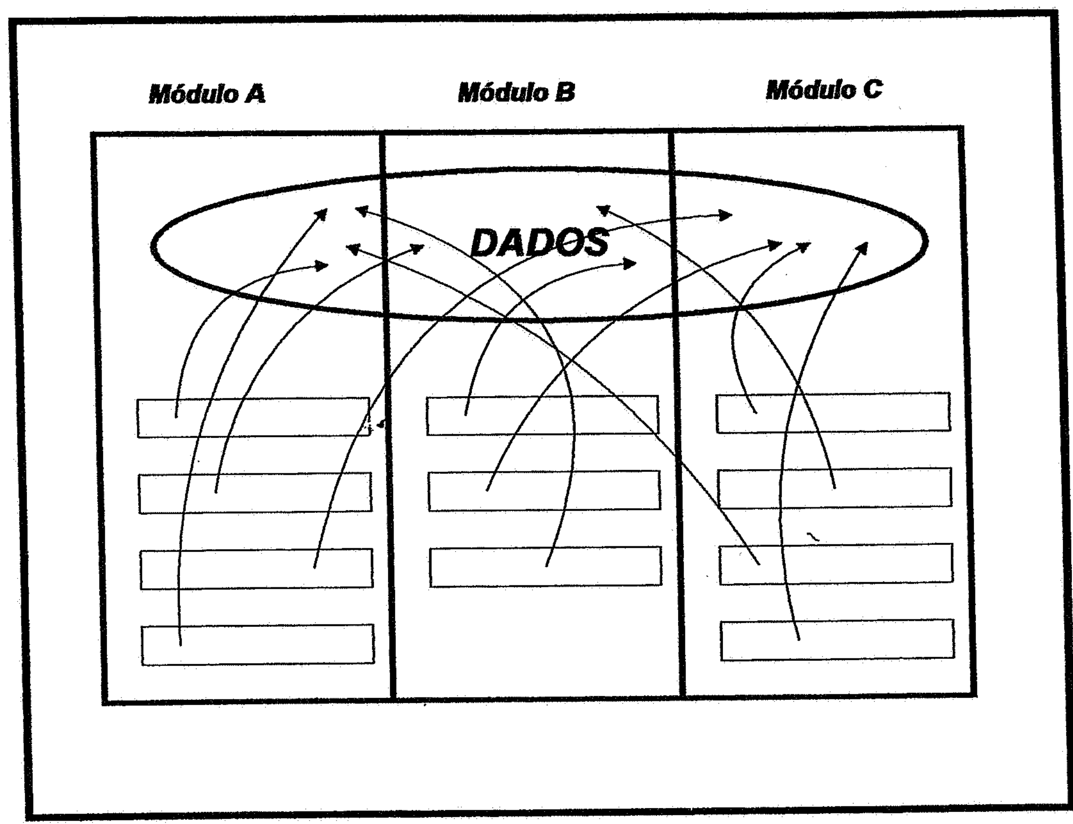


Fig. 1.4 – Módulos Interdependentes

Assim, a única forma de se garantir que um módulo é completamente independente do contexto, ou seja utilizável em qualquer programa, será torná-lo independente de todo e qualquer outro módulo, ou seja, tornando-o absolutamente autónomo. Para tal haverá que garantir que os seus procedimentos apenas acedem às variáveis que são locais ao módulo e, adicionalmente, que no código dos procedimentos não existem instruções de *input/output*.

Com base em tais requisitos, os módulos passam a ser vistos como definindo uma *estrutura de dados interna* e contendo um *conjunto de procedimentos e funções que devem ser o único código com acesso às suas variáveis internas*, sendo igualmente de garantir que tais funções e procedimentos não acedem a quaisquer outras variáveis que não façam parte dos dados locais do módulo.

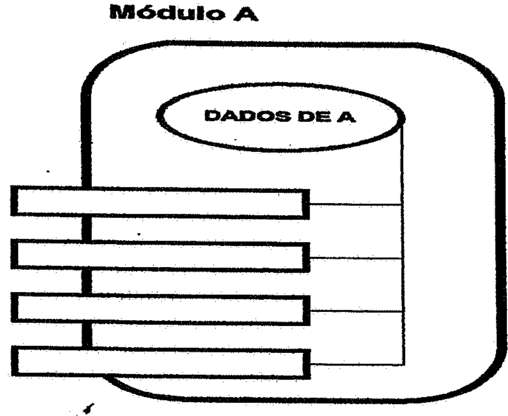


Fig. 1.5 – Módulo independente do contexto

Deste modo, a entidade fundamental de um módulo, que passa a estar protegida e "escondida" e que, idealmente, deve ser inacessível do exterior, é a estrutura de dados local ao módulo. Os procedimentos e funções passam a representar o papel de "serviços" disponibilizados pelo módulo para que do exterior se possa aceder à estrutura de dados. Assim sendo, os módulos passam a ser vistos finalmente como *mecanismos de abstracção de dados* e não de abstracção de controlo.

Se todos os módulos forem programados tendo por base estes princípios, o que apenas dependerá dos programadores dado que em geral as linguagens não fazem qualquer tipo de verificação e validação destas regras, então os módulos passam a ser não só unidades de programação completamente autónomas, portanto de facto independentes do contexto onde vão ser usados e assim sempre reutilizáveis, como ainda passam a poder ser vistos como "cápsulas", no sentido em que "escondem" ao exterior detalhes de implementação quer de dados quer de procedimentos. No entanto, o acesso do exterior à funcionalidade do módulo é garantida dado que o módulo declara como públicos, isto é, invocáveis do seu exterior, um conjunto de procedimentos e funções. A este conjunto de procedimentos e funções invocáveis do exterior de um módulo dá-se em geral o nome de *interface* ou até de API, do inglês *application programmer's interface*. Estes deverão ser, de facto, dentro deste conjunto de regras que temos procurado estabelecer com vista à obtenção de objectivos muito importantes no âmbito da programação, os únicos mecanismos de acesso ao módulo. Assim, um módulo passa a ser uma *abstracção de dados* que se pode dividir em duas camadas: *uma interface* e *uma implementação*.

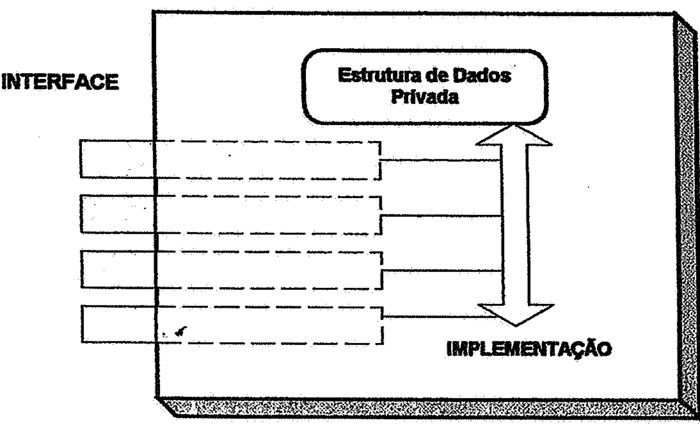


Fig. 1.6 – Módulo como Abstracção de Dados

Porém, este conjunto de regras e princípios conduziu a uma perspectiva diferente do que deveria ser um módulo. De facto, e em conformidade com a noção de *Tipo Abstracto de Dados* desenvolvida nos anos 70, um módulo pode ser visto como uma abstracção não de controlo mas de dados, caso a entidade fundamental que o mesmo implementa seja um *tipo de dados definido pelo programador*, no qual os procedimentos e as funções são os respectivos operadores sobre tal tipo.

Esta visão de um *módulo de software* como sendo uma *cápsula*, ou seja, uma entidade apenas acessível do exterior pelo que de si define como sendo *público*, ou seja os procedimentos que fazem parte da sua *interface*, necessita agora de ser analisada com base em exemplos concretos, para que melhor se compreendam tais vantagens.

Nesta perspectiva, e admitindo sempre uma estreita obediência às regras definidas, por exemplo um módulo de implementação do tipo de dados Ponto2D deveria ter definida uma representação privada, não visível do exterior, para pontos 2D, e um conjunto de procedimentos e funções disponibilizados na sua interface para que do exterior do módulo possam ser criados e manipulados pontos do plano 2D.

Teríamos um módulo que, programado numa linguagem de programação que é fictícia, mas que se pretende seja compreensível, apresentaria o seguinte código:

```
MODULE PONTO2D;

TYPE
    Ponto2D  = RECORD
                  x: INTEGER;
                  y: INTEGER
               END;

(* Procedimentos e Funções *)

PROCEDURE criaPonto2D(x1: INTEGER;
                     y1: INTEGER; VAR p: Ponto2D);
BEGIN  p.x := x1; p.y := y1  END;

PROCEDURE getX(pt: Ponto2D) : INTEGER;
RETURN pt.x;

PROCEDURE getY(pt: Ponto2D) : INTEGER;
RETURN pt.y;

PROCEDURE incX(VAR pt: Ponto) ;
BEGIN pt.x := pt.x + 1 END;

PROCEDURE incY(VAR pt: Ponto) ;
BEGIN pt.y := pt.y + 1 END;

END MODULE;
```

Note-se que o módulo Ponto 2D fornece procedimentos particulares para que do exterior se possa consultar o estado interno de cada Ponto2D criado, bem como procedimentos que implementam a funcionalidade de alterar as suas coordenadas.

Torna-se agora fundamental compreender, com base neste exemplo, não apenas a importância do encapsulamento, mas também a importância de certas regras serem

estritamente obedecidas por quem programa. Em tal sentido vamos considerar dois exemplos distintos de utilização do módulo Ponto2D e verificar o que acontece em cada caso perante uma situação de alteração do módulo.

Consideremos um primeiro programa obedecendo rigorosamente às regras básicas de utilização de um módulo, ou seja, fazendo acesso ao módulo usando apenas o que este exporta, isto é, torna público, que serão sempre procedimentos e nomes de tipos de dados, mas nunca representações internas.

```
IMPORT PONTO2D;
VAR ponto : Ponto2D;        (* usa o tipo definido no módulo *)
    cx, cy : INTEGER;

BEGIN
   criaPonto2D(10, 10, ponto ) ;       (*cria um ponto já inicializado *)
   cx := getX(ponto); writeln("coordenada em X: ", cx);
   cy := getY(ponto); writeln("coordenada em Y: ", cy);
   incX(ponto); incY(ponto);
   writeln("Nova coordenada em X: ", getX(ponto));
   writeln("Nova coordenada em Y: ", getY(ponto));
        ..........................................................
```

Todo o código está escrito usando apenas os procedimentos definidos no módulo e sem nunca utilizar a representação interna do tipo de dados que, apesar de tudo, o programador sabe ser um RECORD com dois campos INTEGER de nomes x e y. No entanto, o programador respeitou as regras do encapsulamento tendo apenas usado a interface do módulo.

Consideremos, em seguida, um segundo programa que realiza exactamente as mesmas operações básicas que este primeiro, mas que, no entanto, as vai realizar usando o conhecimento que o seu programador tem sobre a representação interna do tipo de dados Ponto2D e que usa explicitamente. Ou seja, conhecendo qual é a representação de Ponto2D, usa-a directamente no seu código para fazer acesso aos valores que da mesma pretende manipular.

```
IMPORT PONTO2D;
VAR ponto : Ponto2D;          (* usa o tipo definido no módulo *)
    cx, cy : INTEGER;

BEGIN
   criaPonto2D(10, 10, ponto ) ;       (*cria um ponto já inicializado *)
   cx := ponto.x; writeln("coordenada em X: ", cx);
   cy := ponto.y; writeln("coordenada em Y: ", cy);
   ponto.x := ponto.x +1; ponto.y := ponto.y +1;
   writeln("Nova coordenada em X: ", ponto.x);
   writeln("Nova coordenada em Y: ", ponto.y);
          ..................................................
```

Naturalmente que o programa não vai gerar erros de compilação e vai executar de forma correcta, produzindo até os mesmos resultados que o primeiro. Porém, este segundo programa, contrariamente ao primeiro, não é independente do contexto e a qualquer momento poderá tornar-se num programa errado e que não executa.

De facto, porque usa directamente a representação interna do tipo de dados, que deveria ser protegida, mas que de facto não o é na maioria das linguagens, caso tal representação do tipo de dados seja modificada o programa passará a conter erros.

Imagine-se que o programador do módulo PONTO2D pretende tornar o módulo mais eficiente e altera a representação interna do tipo Ponto2D para um "array" com duas posições, cf. ARRAY [1..2] OF INTEGER, e reprograma o código dos procedimentos em conformidade com esta nova representação. Que vai acontecer aos dois programas anteriores ?

O primeiro programa, porque apenas usou os mecanismos de acesso ao módulo que fazem parte da interface deste, não sofreria qualquer tipo de problema. Porém, o segundo programa deixaria de estar correcto porque usa a representação antiga de forma explícita e esta foi entretanto alterada.

Finalmente, uma outra regra de programação muito importante e que deve ser respeitada sempre que pretendermos escrever código independente do contexto e,

portanto, reutilizável, é nunca introduzir código de leitura e escrita (*input/output*) junto do código que implementa a camada computacional.

Há muitos anos que na área das interfaces com o utilizador foi definido um princípio fundamental para o correcto desenvolvimento e articulação entre as duas camadas, designado por *princípio da separação*, que advoga exactamente a completa separação das instruções que implementam a interface com o utilizador das que implementam a funcionalidade das aplicações, ou seja, a separação da camada interactiva da camada computacional.

A justificação é, cada vez mais, muito fácil de compreender. Antes de mais, para a mesma camada computacional poderá ser necessário ter mais do que um tipo de interface com o utilizador, dependendo dos ambientes de execução. Por exemplo, sem componentes gráficos, com componentes gráficos, como janelas e botões, de um dado sistema de janelas ou de outro, etc. Se introduzirmos código de *input* ou de *output* na camada computacional, é evidente que a funcionalidade desta ficará dependente do tipo de interface com o utilizador que tal código representa. Deixa de imediato de ser uma camada computacional independente do contexto e, assim, não reutilizável ou, no mínimo, não facilmente reutilizável.

Em conclusão, tendo sido durante muitos anos uma das grandes preocupações da Engenharia de Software a criação de unidades de programação que oferecessem a quem desenvolve aplicações não apenas um grau de abstracção que permitisse que a complexidade dos projectos fosse diminuída, mas também que permitissem reduzir custos de projecto caso pudessem ser reutilizáveis, e tendo-se pensado inicialmente que a solução estava nas *abstracções de controlo ou procedimentais*, a prática veio a demonstrar que a evolução de tais mecanismos se deveria centrar nos dados e não nas instruções, o que conduziu à definição actualmente em vigor de que tais unidades de abstracção, independentes do contexto e reutilizáveis, devem ser de facto *mecanismos de abstracção de dados*, que assumem a estrutura de uma cápsula cujo acesso exterior deverá apenas ser realizado através do que tal cápsula disponibiliza para o exterior através da definição da sua *interface* ou *API*.

Através de pequenos exemplos foram apresentadas regras quer para a construção quer para a utilização de tais módulos, regras que são essenciais para que tais mecanismos sejam não só bem programados, como também bem utilizados. A noção de *objecto*, crucial à PPO, que se apresenta na secção seguinte, assenta fundamentalmente em tais conceitos e regras, que são ainda complementadas em

PPO pela existência de mecanismos adicionais de abstracção, de generalização e de extensibilidade.

1.2 O QUE É UM OBJECTO ?

A noção de *objecto* é uma das noções cruciais ao paradigma da PPO, dado que tal conceito pretende em si concentrar todas as virtudes de um modelo de concepção e desenvolvimento de *software* baseado nas propriedades anteriormente estudadas e vistas como fundamentais, e que são: *a independência de contexto* (que permite reutilização), *a abstracção de dados* (que garante abstracção), o *encapsulamento* (que garante abstracção e protecção) e a *modularidade* (que garante composição de partes simples no desenvolvimento e na concepção).

Um *objecto* é, no contexto da PPO, o módulo computacional básico e único, e, por definição, corresponde à representação abstracta de uma entidade autónoma sob a forma de :

- Um *identificador único*;
- Um *conjunto de atributos privados* (o **estado** interno do objecto);
- Um *conjunto de operações* que são as únicas que podem aceder de forma directa a tal *estado interno*. Destas operações algumas podem ser definidas como invocáveis a partir do exterior do objecto (*públicas*), constituindo a sua **interface**, enquanto que outras poderão ser declaradas como apenas acessíveis a partir de outras internas ao objecto (*privadas*); Tais operações representam no seu conjunto o **comportamento** total do objecto.

As operações que um objecto é capaz de realizar e que são por si definidas como acessíveis ou invocáveis do seu exterior, constituem aquilo que normalmente se designa por **interface** do objecto, ou até a **API** (de *application programmer's interface*) do objecto, no sentido de que quem pretender utilizar a funcionalidade oferecida pelo mesmo apenas o poderá fazer usando as operações definidas por tal objecto como públicas.

Dadas estas características e propriedades de acesso e visibilidade, um *objecto* é de facto uma entidade cuja estrutura interna *deve ser* desconhecida no exterior e, por isso, não directamente acessível, e que apenas divulga para o exterior um conjunto

de operações que é capaz de executar quando externamente invocadas, operações essas que poderão, ou não, devolver a quem as invocou um resultado. É comum até associar este comportamento dos objecto à noção de prestação de *serviços* às entidades que os solicitem via interface.

Assim, um objecto pode ser visto como sendo uma *caixa preta* ou *cápsula* que disponibiliza alguns *botões* que, quando são accionados, realizam uma computação interna no objecto e devolvem, ou não, um resultado. O conjunto de *serviços* que um objecto é capaz de realizar coincide com a sua *interface* ou *API*.

Passaremos a designar os identificadores que guardam os valores dos atributos de um dado objecto por **variáveis de instância**, e as operações que representam o seu comportamento, ou seja, as computações que é capaz de realizar internamente, por **métodos de instância**.

1.3 ENCAPSULAMENTO: PROPRIEDADE FUNDAMENTAL

A figura seguinte procura reforçar visualmente esta perspectiva de que um *objecto* é uma *cápsula* (uma *capsulação de dados*, ou ainda, usando um neologismo informático de origem anglo-saxónica, um *encapsulamento de dados*).

Em termos genéricos, ou seja, independentemente de qualquer paradigma e tendo apenas em atenção propriedades desejáveis do *software*, demonstrou-se na secção anterior que o *encapsulamento dos dados* é uma propriedade fundamental para que se possa atingir a tão desejada *independência de contexto*, que vai, por sua vez garantir propriedades importantes tais como facilidade de *reutilização*, facilidade de *detecção de erros* e *modularidade*.

Tal como a figura permite analisar, um objecto em PPO possui uma *estrutura de dados interna e privada*, que corresponde à sua representação definida. Claro que tal representação atributiva e de carácter informático é apenas uma das muitas que poderiam ter sido encontradas. Por exemplo, a representação de um Ponto2D, ou seja, um ponto do plano que necessita de ser caracterizado por dois valores inteiros ou reais que representam as suas coordenadas em x e y, tanto poderia ser conseguida à custa de 2 variáveis simples do tipo inteiro ou real, como por um

array com duas posições capazes de armazenar tais valores, como por um grande número de muitas outras representações equivalentes.

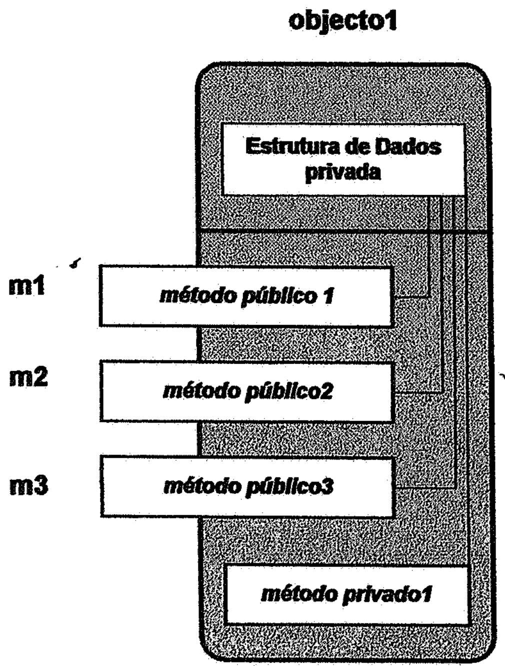
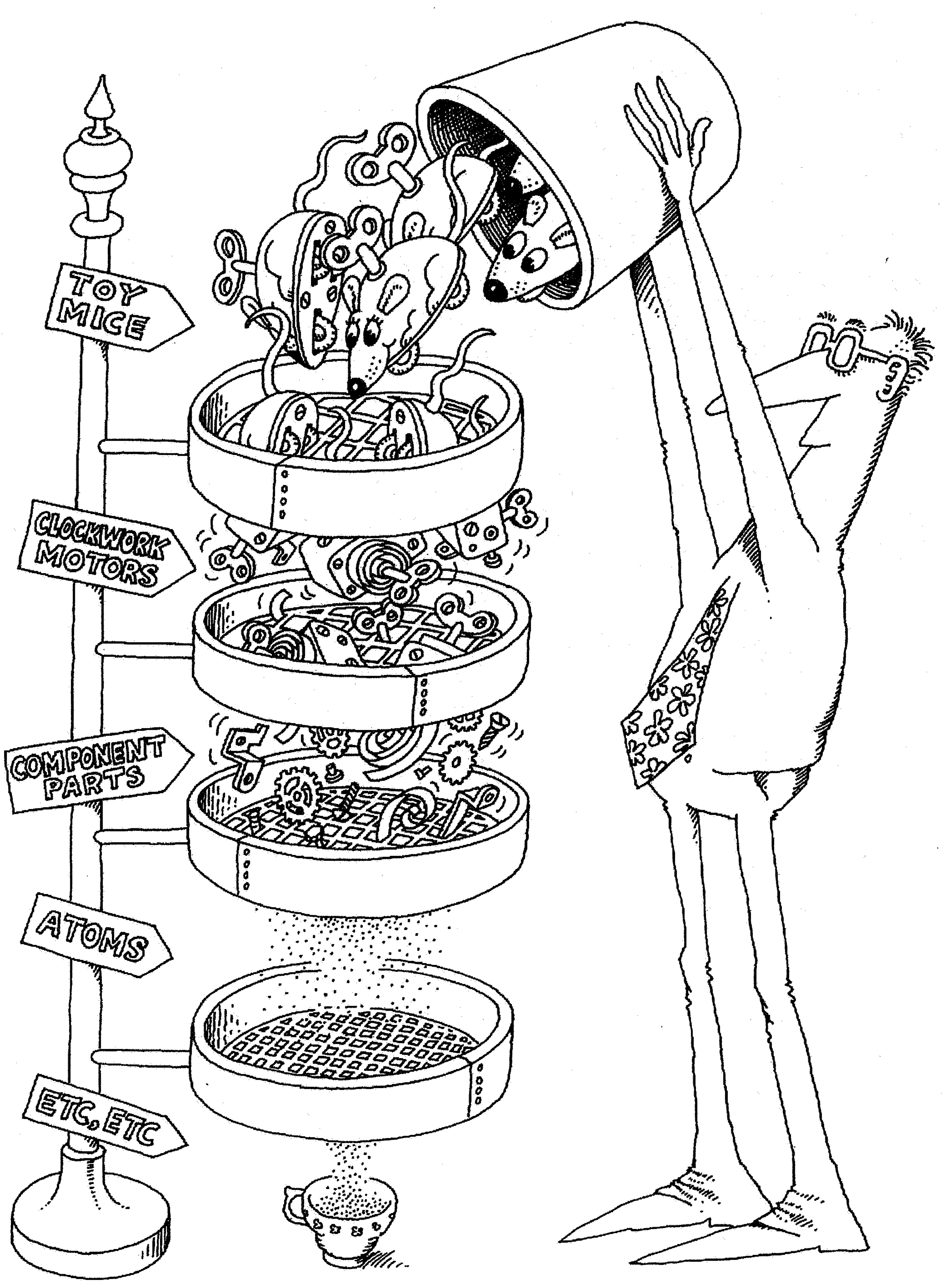


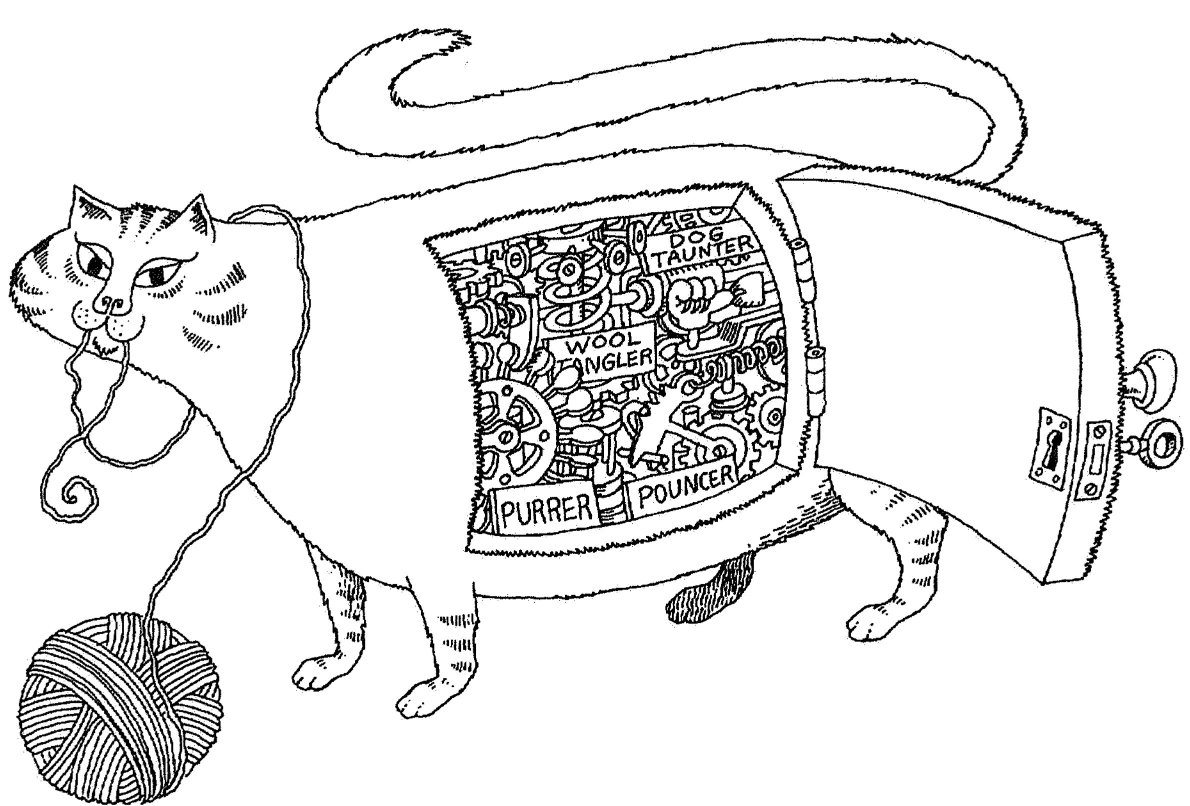
Fig. 1.7 – Objectos como cápsulas

Ideal e teoricamente, para que se pudesse garantir numa qualquer linguagem de programação por objectos o "total encapsulamento" e, portanto, a garantia de uma total independência do contexto e, assim, a reutilização, tal *estrutura interna* deveria estar "escondida", ou seja, ainda que pudesse ser conhecida do exterior, não poderia ser directamente acedida e manipulada, tal conduzindo a um erro que deveria ser detectado pelo compilador da linguagem.

No exemplo dado, objectos do tipo Ponto2D, do exterior destes não deveria ser possível saber se um ponto é representado por duas variáveis simples, se por um *array* com duas posições, se por um registo com dois campos.



Abstractions form a hierarchy.



Encapsulation hides the details of the implementation of an object.

Não sendo possível saber qual a representação definida para tal estrutura interna do objecto, significa que esta é, para o seu exterior, absolutamente "secreta", e assim ninguém do exterior do objecto poderá à mesma aceder directamente, ou seja, usar de forma explícita e directa a definição interna da mesma, dado ser esta desconhecida. Deste modo, quem do exterior pretender usufruir dos "serviços" do objecto, apenas o pode fazer invocando as operações particulares disponibilizadas através da *interface* para a consulta de tais valores.

A Figura 1.8, que se apresenta a seguir, representa a estrutura e o comportamento de um objecto *triângulo1*, cuja *estrutura* ou *estado* é formado por 3 variáveis de instância definidas como sendo do tipo Ponto, cada uma com o respectivo valor, um par de coordenadas, e cujo *comportamento* é definido à custa de 4 métodos, dos quais 3 são públicos e um é privado.

Os *métodos privados*, dado apenas poderem ser invocados no código de outros métodos do objecto e não a partir do exterior do objecto, funcionam portanto como métodos auxiliares na definição do comportamento deste.

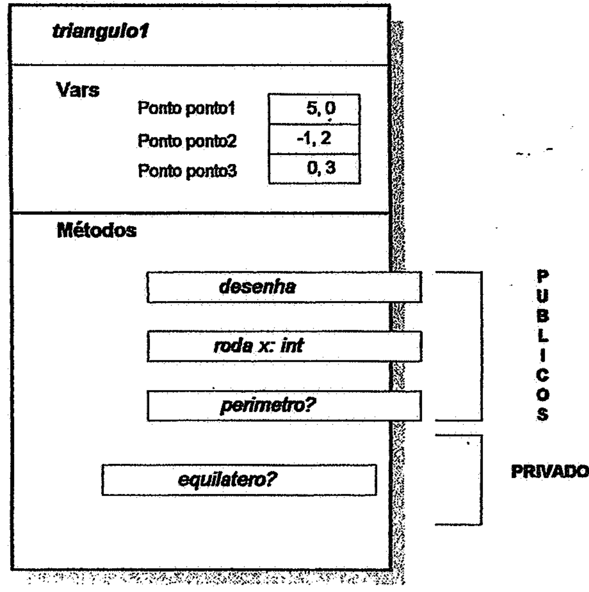


Fig. 1.8 – Um Objecto

Um objecto apresenta-se deste modo como uma *unidade computacional* fechada e autónoma, ou seja, um *módulo* capaz de realizar operações sobre o seu próprio estado interno e devolver respostas para o exterior sempre que os seus métodos definidos como públicos sejam solicitados, isto é, accionados. Ou seja, um objecto é capaz de prestar *serviços* através da activação dos métodos que foram tornados públicos, *serviços* que se traduzem no envio de *respostas* (ou seja *resultados*) às activações realizadas a partir do seu exterior.

Uma questão óbvia a que cumpre de imediato responder é a que diz respeito a uma clara definição do que se deve entender pelo *exterior* de um objecto num programa desenvolvido usando o paradigma da PPO. Como veremos mais detalhadamente adiante, numa linguagem de PPO todas as *unidades computacionais* são *objectos*.

Como veremos na secção seguinte, objectos vão interactuar entre si através de um mecanismo de envio de *mensagens* de uns para outros, mensagens que vão activar os respectivos comportamentos internos programados nos seus métodos, métodos esses que serão responsáveis quer pela alteração do estado interno dos objectos, quer pela determinação de resultados que vão ser comunicados ao objecto emissor de tal mensagem. Desta interacção entre objectos através do envio de mensagens entre si, resulta a computação e as necessárias transições de estado dos objectos, estados individuais que representam, no seu somatório, o estado do programa, que é, portanto, neste paradigma, um estado claramente distribuído.

1.4 MENSAGENS

Assim sendo, torna-se desde já importante analisar qual o mecanismo que em linguagens de PPO permite que uns objectos possam invocar métodos de outros, desta forma solicitando resultados do comportamento interno de outros objectos.

De facto, por exemplo em JAVA, os métodos de um objecto não são invocados de forma directa, isto é, não é uma usual e directa invocação de um procedimento. Em PPO, a interacção entre diferentes objectos faz-se através de um mecanismo de *mensagens*. Quando um objecto pretende invocar um método de um outro objecto a que tem acesso, tem que a este enviar a *mensagem* adequada para que tal método seja executado.

Assim, em PPO, em cada computação, ou seja, em cada transição de estado do programa, existe um objecto que é *emissor* de uma *mensagem* e um outro objecto que é o *receptor* da mesma. A *computação* resulta do facto de que um objecto que recebe uma dada mensagem vai executar, caso tal seja possível, o método que a tal mensagem é por si associado, segundo regras bem definidas, sendo resultados possíveis de tal execução quer a sua alteração de estado interno, quer a devolução de um resultado ao objecto que lhe enviou tal mensagem.

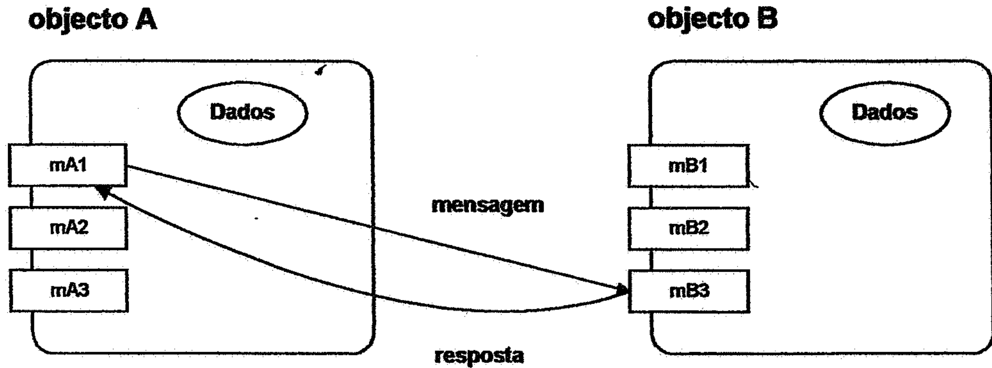


Fig. 1.9 – Interacção entre objectos usando mensagens

Tal como a figura procura representar, o envio de uma mensagem de um objecto a outro é realizado durante a execução de um determinado método do emissor, dado que este necessita de um *serviço* particular do receptor para realizar a sua própria computação, que será certamente para ele próprio poder igualmente prestar um serviço solicitado por um outro qualquer objecto.

Da Figura 1.9 poderíamos inferir que, tendo o objecto A recebido uma mensagem que activou o método *mA1*, durante a execução deste método foi necessário enviar a mensagem *mB3* ao objecto B, cujo resultado é fundamental para a continuação da execução do método *mA1* e cujo resultado é fundamental para a continuação da execução do método que havia enviado a mensagem *mA1* ao objecto B.

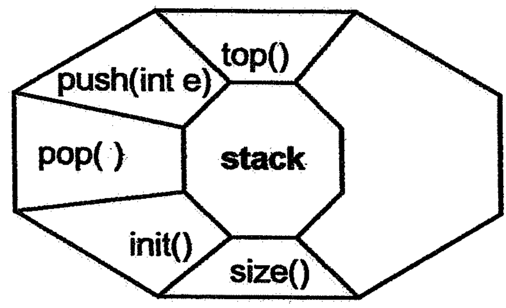


Fig. 1.10. – Um objecto *stack* e a sua *interface*

Na figura anterior representa-se um objecto que implementa uma *stack de inteiros* e que torna públicos os usuais métodos de inicialização de uma *stack*, introdução de um inteiro, remoção do elemento do topo, consulta do elemento no topo e de determinação da actual dimensão da *stack*.

As *mensagens* são, como vimos, um mecanismo de acesso indirecto ao código e estado de um dado objecto. Assim, para cada mensagem enviada a um objecto, e em função do *identificador* e *parâmetros* da mensagem, é activado, caso exista no objecto receptor, o método de igual identificador ao da mensagem, e compatível em termos do número e tipo dos parâmetros de tal mensagem. Para além desta reacção de um objecto à recepção de uma mensagem, se da execução do método resultar um dado valor resultado, este é de imediato utilizável pelo objecto emissor da mensagem.

Ainda que de linguagem para linguagem de PPO possam existir ligeiras variações de sintaxe, a sintaxe geral para o envio de uma mensagem a um objecto assume sempre uma das seguintes formas simples e genéricas:

- *Envio de uma mensagem sem argumentos a um objecto, sem que haja retorno de resultado pelo método correspondente.*

receptor.mensagem( );

* *Envio de uma mensagem com argumentos a um objecto, sem que haja retorno de resultado pelo método correspondente.*

receptor.mensagem(arg1, arg2, ..., argn);

* *Envio de uma mensagem sem argumentos a um objecto, havendo retorno de resultado pelo método correspondente.*

resultado = receptor.mensagem();

* *Envio de uma mensagem com argumentos a um objecto, havendo retorno de resultado pelo método correspondente.*

resultado = receptor.mensagem(arg1, arg2, ..., argn);

Note-se que nas expressões anteriores **resultado** e **receptor** representam de forma genérica identificadores de variáveis que em C++ e em JAVA, que são linguagens de PPO com "type checking", ao contrário, por exemplo, de Smalltalk, têm tipos de dados a si associados, ou seja, apenas podem referenciar valores de tais tipos. Quanto à expressão **mensagem(arg1, arg2, ..., argn)**, ela representa uma forma de identificação de um método a executar no receptor, método esse que, cf. se disse atrás, deve possuir o mesmo nome, ter igual número de parâmetros, sendo os seus parâmetros compatíveis em tipo com os tipos dos argumentos da mensagem, sejam estes valores explícitos (cf. *10*, *"abc"*) ou valores contidos em variáveis (cf. *s*, *x*).

Procurando concretizar agora um pouco mais, através de exemplos simples, todo o poder de um mecanismo tal como o mecanismo de *mensagens*, único em PPO, e a sua efectiva diferenciação da noção de *método*, consideremos um objecto *stack* tal como apresentado anteriormente na Figura 1.10.

Conforme publicitado na sua *interface*, um objecto *stack de inteiros* é capaz de responder às mensagens seguintes, as únicas compatíveis com os métodos tornados públicos:

* push(int e);
* init();

- pop();
- top();
- size();

Admitindo que um qualquer objecto emissor solicitou a um dos vários possíveis objectos que implementam uma *stack* a determinação do seu tamanho, seja no exemplo tal objecto associado à variável *stack1*, tal funcionalidade descrita na Fig. 1.11, seria implementada pela expressão,

tam = stack1.size();

Caso um emissor apenas pretendesse determinar qual o topo actual de tal stack, então a mensagem a enviar ao objecto particular ***stack1***, deveria ser,

elem = stack1.top();

sendo o resultado do envio de tal mensagem, ou seja, o resultado da execução do método top() guardado na variável designada por *elem*.

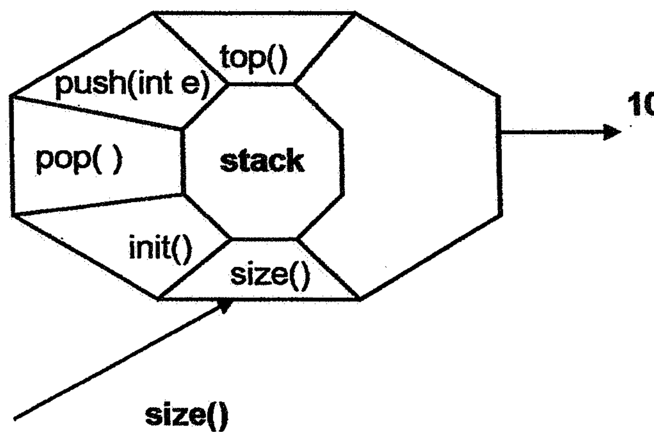


Fig. 1.11 – Trilogia: Mensagem, execução de método e resultado

Admitindo que um dado objecto emissor possa ter acesso ao objecto *stack1*, caso o primeiro pretendesse durante a execução de um qualquer método seu modificar o estado interno de *stack1*, por exemplo, inserindo em *stack1* mais um inteiro, então

MECANISMO DE MENSAGENS ENTRE OBJECTOS COMO MECANISMO DE ABSTRACÇÃO DO TIPO CLIENTE-SERVIDOR

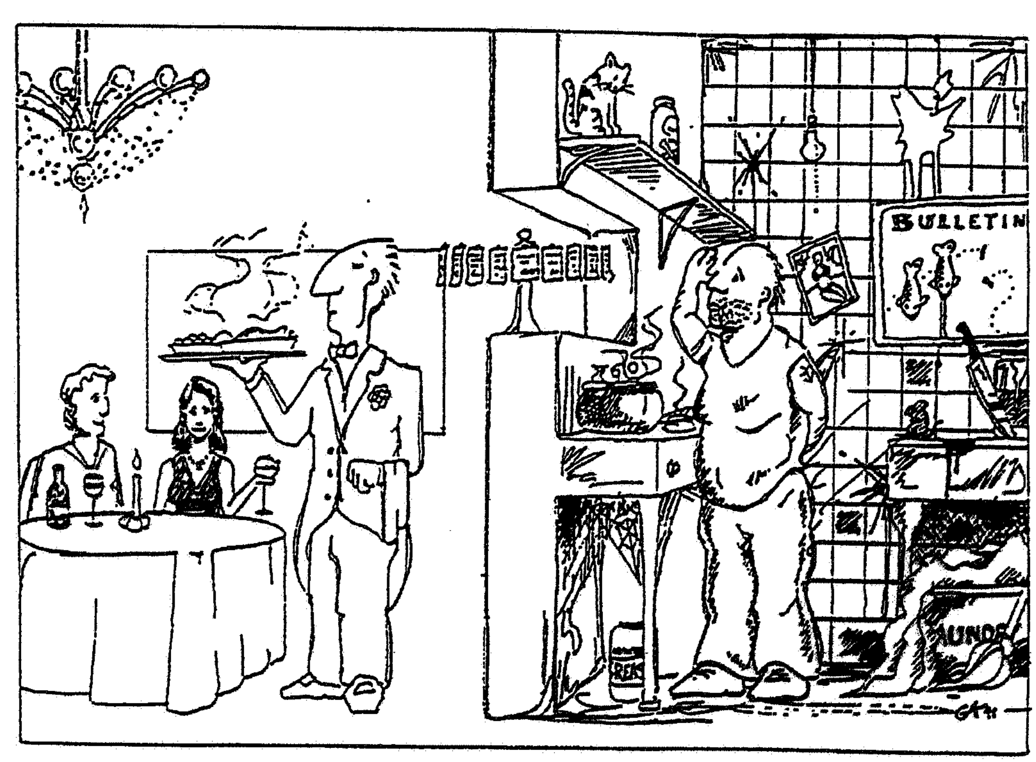


Figure 2-12. Once a message has been passed to an object, objects outside can't know and don't care how processing takes place.

no código de tal método deveria aparecer a expressão correspondente à efectiva execução de tal objectivo, ou seja:

```
stack1.push(12);
```

O facto de o mecanismo de **mensagens** ser independente dos próprios métodos dos objectos, confere a este mecanismo um grau de abstracção e generalização de grande interesse, dado que a mesma mensagem poderá ser reconhecida por vários objectos distintos que executarão os seus próprios métodos adequados.

Assim, por exemplo, se tivermos objectos distintos que são triângulos, rectângulos ou círculos, faz todo sentido que todos eles respondam à mensagem *desenha*:

```
triangulo1.desenha();
rectang1.desenha();
circulo2. desenha();
```

cada um deles activando o método específico para que se obtenha tal resultado. Por outro lado, ao invés do modelo imperativo, se mais tarde criarmos objectos que são, por exemplo, trapézios, então estes objectos também poderão responder à mensagem *desenha()*, desde que implementem o respectivo método *desenha()*.

As mensagens são igualmente um mecanismo de suporte à abstracção, dado que do exterior de um objecto apenas a sua *interface* deverá ser visível, representando esta o conjunto de mensagens que o objecto reconhece. Naturalmente que se enviarmos a um objecto uma mensagem que não faz parte da sua *interface*, será gerada uma situação de erro que em linguagens como C++ e JAVA será detectada em tempo de compilação enquanto que noutras, como por exemplo Smalltalk, seria apenas detectado em tempo de execução.

O facto das mensagens serem independentes dos métodos, para além da necessária coincidência nos nomes e argumentos, é também positivo para que os vocabulários das linguagens, ainda por cima de linguagens extensíveis pelo utilizador, sejam de certa forma reduzidos. Por exemplo, se numa dada linguagem de PPO for usual que a mensagem enviada a um objecto para se determinar a sua dimensão seja, por exemplo, *size()*, torna-se natural que o programador de novos objectos que possuam características dimensionais siga a regra, e programe um método *size()* que responde a tal mensagem. Desta forma, passa a existir não só uma redução do

vocabulário de mensagens e métodos, como também uma natural normalização o que tem grandes vantagens, principalmente tendo em consideração que estas são em gaeral linguagens com bibliotecas extensas e que, existindo um claro objectivo de reutilização, tal significa um esforço inicial de conhecimento do que já existe em geral muito grande.

1.5 OBJECTOS EM PPO: INSTÂNCIAS VS. CLASSES

Introduzimos anteriormente a noção de *objecto* em PPO como sendo uma entidade encapsulada, protegida e unicamente acessível através do que torna público pela sua interface e cujo comportamento definido é acessível do exterior através de um mecanismo de envio de mensagens.

Esta definição genérica está correcta. Porém, deveremos agora analisar a questão de como poderá ser possível, em tal modelo, garantir que todos os objectos de um dado tipo, por exemplo *Triângulo*, que sejam criados num dado programa PPO, tenham exactamente a mesma *estrutura interna* e também igual *comportamento*, isto é, que todos os *triângulos* criados possuem a mesma estrutura e respondem às mesmas mensagens.

Racionalmente, as únicas formas de garantir que todos os *objectos triângulo* são de facto iguais em *estrutura* (possuem as mesmas variáveis de instância necessárias à sua definição) e em *comportamento* (possuem o mesmo conjunto de métodos que implementam as eventuais respostas às mensagens recebidas), são as seguintes:

a) *Qualquer novo objecto de um dado tipo que se pretenda criar é construído a partir de um outro objecto do mesmo tipo usando um mecanismo de "copy & paste", dando-se, em seguida, os valores desejados às variáveis de instância do novo objecto;*

b) *Objectos de um dado tipo possuem a sua representação estrutural, ou seja, as suas variáveis de instância, e comportamental, ou seja, os seus métodos, guardados num objecto especial que representa a definição de todos os objectos desse tipo, definição padrão que é reproduzida sempre que um novo objecto de tal tipo tem que ser criado.*

Ainda que tenham existido linguagens de PPO que tenham implementado a solução a), de facto, as principais linguagens de PPO, como Smalltalk, C++ e JAVA, por razões que apresentaremos a seguir, adoptaram a solução b).

Nestas linguagens, a forma encontrada para se garantir que todos os objectos de um dado tipo têm igual estrutura e comportamento, foi criar uma espécie particular de objectos que guardam a definição de tal estrutura e de tal comportamento.

Estes objectos especiais designam-se em PPO por CLASSES. Assim, CLASSES são, numa primeira definição, objectos particulares que servem para:

- *Conter a descrição da estrutura e comportamento de objectos similares;*
- *Criar objectos particulares possuindo tal estrutura e comportamento.*

Assim, passaremos a designar por CLASSES todos os objectos que irão guardar a estrutura e o comportamento comuns a todos os objectos a partir de si criados, e que designaremos por INSTÂNCIAS (tradução discutível mas estabelecida da respectiva palavra inglesa "instance"), sendo certo que todo o objecto criado a partir de uma dada classe exibirá todas as propriedades estruturais e funcionalidade definidas pela sua classe.

Isto é, a única evolução relativamente a tudo o que até ao momento havia sido dito relativamente ao paradigma da PPO, é que ficamos agora a saber que os objectos a que até agora nos havíamos referido são de facto INSTÂNCIAS de uma CLASSE, sendo tal CLASSE representada por um objecto especial que garante que todas as instâncias a partir de si criadas são coerentes, ou seja, possuem a mesma estrutura e o mesmo comportamento.

Portanto, se num dado programa necessitarmos de objectos do tipo Ponto2D, que possuem duas variáveis de instância para representar as suas coordenadas inteiras e alguns métodos para a sua manipulação, então, deveremos construir a CLASSE de nome Ponto2D e, a partir desta, as INSTÂNCIAS de Ponto2D que necessitarmos.

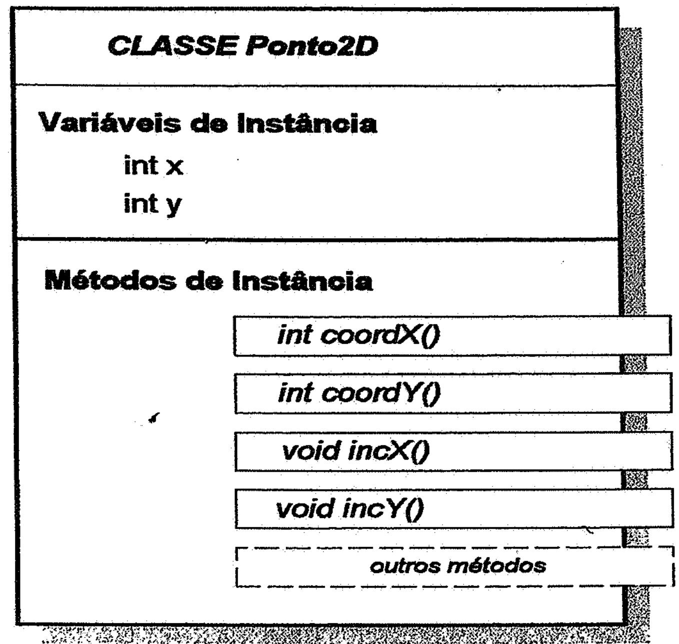


Fig. 1.12. – A classe Ponto2D

Tal como definida, numa notação já muito próxima de JAVA, esta classe Ponto2D especifica que qualquer objecto *instância* de Ponto2D que a partir da mesma possa vir a ser criado, deve conter duas variáveis de instância de nome **x** e **y**, variáveis que apenas poderão conter valores do tipo inteiro e, ainda, que qualquer instância de Ponto2D deverá ser capaz de responder às mensagens *coordX()* e *coordY()*, que terão como resultado, respectivamente, os valores inteiros das coordenadas em x e em y dos pontos receptores das mesmas, bem como serão capazes de incrementar os seus valores internos em x e y ao receberem as mensagens *incX()* e *incY()*, em cujo caso apenas é alterado o estado interno do objecto, não sendo pois devolvido qualquer resultado, conforme se especifica usando a palavra reservada *void*.

Definida a classe Ponto2D, poderemos a partir de agora criar objectos que são de facto as INSTÂNCIAS desta CLASSE Ponto2D. Em JAVA tal é realizado usando expressões da forma

```
Ponto2D pt1 = new Ponto2D();
Ponto2D pt2 = new Ponto2D();
```

que serão mais tarde completamente analisadas, mas que, conforme facilmente se pode compreender, criam duas instâncias de Ponto2D referenciadas por *pt1* e *pt2*, variáveis que foram declaradas como tendo tal tipo.

Possuindo um objecto do tipo Ponto2D referenciado pela variável *pt1*, poderemos a partir de então, e tal como vimos atrás, enviar a tal objecto, que é instância da classe Ponto2D, as mensagens que na sua classe foram definidas como sendo as mensagens a que este é capaz de responder, por exemplo as seguintes:

```
cx = pt1.coordX();
cy = pt1.coordY();
pt1.incX();
pt1.incY();
```

Torna-se também de imediato claro que em PPO qualquer INSTÂNCIA possui um só tipo, isto é, é instância de uma e uma só CLASSE.

Por outro lado, numa linguagem de PPO pura, todas as suas entidades definidoras deveriam ser *objectos*. Smalltalk é a linguagem de PPO mais próxima do conceito de linguagem "pura" de PPO, já que quer classes quer instâncias quer mensagens são de facto *objectos*. C++ e JAVA não são linguagens de PPO "puras" já que nas suas definições surgem entidades que não são objectos. Por exemplo, quer C++ quer JAVA possuem tipos de dados simples, como o tipo inteiro ou o tipo real, que representam conjuntos de valores, valores estes que não são objectos. Tal não representa qualquer tipo de desvantagem da linguagem, antes pelo contrário, tanto mais que, por exemplo em JAVA, valores de tipos simples podem ser convertidos em instâncias de classes equivalentes, sendo por exemplo possível converter um valor inteiro numa instância da classe *Integer*, que é um objecto. Enquanto que um valor inteiro não tem comportamento, apenas servindo de operando a certas operações, a instância de *Integer* é um objecto, logo possui estado e responde a mensagens.

Na figura seguinte apresenta-se uma definição de uma classe Triângulo, visando apenas mostrar que as variáveis de instância definidas numa classe podem não só ser de tipo simples, mas também definidas como sendo instâncias de uma dada classe já existente.

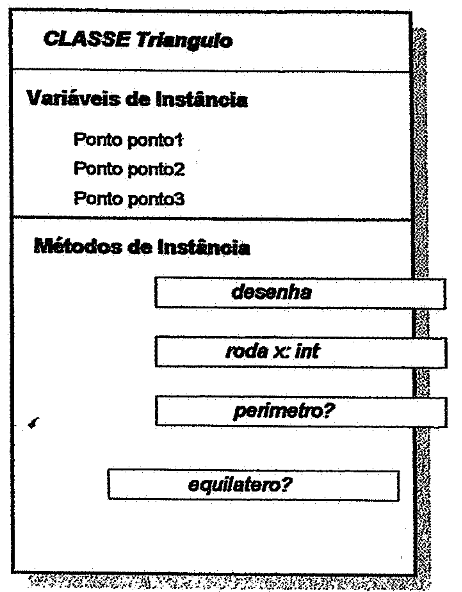


Fig. 1.13 – Classe Triângulo

Finalmente, e no sentido de concluir a apresentação do paradigma da PPO, para que uma dada linguagem de programação possa ser considerada uma linguagem de PPO, para além dos conceitos de *objecto* e *classe* neste capítulo introduzidos, será também necessário que a linguagem implemente o mecanismo de *herança*, que será estudado posteriormente. Existem várias linguagens que possibilitam programar com objectos e classes mas não possuem este terceiro mecanismo tão característico e importante em PPO.

1.6 Síntese do Capítulo

Apresentou-se neste capítulo a génese do paradigma da Programação Orientada aos Objectos, com especial incidência, dada a sua importância para a compreensão do paradigma, na evolução dos conceitos na Engenharia de Software que vieram a conduzir à noção de *abstracção de dados*, da qual deriva a definição de *objecto*, a entidade fundamental em PPO.

Foram introduzidos alguns princípios de programação que visam garantir que os *objectos* que desenvolvemos, bem como quem deles faz uso, são unidades que, por

serem independentes do contexto, garantem facilidade de reutilização. Duas regras foram consideradas como fundamentais:

a) Um objecto não deve manipular directamente os dados internos de outro;
b) Um objecto genérico não deve ter instruções de input/output no seu código.

Assim, a visão de criação e utilização de um objecto é a de uma ***cápsula*** contendo dados privados e que disponibiliza, através da sua ***interface***, um conjunto de métodos que são accionáveis do exterior através de um mecanismo de ***mensagens***.

Esta perspectiva favorece uma outra comum interpretação do que se deve entender por objectos, apresentado-os como "caixas pretas" que prestam um conjunto de *serviços* ao exterior ao receberem *mensagens* que lhes sejam apropriadas. Esta visão permite ainda que, por vezes, se refira o objecto que envia a mensagem como *objecto-emissor* ou *objecto-cliente*, e o que a recebe e, eventualmente, responde à mesma como *receptor* ou *servidor*.

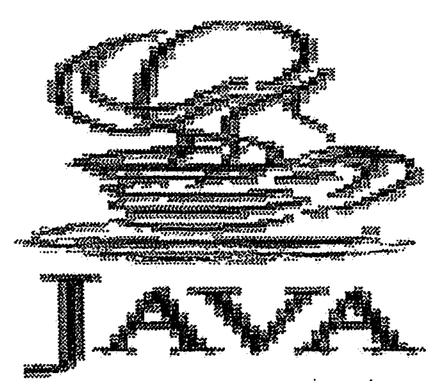
A computação num sistema de objectos decorre da troca de mensagens entre os vários objectos, dessa troca de mensagens resultando alterações nos seus estados internos e, consequentemente, do estado global do programa, que, neste modelo computacinal, é um estado distribuído por todos os objectos que o constituem.

Finalmente, foi realizada a distinção entre dois tipos de objectos, ***instâncias*** e *classes*, tendo as classes sido definidas como objectos particulares que servem de padrão, por forma a garantirem que todas as suas instâncias possuem igual estrutura e comportamento, sendo por isso todas as instâncias criadas a partir de si.

No próximo capítulo introduziremos a tecnologia e as bases da linguagem JAVA que iremos posteriormente utilizar e enriquecer nos capítulos seguintes.

PROGRAMAÇÃO POR OBJECTOS

EM



A TECNOLOGIA JAVA

INTRODUÇÃO

Ainda que o objectivo fundamental deste curso seja a Programação Orientada aos Objectos usando a linguagem JAVA, JAVA é hoje em dia uma palavra reservada que significa muito mais do que mais uma Linguagem de Programação por Objectos, pelo que se torna importante perceber as razões que fazem com que seja actualmente tão relevante conhecer a linguagem, saber usá-la em *programar por objectos*, conhecer a tecnologia subjacente à linguagem, e perceber as razões do tão grande impacto causado por esta tecnologia.

JAVA é, já hoje, não apenas uma *melhor* linguagem de programação por objectos - que do ponto de vista do autor é, de forma indiscutível -, mas, mais do que isso, uma verdadeira nova tecnologia, que foi repescar ideias há muito desenvolvidas mas entretanto esquecidas ou não completamente exploradas, e que, com os devidos apoios, as apresentou como soluções num contexto informático extraordinariamente dependente de tal tipo de soluções.

GÉNESE DA TECNOLOGIA JAVA

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

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

A IBM ainda chegou a anunciar em 1994/95 a linguagem Smalltalk como a sua linguagem de Programação por Objectos, tendo até desenvolvido um ambiente de projecto, o VisualAge-Smalltalk, ainda hoje comercializado para os OS/2 e OS/390 da IBM e para o Windows-NT.

A linguagem Java começou a ser desenvolvida no início dos anos 90 no seio de uma pequena equipa de engenheiros de software da Sun Microsystems liderada por James Gosling. O objectivo era desenvolver uma pequena linguagem para equipamentos electrónicos com "chips"

programáveis, tais como torradeiras, máquinas de lavar, agendas electrónicas de bolso, etc. Os principais requisitos da linguagem a desenvolver eram a sua robustez e segurança (os utilizadores destes dispositivos não admitem erros ou falhas), barata (os programas teriam que ser simples) e independente dos "chips" (dado que os construtores muito facilmente os substituem por outros). O projecto designava-se *Green*.

Numa primeira fase, o modelo do UCSD Pascal, uma das bem sucedidas linguagens para PCs, ainda terá sido considerado, ou seja, a geração de um código intermédio após a compilação - o *p-code* - e criação de interpretadores para arquitecturas específicas. Em geral, o *p-code* era pequeno e interpretadores de *p-code* igualmente de reduzida dimensão. Porém, a maioria dos engenheiros de software do grupo tinham grande formação em Unix e, portanto, grandes conhecimentos de C e C++, pelo que pretenderam usar os conhecimentos já adquiridos e a sua segurança nestas linguagens. Porém, as análises feitas pela equipa às linguagens C e C++ revelaram não serem estas as linguagens adequadas, por um lado dada a sua complexidade e dificuldade associada em escrever código garantidamente seguro, e por outro, por necessitarem de um compilador específico para cada tipo de "chip". A equipa desenvolve então a linguagem Java, inicialmente designada *Oak*, simples, segura e independente de arquitecturas. O primeiro produto lançado foi um controlo remoto extremamente inteligente , o *7, que, apesar de tudo, ninguém pretendeu produzir.

Em 1993 a equipa é confrontada com o aparecimento da World Wide Web via Internet, e do extraordinário impacto produzido. Java perfila-se então, dadas as suas características, como uma linguagem fundamental para a Internet. A sua neutralidade relativamente a arquitecturas, torna-a ideal para o desenvolvimento de aplicações que com facilidade possam ser executadas em qualquer uma das diferentes máquinas ligadas à Internet.

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

A ideia forte associada à linguagem Java é *"escrever código uma vez e executá-lo em qualquer parte"*. Para que tal seja possível, o código fonte das aplicações escritas em Java tem que ser isolado pelo ambiente Java dos sistemas operativos das máquinas e dos dispositivos hardware. Logo, tal código não pode ser compilado directamente para código nativo das máquinas, mas antes para uma representação especial, neutra, designada *byte-code*. De seguida, este código intermédio será interpretado sobre o ambiente particular de cada máquina, para tal sendo desenvolvido um interpretador particular de byte-code para cada plataforma onde se pretendam executar programas Java. Este *runtime-engine*, ou motor de execução, designa-se por Java Virtual Machine (JVM). A JVM recebe *byte-code* e transforma-o em instruções executáveis da máquina particular onde Java é instalado. Assim, como é fácil compreender, existem muitas JVM, por exemplo, para Windows 3.1, para Windows95, para Windows-NT, para Linux, para UNIX, para OS/2, para Solaris, para MacIntosh, etc.

A figura seguinte ilustra este processo de compatibilização do código fonte Java com as diferentes plataformas.

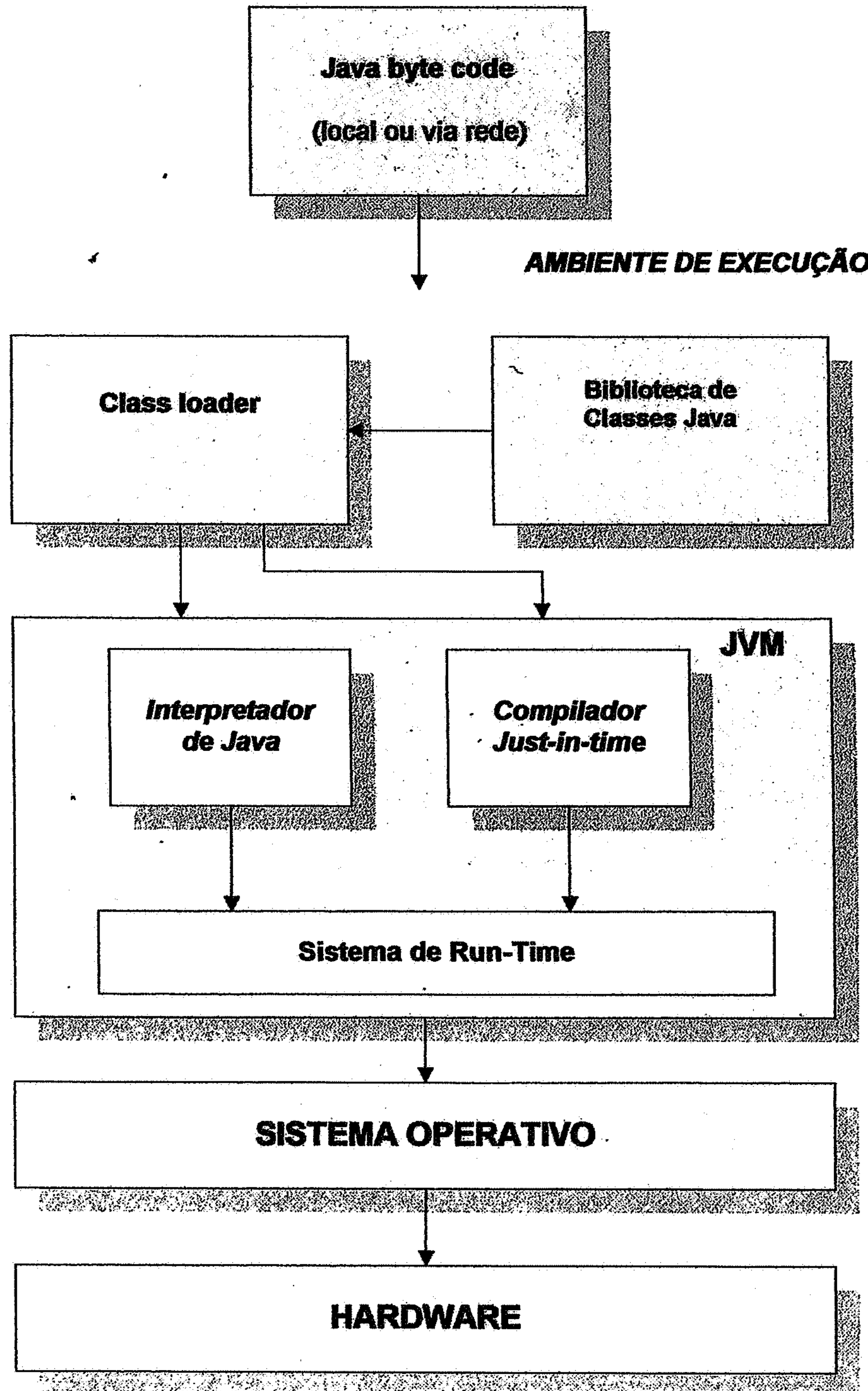


Fig. 1

Java pode ser usada para criar dois tipos de programas: *aplicações* e *"applets"*. As aplicações Java, tal como outras escritas noutra qualquer linguagem de programação, são programas que requerem uma JVM para serem executados, podendo esta JVM existir ao nível do sistema operativo ou ligada à própria aplicação, e que executam por si próprios, ou seja, são o que em geral se designada por aplicações *"standalone"*. *Applets* são porções de código Java não executáveis por si próprias, e que dependem de um *browser* que incorpore e execute a JVM (cf. Netscape ou Explorer) para que possam ser executadas. Como *applets* são escritos para serem "descarregados" via rede, são em geral muito mais pequenos em dimensão que as aplicações.

É importante desde já perceber que o que torna Java uma linguagem muito atractiva para toda a indústria de computadores, é que Java não é apenas uma nova linguagem de programação, e ainda por cima por objectos, e adicionalmente - ao contrário de C++ - quase totalmente pura, mas o facto de que Java, principalmente a partir do lançamento do seu sistema de desenvolvimento, o JDK1.1, se posicionou com um atractivo e apropriado, no contexto actual, *ambiente de programação/desenvolvimento de aplicações*.

Curiosamente, Smalltalk, linguagem desenvolvida nos anos 70, era também não apenas uma linguagem, mas também um ambiente e, mais do que isso, a demonstração da viabilidade e eficácia de um, então, novo paradigma, o da programação por objectos. Porém, Smalltalk acabou por nunca vingar verdadeiramente - apesar de ser uma das mais extraordinárias linguagens de programação alguma vez desenvolvidas.

Quais então as razões do sucesso de Java, que é neste momento a linguagem de programação com maior taxa de crescimento e aderência alguma vez verificada na história das Tecnologias de Informação ? Alguns autores afirmam que, para além das capacidades e potencialidades intrínsecas da linguagem, com vocação - via bibliotecas adequadas - para processamento usando tecnologia cliente-servidor e fácil acesso a comunicações, Java terá tido a "sorte", ou a visão comercial dos seus autores, de ter visto a sua JVM distribuída com os dois Web-browsers de maior sucesso no mundo - Nestscape e Explorer. A penetração de Java no mercado foi, por esta razão, para além dos méritos intrínsecos da linguagem, de nível esmagador.

Os passos seguintes da curta história de Java, são a criação em 1996 da JavaSoft pela Sun, empresa destinada a desenvolver produtos Java para as mais diversas áreas - e que em 1998 apresenta já um catálogo de centenas de produtos nas mais diversas áreas -, o lançamento do ambiente de desenvolvimento JDK1.0 em 1996, a primeira conferência sobre produtos Java - a JavaOne Conference -, em 1996, a adesão da Corel em 1996 - que acaba por falhar , o lançamento do JDK1.1 em Fevereiro de 1997 já com JavaBeans e outras facilidades de desenvolvimento, o processo em tribunal da Sun contra a Microsoft, em Outubro de 1997, por esta empresa ter desenvolvido código Java para o Explorer 4.0 que não passou nos testes de compatibilidade de Java da Sun, requisito acordado na licença de Java assinada pela MicroSoft - a Microsoft vingar-se-ia tecnologicamente em 1998, ao apresentar - cf. testes realizados por uma equipa de peritos da PC Magazine em Abril de 1998- a mais compatível JVM para Windows, e o melhor ambiente de desenvolvimento, o Microsoft J++. Em Janeiro de 1998, a Sun, via Java, conquista outra enorme vitória: a TCI (Tele-Comunications Inc.), o maior fornecedor americano de infraestruturas de TV-Cabo, seleccionou o *PersonalJava* como um dos dois sistemas operativos - o outro é o Windows ! - como sistema operativo de base das suas *caixas-digitais*.

JAVA é, neste momento, enquanto linguagem e ambiente de desenvolvimento, e também enquanto representante do paradigma da programação, desenvolvimento e análise por objectos, indiscutivelmente importante para todas as pessoas ligadas às Tecnologias de Informação.

O futuro da tecnologia JAVA estará certamente, a avaliar pelos investimentos já realizados, assegurado vendo Java como linguagem de PPO e, talvez até, como ambiente de programação e mesmo Sistema Operativo. A expectativa gira, neste momento, apenas à volta da versão de JDK1.2 a lançar em 1998, e de JDK2.0. Alguns desenvolvimentos são já conhecidos.

Visando evitar o que a actual JVM faz que é apresentar aplicações usando *widgets* nativos da máquina onde executa, o que conduz a IU semelhantes apenas em plataformas iguais - destruindo assim a consistência das IU entre plataformas - serão no JDK1.2 incluídos componentes de suporte para Interfaces com o Utilizador (IU) "pluggable", ou seja, programáveis e integráveis na própria aplicação, permitindo a criação de IU independentes das plataformas.

Adicionalmente, será em JDK1.2 suportada a edição de JavaBeans (suporte a *authoring*), aumentando assim a possibilidade de escrita de código reutilizável em novos projectos. Os *Beans* Java são um pouco mais portáveis que os *ActiveX* da Microsoft, tendo a adicional vantagem de encapsularem código Java, enquanto que os ActiveX não suportam qualquer linguagem de programação.

Finalmente, e com respeito à sua eficiência e velocidade, espera-se que JDK1.2 e JDK2.0 passem a usar mais intensivamente a técnica de *"just-in-time compiling"*, melhorando o compilador JIT da JVM, que parece ter ainda uma grande margem de desenvolvimento possível.

Ambientes, Ferramentas, Aplicações JAVA Disponíveis.

Muitos têm sido os ambientes de desenvolvimento, as ferramentas e as aplicações dos mais diversos tipos, desenvolvidos em Java nos últimos anos. A PC-MagaZine de Abril de 1998 fez um louvável e rigoroso teste a todas estas diferentes utilizações de Java, podendo os resultados de tal estudo, o mais actual existente, ser sintetizados nas seguintes conclusões.

Crescimento da Adesão a Java

O quadro seguinte visa apenas demonstrar o crescimento da adesão à linguagem Java entre 1997 e 1999. Os números falam por si, sendo de acrescentar que foram inquiridas 50 empresas das que figuram anualmente nas 1000 maiores do mundo segundo a revista Fortune.

Aplicações Java

Ano	*Nenhuma*	*Experimentais*	*Baixo Impacto*	*Críticas*
1997	48%	36%	12%	4%
1999	8%	24%	12%	56%

PLATAFORMAS - COMPATIBILIDADE

Conforme se pode ver pelo quadro seguinte, os ambientes Windows (em particular NT), são ainda não só os mais compatíveis com Java, mas igualmente os que apresentam melhores performances de execução.

Windows	*Solaris*	*OS/2*	*Macintosh*
70%	53%	33%	25%

AMBIENTES - MÁQUINAS VIRTUAIS JAVA (JVMS)

Requisito proposto: *"Qualquer aplicação escrita em Java corre em qualquer SO que execute a versão correcta do ambiente Java, neste caso JDK1.1."*

Facto: *"Qualquer ambiente Java é uma máquina virtual, JVM - também designada plataforma computacional - instalada num sistema operativo receptor (host), que é capaz de executar aplicações e "applets" Java".*

Casos de Estudo: *6 diferentes vendedores, com 5 diferentes sistemas operativos, num total de 18 configurações diferentes. Linux não foi testado por não existir JDK1.1 para Linux.*

Testes: Com ou sem **JIT** (Compilador Just-in-Time); o teste do **Processador** visa determinar até que ponto as JVM exercitam o processador e a memória (com e sem JIT); o teste **AWT** visa testar quão rápidas são as JVM a produzir texto e gráficos no ecrã; o teste de **Aritmética** complexa visa determinar a eficiência das JVM na subdivisão dos cálculos; o teste de **Threads** visa determinar trabalham com múltiplas *threads*; o teste **Stack** procura determinar quais as melhores implementações de *stacks*; o teste de **Ovais** testa a eficiência das invocações e implementações de rotinas de processamento gráfico; o teste **Texto** visa avaliar a eficiência dos diversos applets na apresentação de uma grande variedade de texto formatado; o teste **Gráficos** procura determinar, com base na utilização de threads, como se comportam os ambientes na apresentação de gráficos em múltiplas "janelas".

Resultados: Quanto maior o valor apresentado no quadro final, melhor a performance.

AMBIENTES JAVA - COMPARAÇÃO

	JIT	Processador	AWT	Aritmética	Threads	Stack	Linhas	Ovais	Texto	Gráficos
MAC OS8										
MS IExplorer 4.0	Sim	200	Não executa	418	Não executa	270	186	188	164	Não executa
MRJ 2.0	Sim	167	Não executa	234	234	143	67	99	127	111
Windows 95										
MS IExplorer 4.0	Sim	509	1698	473	343	972	183	124	1212	189
Netscape Com. 4.04	Sim	437	347	343	341	351	209	132	375	179
JDK 1.1.5	Não	32	179	204	112	105	179	144	315	141
JDK 1.1.5 + Win32 PP	Sim	334	276	545	337	485	337	146	553	154
Visual Café 2.1	Sim	366	279	542	678	1014	202	144	621	166
Windows NT										
MS IExplorer 4.0	Sim	505	1715	482	359	978	400	345	2807	312
Netscape Com. 4.04	Sim	435	383	352	358	339	771	396	383	414
JDK 1.1.5	Não	32	304	205	112	106	519	412	369	402
JDK 1.1.5 + Win32 PP	Sim	338	303	538	354	511	630	418	686	504
Visual Café 2.1	Sim	384	474	555	737	1076	716	433	752	556
OS/2 Warp 4.0										
IBM Java 1.1.4	Sim	232	Não executa	609	340	1347	258	56	298	56
Solaris 2.6										
MS IExplorer 4.0	Não	12	470	72	61	95	65	120	674	12
Netscape Com. 4.04	Não	Não executa	Não executa	Não executa	Não executa	Não executa	Não executa	Não executa	Não executa	Não executa
JDK 1.1.3	Sim	206	503	402	301	334	314	636	1172	265
JDK 1.1.5	Não	33	584	200	108	118	376	645	435	157
JDK 1.1.5 + Threads Pack	Não	32	422	197	112	115	273	616	413	231

Análise de Resultados:

Apesar de tudo o que a imprensa informática tem escrito sobre a MicroSoft, este "benchmark" apenas veio demonstrar que a MicroSoft, relativamente à tecnologia Java, a está a encarar com um rigor e interesse elevados. A MicroSoft JVM revelou-se, pela segunda vez consecutiva (no ano passado havia sido testado o JDK1.0), como a mais compatível e efectiva *máquina virtual* de Java de todas as testadas.

A MicroSoft distribui a sua JVM não só como parte do Internet Explorer 4.0, mas também, em conjunto com algumas bibliotecas de classes, com o MicroSoft Software Development Kit for Java (SDKJ).

Os testes mostram que a MicroSoft JVM é a mais rápida, sendo a versão para Windows-NT mais rápida que a versão para Windows-95. Em ambas as plataformas, o IE 4.0 foi 16% mais rápido que o seu adversário mais directo, o NC 4.04 no teste do Processador. Relativamente a operações sobre AWT, a MicroSoft JVM revelou-se 3,5 vezes mais rápida que a Symantec VM, sendo aqui o Windows-NT muito mais eficiente que o 95.

Quanto à compatibilidade, as plataformas MicroSoft revelaram-se mais compatíveis que as plataformas Sun em cerca de 17% (70% contra 53%), sendo no entanto de referir que a Netscape VM sobre Solaris apresentou uma compatibilidade equivalente ao da MicroSoft VM.

A performance e compatibilidade do IE 4.0 são mesmo irónicos tendo em atenção que a Sun colocou uma acção em tribunal contra a MicroSoft acusando-a de ter alterado o código Java por forma a que este incluisse especificidades próprias do Windows. De facto, a MicroSoft não distribui duas das partes do standard JDK, a saber, o Remote Method Invocation (RMI) e o Java Native Interface (JNI). A MicroSoft não implementa o JNI e disponibiliza o RMI no seu Web-site. Programas Java usando RMI não correrão sobre a versão da JVM incluída no IE. A não implementação de JNI implica que facilidades tais como a *ponte* JDBC-ODBC, que se baseiam em JNI, não são suportadas. Em contrapartida, a MicroSoft, que acusa a Sun da ineficiência da JNI, desenvolveu a sua própria *Raw Native Interface* (RNI), muito mais rápida.

Ao fornecer a todos os programadores de Java em ambiente Windows facilidades adicionais, tais como as MFC (MicroSoft Foundation Classes), mais fáceis de usar para a construção de IU que o standard AWT, bem como a possibilidade via MicroSoft J/Direct de realizar invocações directas a qualquer Win32 DLL (incluindo o sistema operativo), mais fáceis do que via JNI ou RMI, e ainda, fácil interface entre a MicroSoft VM e os controlos ActiveX, desta forma tornando fácil a comunicação com outras aplicações Windows, a MicroSoft está a desenvolver a sua estratégia própria que é: *esqueçam a compatibilidade entre plataformas prometida mas ainda não conseguida via Sun-Java, e pensem em Java como uma forma mais fácil de realizarem desenvolvimento de aplicações distribuídas e interactivas em Windows.*

O futuro, como é usual nestas questões, será, quase de certeza, uma bifurcação sem grandes pontos possíveis de retorno.

AMBIENTES/FERRAMENTAS DE DESENVOLVIMENTO EM JAVA

A Sun levou o projecto Java ainda mais longe, ao passar a considerar Java não apenas uma linguagem, ou um ambiente de desenvolvimento, mas uma verdadeira *arquitectura*, capaz de substutuir o Windows de 32 bits. Neste sentido, a Sun lança em Outubro de 1996 a versão 1.1 do JDK (Java Development Kit), integrando o AWT (Abstract WIndows Toolkit), JDBC para acesso a Bases de Dados via ODBC (Open Data Base Connectivity), Java Beans (biblioteca de componentes reutilizáveis), com JVM (Java Virtual Machine) e JOS (Java Operating System). Esta arquitectura e ainda a arquitectura de acesso a Bases de Dados via JDBC são apresentadas nas duas figuras seguintes.

Muitos, e bons, são já os ambientes de desenvolvimento em Java apresentados por diversas companhias, supostamente ferramentas que incluem todos os componentes necessários ao desenvolvimento de aplicações à escala empresarial, cf. acesso a bases de dados distribuídas, comércio electrónico seguro, reutilização de componentes, GUIs, etc., todos eles procurando implementar uma filosofia de concepção e desenvolvimento de aplicações designada por RAD - *Rapid Application Development*.

A opinião geral relativamente aos ambiente compatíveis com o JDK1.0, era a de que, de todas as propostas existentes, o Visual J++ da Microsoft era o ambiente de desenvolvimento mais bem conseguido. Este ambiente encontra-se agora em reformulação para a versão JDK2.0, pelo que não foi considerado na análise que a seguir se sintetiza.

Os mais significativos e actualmente disponíveis ambientes de desenvolvimento de aplicações Java, são o Visual-Café da Symantec, o Visual-Age Java da IBM, o CodeWarrior Professional da Metrowerks, o Sybase PowerJ Enterprise 2.1 da Sybase, o JBuilder Client/Server Suite da Borland, e o Sun Java WorkShop 2.0 da Sun.

Diversos testes realizados sobre estes ambientes de desenvolvimento realizados por uma equipa de peritos da PC-Magazine de Abril de 1998, produziram os seguintes interessantes resultados.

Dimensão dos Ficheiros das Aplicações (para 1 aplicação exemplo)

Ambiente de Desenvolvimento	*Tamanho em Bytes*
CodeWarrior Professional 2.0	35152
JBuilder Client/Server Suite	41689
Sun Java WorkShop 2.0, native	40403
Sun Java WorkShop 2.0, fast compiler	50145
Sybase Power Enterprise 2.0	**34140**
Sybase Power Enterprise 2.1	34477
Visual-Age for Java 1.0	34501
Visual-Café for Java 2.1, não optimizado	41689
Visual-Café for Java 2.1, optimizado	35476

Tempos de Compilação - Aplicação de Xadrez em Full Debug mode

Ambiente de Desenvolvimento	*Tempo de Compilação*
CodeWarrior Professional 2.0	6,5 s
JBuilder Client/Server Suite	**1,9 s**
Sun Java WorkShop 2.0, native	8,7 s
Sun Java WorkShop 2.0, fast compiler	**2,2 s**
Sybase Power Enterprise 2.0	6,2 s
Sybase Power Enterprise 2.1	8,6 s
Visual-Age for Java 1.0	Não executável
Visual-Café for Java 2.1, optimizado	**2,1 s**

Testes de Compatibilidade vs. Especificação de Java (Testes Grinder)

Ambiente de Desenvolvimento	*Total de Testes OK*
CodeWarrior Professional 2.0	**520**
JBuilder Client/Server Suite	**515**
Sun Java WorkShop 2.0	**517**
Sybase Power Enterprise 2.1	504
Visual-Age for Java 1.0	456
Visual-Café for Java 2.1, optimizado	503

Testes de Compatibilidade vs. Especificação de Java (Testes Expresso)

Ambiente de Desenvolvimento	*Total de Testes OK*
CodeWarrior Professional 2.0	2200
JBuilder Client/Server Suite	**2297**
Sun Java WorkShop 2.0	2200
Sybase Power Enterprise 2.1	**2297**
Visual-Age for Java 1.0	2269
Visual-Café for Java 2.1, optimizado	2201

A tabela seguinte apresenta ainda um conjunto de características adicionais deste ambientes de desenvolvimento.

Resultados: A escolha do editor da revista PC-Magazine recai sobre o **Sybase PowerJ**, por muitas razões, não só as visíveis nos testes comparativos, mas também por um conjunto de outras razões que têm a ver com o estágio actual do seu desenvolvimento, como por exemplo, a inclusão de *Beans* empresariais, etc., reunindo assim as maiores capacidades de extensibilidade, flexibilidade, compatibilidade e potência actualmente existentes no mercado, tudo por um preço que, na versão empresarial, rondará os 2.000 dólares (aprox. 380 contos nos USA).

CARACTERÍSTICAS	*CWP*	*JBCS*	*SJW*	*SPE*	*VAJ*	*SVC*
Ambiente de Desenvolvimento						
Help Sensível ao Contexto	N	S	S	S	N	S
Wizards	N	S	S	S	S	S
Lanc. de third-party via IDE	S	S	S	S	N	S
Inclui compilador JIT	S	S	S	S	N	S
Suporta Compilador da SUN	S	N	S	S	N	S
Pode adoptar JIT/VM	SS	SS	SS	SS	NN	SS
Editor						
Faz "highlight" da sintaxe	S	S	S	S	S	S
Mapa de teclas/macros	SS	NN	SS	NN	NN	SS
Ferramentas de Programação						
Wizards podem juntar eventos de controlo	N	S	S	S	S	S
Browser de Classes	N	S	S	N	S	S
Pode fazer Browsing sobre projectos incompletos	N	S	S	S	S	S
Construção Visual de Classes	N	N	N	S	S	S
Criador Visual de GUIs	N	S	S	S	S	S
Pode juntar código a controlos na edição	N	S	S	S	S	S
Inclui editor de HTML	N	N	N	S	N	S
Suporta ActiveX	N	N	N	S	N	N
Pode criar ActiveX em Java	N	N	N	S	N	N
Consome/Constrói JavaBeans	NN	SS	SS	SS	SS	SS
Fornece Classes para ODBC	S	S	N	S	S	S
Correcção de Erros						
Corrector Integrado	N	S	S	S	S	S
Pode corrigir via Browser	N	N	S	S	N	S
Corrector detecta excepções Java nas linhas	S	S	S	S	S	S
Suporta variáveis de Breakpoint/Watch	SS	SS	SN	SS	SS	SS
Faz Popup de Watches	S	S	N	S	S	S
Faz etiquetagem de datas	S	S	S	S	N	S
Permite Step/Traces	SS	SS	SS	SS	SS	SS
Suporta Correcção Multi-Threading	S	S	S	S	S	S
Suporta Correcção Remota	S	N	S	S	N	S
Contolo Integrado de Fontes	N	S	N	S	S	N
Correcção de JavaScript	N	N	N	S	N	S
Suporta Correcção de HTML	N	N	N	N	N	S
Publicação/Documentação						
Permite criar ficheiros JAR/ZIP	SS	SS	SN	SS	SS	SS
Permite lançar projectos na WEB	N	N	S	S	N	S
Permite Optimizações do Compilador	S	N	S	S	N	S
IDE (Class-Platform Host)	S	N	S	N	S	S
Total de SIMs	21	27	30	36	23	38

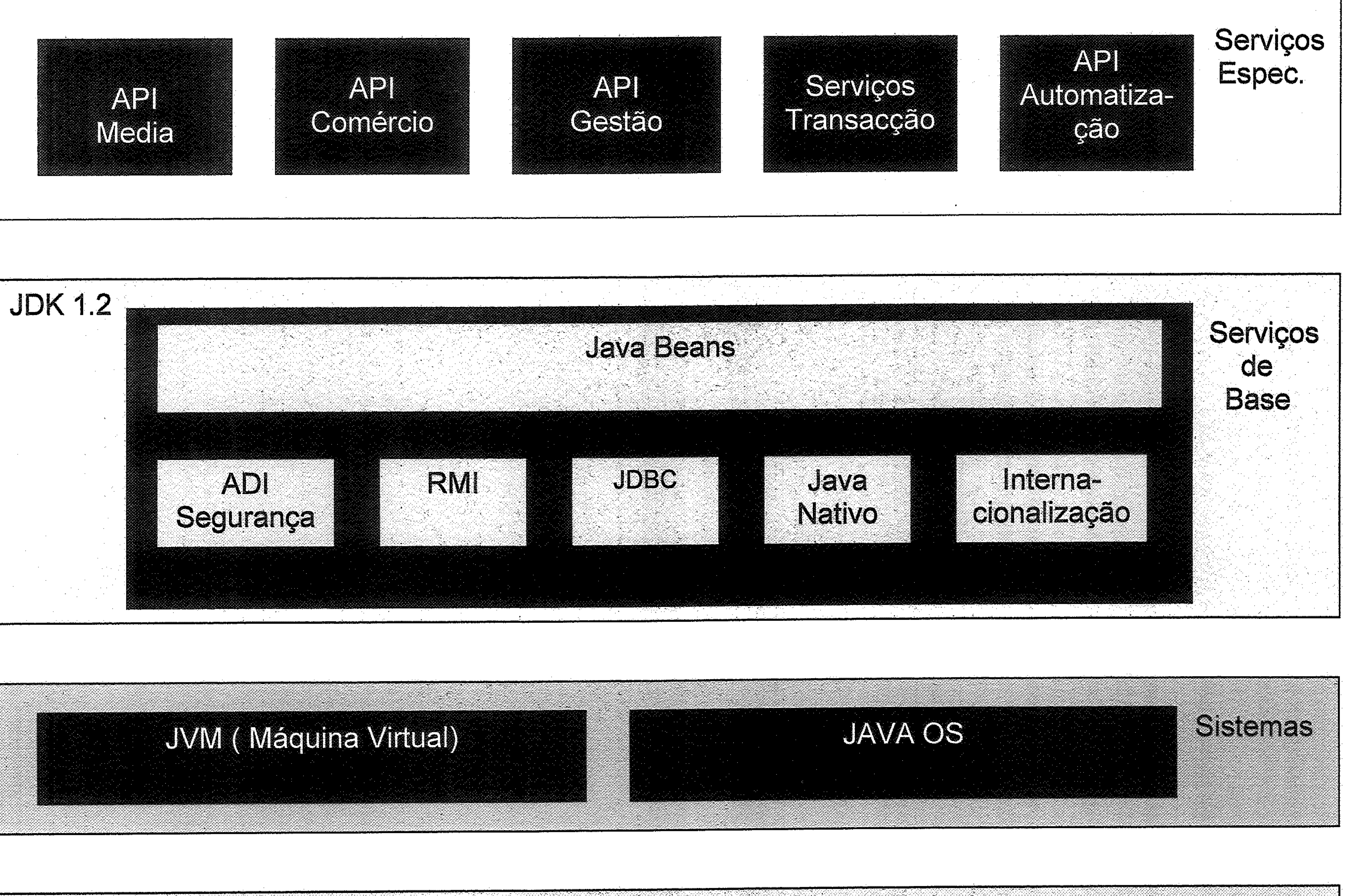
Durante este curso, bem como em cursos futuros, diferentes componentes desta arquitectura Java serão explicados. De momento vamos concentrar-nos nas características da linguagem Java, nas suas bibliotecas mais simples, e na metodologia adequada de criar programas em Java que respeitem e sigam o paradigma da Programação por Objectos.

Para tal introduziremos as principais características de base da linguagem Java, a sua sintaxe, as construções orientadas à Programação por Objectos, designadamente objectos, mensagens, classes, herança, classes abstractas e polimorfismo. Introduziremos ainda, a meio deste percur-so o mecanismo de captura e tratamento de excepções da linguagem.

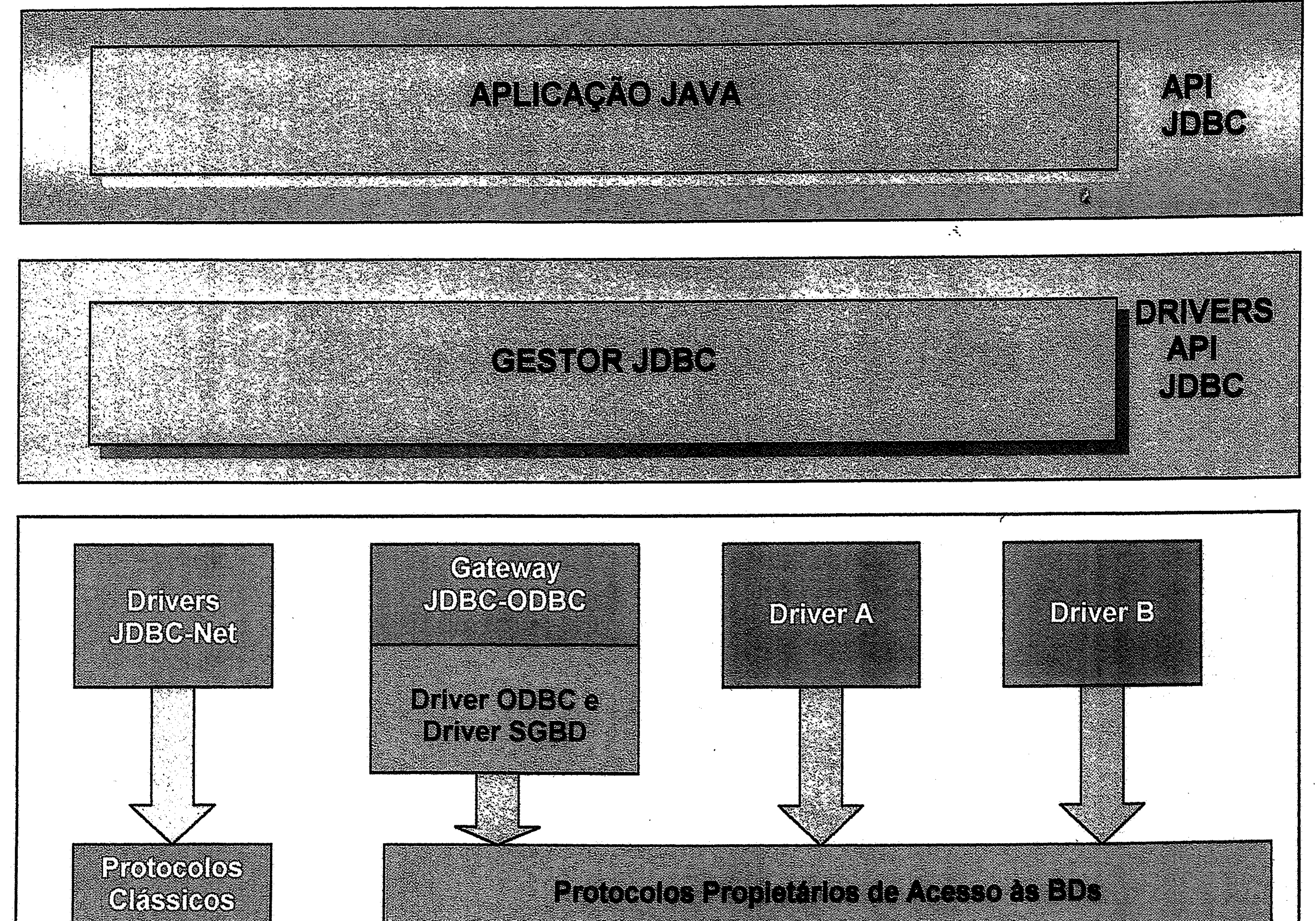
Posteriormente, exploraremos o mecanismo de *Interfaces*, um mecanismo de tipos, tão importante e tão inovador, em particular a forma de o utilizar correctamente no contexto de uma linguagem que já possui *classes* como mecanismo de classificação e tipagem.

Finalmente, serão passadas em revista os principais "packages" de Java e, dentro destes, as classes existentes que mais nos facilitarão, por reutilização, a escrita de programas Java.

ARQUITECTURA DA PLATAFORMA JAVA



ARQUITECTURA DE ACESSO A DADOS COM JDBC



A Linguagem Java: Características

A própria Sun apresenta a linguagem JAVA como possuidora de um conjunto de características que aqui se reproduzem e procuram definir, mas que devemos ao longo do curso permanentemente julgar.

A Sun apresenta JAVA como sendo uma linguagem: *simples, orientada aos objectos, distribuída, interpretada, robusta, segura, neutra em termos de arquitectura, portável, performante, de múltiplas "threads", dinâmica e para a Internet"*.

Analisaremos, de momento, apenas alguns destes atributos:

- **Simples**

Porque de acordo com os seus autores, profundos conhecedores de C e de C++, Java, apesar de ter herdado muitas construções das linguagens C e C++, eliminou um razoável conjunto de construções responsáveis pela "pouca transparência", e portanto grande obscuridade semântica e complexidade, dos programas C e C++. Em particular, os apontadores (cf. *, char *), talvez a eliminação principal, as alocações explícitas de memória (alloc, malloc, dispose, etc.), as instruções para o pré-processador (cf. #define, etc.), os goto, as "header files", e ainda os "templates", usados na construção de estruturas genéricas.

Talvez mais do que simplicidade, dado que, naturalmente sem estas construções alguma redundância terá que surgir, e o código Java tem alguma tendência para a expansão, como veremos, a mais-valia possa ser um inequívoco aumento da clareza e legibilidade dos programas Java relativamente aos programas C ou C++.

No entanto, e convém desde já deixar expressa a opinião de que, porque Java oferece em contrapartida construções visando aspectos de robustez dos seus programas (cf. mecanismos de captura e tratamento de Excepções) e mecanismos alternativos de expressão de propriedades (cf. as interfaces), aliás fundamentais, surge, ainda que para benefício claro do programador, uma complexidade intrínseca à necessidade de domínio de utilização de tais mecanismos.

- **Orientada aos Objectos**

JAVA foi criada como linguagem de Programação Orientada aos Objectos, ao contrário de C++ que consistiu numa quase bem conseguida extensão de C para programação por objectos. Objectos, Classes e Herança Simples existem em Java de forma quase natural. Porém, e para quem teve o privilégio de estudar e programar em Smalltalk, Java não apresenta a "pureza" e a coerência do Smalltalk, dado que Java é, como veremos, uma linguagem de dois níveis: o nível dos tipos de dados e operações sobre estes (cf. tipo int, real, bool, etc.), e o nível em que todas as entidades com existência real são *objectos*. Porém, e inteligentemente, os dois níveis são em Java tornados mutuamente convertíveis, logo representacionalmente compatíveis.

- **Interpretada**

JAVA é, de facto, uma linguagem, em termos de execução final, interpretada. Porém, deste pequeno pormenor vem muita da sua flexibilidade num contexto mundial actual que não dispensa "comunicabilidade", "interoperacionabilidade", "acesso global", "portabilidade", etc., ou seja, soluções de arquitectura software completamente horizontais e adaptáveis a "hardware" investido. O código gerado pelo compilador e a interpretar pelos interrpetadores da linguagem designa-se por *byte-code*. Este código é gerado para ser executado pela Java Virtual Machine, que é, de facto, o software de interpretação.

As empresas que, no passado próximo, realizaram grandes investimentos e apostas em plataformas "menos abertas" (cf. IBM, Apple, etc.) não pretendem agora "ficar de for a" na reengenharia que a tecnologia Java preconiza. Por isso, nada melhor que tal tecnologia ser adaptável a tais arquitecturas e plataformas. Daí muitas das vezes se dizer que JAVA é *neutra* relativamente às arquitecturas.

Um outro aspecto importante relativamente à portabilidade da linguagem, é o facto de Java ter definido como sendo de *dimensões fixas* todos os tipos primitivos, em especial os numéricos, e portanto, tais valores são independentes das máquinas.

A solução Java, embora, conforme se disse atrás, não seja original, é, no contexto actual, inegavelmente ideal, quase se podendo sintetizar da forma "com Java, há de tudo para todos".

- **Robusta**

JAVA possui uma verificação forte de tipos. Não tem "pointers". Acessos a "arrays" e strings são validados pelo compilador. Conversões entre tipos ("casting") são estaticamente verificadas. Possui tratamento explícito de Excepções, ou seja, a ocorrência de uma dada excepção durante a execução do programa pode ser tratada através de instruções próprias codificadas pelo programador. Possui "garbage collection" automático.

- **Segura**

Segurança é hoje em dia uma característica muito importante a ter em conta relativamente aos programas, dado que muitas das vezes estes são de origem desconhecida (cf. via internet). A segurança dos programas tem não só a ver com a garantia de que a sua execução não vai corromper a máquina onde este é executado, mas também com possiveís garantias quanto à sua origem. Assim, Java é segura antes de mais por ser uma linguagem robusta, mas também por possuir processos internos de verificação do *byte-code*. O processo mais seguro de verificação de possível código suspeito consiste em executá-lo sem dar acesso directo ao hardware, o que pode ser feito usando um mecanismo de execução designado *"sand box model"*.
Para além disto, Java permitir incluir chaves criptográficas no próprio código.

- **Performante**

Sendo interpretada, e apesar de muita da execução do código poder ser hoje optimizada à custa de técnicas de compilação e geração de código especiais, JAVA é ainda uma linguagem cuja

performance não pode ser comparável à conseguida por linguagens como C ou mesmo Pascal. Porém, sendo uma linguagem para aplicações distribuídas e sobre a Internet, a sua eventual falta de performance é relativizada se comparada com as velocidades de comunicação e transmissão de dados. Apesar de tudo, um grande esforço de desenvolvimento de compiladores *just-in-time* mais eficiêntes vem sendo realizado.

- **Multithreaded**

Possibilita a execução simultânea de diversos processos leves (*"light processes"*) que realizam diferentes tarefas, por exemplo, apresentar uma animação numa janela e simultaneamente manipular uma "scrollbar".

Possui construções que em muito facilitam a construção deste tipo de programas, em particular construções que permitem realizar a sincronização de tarefas. Como veremos posteriormente, as *applets* - que são porções de código Java que não executam por si próprios, ou seja não são programas, mas que podem ser executadas no contexto de certas aplicações, por exemplo *browsers* ou *viewers* - são construídas usando estes mecanismos.

- **Distribuída e Dinâmica**

Distribuída porque possui um conjunto de classes que implementam mecanismos de acesso remoto (por exemplo *sockets*), as designadas classes de Rede, bem como mecanismos de invocação de métodos remotos, isto é, noutras máquinas, o RMI (de *Remote Method Invocation)*.

Dinâmica porque possibilita que porções de código (em particular classes) sejam apenas "carregadas" para o ambiente de execução em tempo efectivo de execução do programa, através de instruções do próprio programa. Por outro lado, durante a execução dos programas é possível obter informações sobre as classes que constituem o mesmo, o que, como veremos assume uma grande importância para a própria robustez do código final.

- **Para a Internet**

Enfim, pelo somatório de características anteriormente apresentadas.

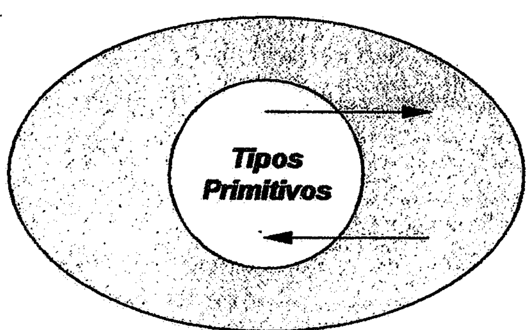
A LINGUAGEM JAVA: INTRODUÇÃO

A linguagem JAVA é, indiscutivelmente, uma linguagem de programação orientada aos objectos (cf. Object-Oriented Language, ou, OOL). Porém, Java, ao contrário de Smalltalk, não é uma linguagem OO "pura", ou seja, em Java nem tudo são *objectos*.

Por isso, torna-se importante desde já deixar claro que na linguagem Java existe um nível em que, tal como em qualquer linguagem não OO, lidamos com entidades (*constantes e variáveis*) que são representadas por valores, e que estão associadas, através de declarações convencionais, a *tipos de dados*. Java possui pois um nível, existente como nível de suporte ao nível fundamental que é o nível das construções OO, em que a programação se faz declarando variáveis e constantes como sendo de dado tipo, manipulando estes valores usando operadores pré-definidos para tais tipos, e até recorrendo a estruturas de controlo comuns nas linguagens de tipo imperativo, como sejam as estruturas de controlo condicionais e repetitivas, aliás seguindo formas sintácticas muito semelhantes - ou até iguais - às definidas na linguagem C.

Este é o nível designado em Java como o nível dos *tipos primitivos* e respectivos operadores, ou, se quisermos, o nível dos *não-objectos*. Fundamentalmente, a este nível trabalhamos directamente com valores. Variáveis contêm valores e é sobre tais valores que realizamos operações e programamos algum controlo de execução. Como veremos mais adiante, este é o nível de suporte à programação orientada aos objectos em Java.

A figura seguinte procura caracterizar esta relação entre o nível dos tipos primitivos e o nível dos objectos e da programação OO.



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

O estudo da linguagem JAVA prosseguirá assim sob duas perspectivas. Em primeiro lugar serão apresentadas as construções relacionadas com o nível dos tipos primitivos, construções em tudo semelhantes às usadas em linguagens do tipo imperativo. De seguida, serão introduzidas, uma a uma, as bases fundamentais do Paradigma da Programação por Objectos, e, para cada uma destas, as correspondentes construções em Java.

NÍVEL DOS TIPOS PRIMITIVOS

Os tipos primitivos Java, sua gama de valores, valores por omissão e dimensão, são sintetizados na tabela seguinte, sendo usado o identificador - ou *keyword* - que permite realizar a declaração de variáveis que irão conter valores de tal tipo.

Tipo	Valores	Default	Bits	Gama
boolean	true, false	false	1	------
char	caracteres unicode	\u0000	16	\u0000 a \uFFFF
byte	inteiro c/ sinal	0	8	-128 a +127
short	inteiro c/ sinal	0	16	-32768 a +32767
int	inteiro c/ sinal	0	32	-2147483648 a 2147483647
long	inteiro c/ sinal	0	64	≈ -1E+20 a 1E+20
float	IEEE 754 FP	0.0	32	≈ ± 3.4E+38 a ± 1.4E-45
double	IEEE 754 FP	0.0	64	≈ ± 1.8E+308 a ± 5E-324

DECLARAÇÕES DE VARIÁVEIS COM E SEM INICIALIZAÇÃO - EXEMPLOS

Forma geral: *id_tipo id_variável [= valor] [, id_variável [= valor] ...] ;*

```
int x;
int x = 10;   /* declaração com inicialização */
int x = 20, y, Z = 30;
int x, y = 10;
int a = x+ y;

char um = '1';
char c = 'A'; /* usando formato UNICODE, caracteres são ASCII compatíveis
*/
char newline = '\n';

boolean fim;
boolean fechado = true;

byte b1 = 0x49 ; /* hexadecimal, cf. 0x, de inicialização */

long diametro;
long raio = -1.7E+5;

double d;
double j = .000000123
double pi = 3.14159273269;
```

Declarações De Constantes

Como veremos posteriormente, as declarações de *constantes* são semelhantes às declarações de variáveis, mas requerem a inclusão de um atributo particular que especifica que tais inicializações são imutáveis, cf. o atributo *final*. As constantes são também em geral identificadas por nomes em letras maiúsculas, o que se aconselha desde já.

Assim, e porque constantes podem ser declaradas em vários contextos, a declaração genérica

```
final double PI = 3.14159273269;
```

define, num dado contexto a estudar posteriormente, uma constante identificada como PI, que assumirá, indefinidamente, o valor real que lhe foi atribuído, ou seja, 3.14159273269;

Conversão entre Tipos - Casting.

Existe um operador em Java, operador (id_tipo) que pode ser usado de forma unária para converter valores de dado tipo à sua direita para valores de variáveis de dado tipo à sua eaquerda, desde que, na tabela de compatibilidades, tal compatibilidade de tipos seja possível.

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

```
int x = 12;
float f;

f = (float) x;
```

é também aceitável que um inteiro possa ser associado ao tipo *char*, e que aritmética de inteiros possa ser usada como forma de manipularmos caracteres. Admitindo de novo declarações tais como:

```
char c1, c2;
char c3 = 'a';
int x = 67;
int y = 4;

c1 = (char) (x + y) ;
c2 = (char) ((int) c3) + 1 );  /* c2, via casting, toma o valor de c3 incrementado */
```

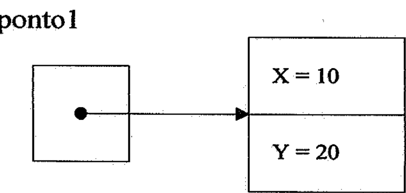
Nota: A compatibilidade entre tipos, em particular aritméticos, é um assunto complexo a ser de novo abordado mais tarde.

NÍVEL DOS TIPOS REFERENCIADOS - OBJECTOS E ARRAYS.

Em Java *tipos não-primitivos* são associados a *objectos* e a *arrays*. Estes tipos não-primitivos são muitas das vezes designados por *tipos de referência,* dado que os seus representantes são tratados por *referência*, ou seja, através de uma variável que, *contendo o seu endereço,* dá acesso indirecto ao seu *valor efectivo*, por oposição às variáveis dos *tipos primitivos* que se limitam a armazenar, ou conter, os *valores* necessários à manipulação.

A figura seguinte mostra o resultado de termos, através de uma declaração Java, associado o identificador de variável *ponto1* a um objecto da *classe* Ponto, usando um mecanismo de classificação e de construção em Java a estudar e analisar posteriormente, cf.

```
Ponto ponto1 = new Ponto(10, 20) ;
```



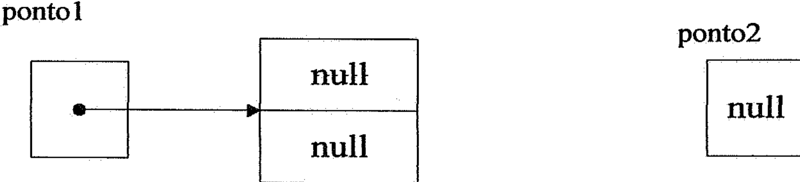
A variável ponto1 foi declarada como podendo conter um objecto da classe Ponto - ***os identificadores de classes iniciam-se por letra maiúscula*** - tendo depois a variável sido associada a um ponto concreto, em particular aquele que foi criado pela frase de construção de objectos da classe Ponto, no exemplo **new Ponto(10, 20)**.

Outros exemplos seriam :

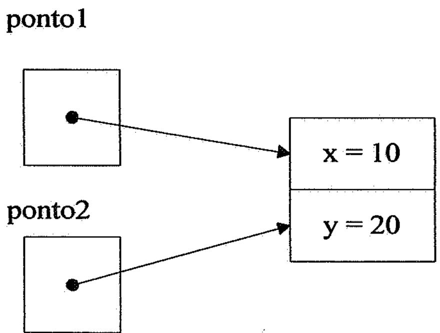
```
Quadrado q1, q2;
ContaBanco c = new ContaBanco( );
```

Admitindo que para criarmos um ponto necessitaríamos de saber logo de início as suas duas coordenadas, Java associa a cada classe criada de nome X um construtor por omissão de nome X( ). Este construtor cria um objecto dessa classe mas não atribui valores aos possíveis campos desses objectos, ficando estes com o valor genérico null. Assim, após as declarações

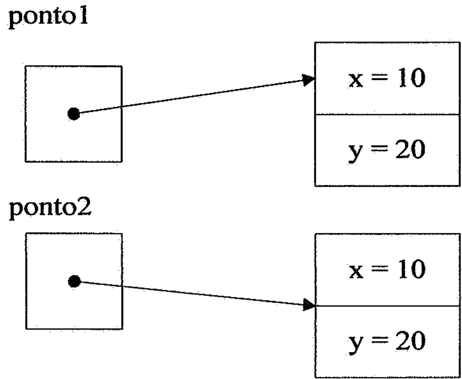
```
Ponto ponto1 = new Ponto( ) ;
Ponto ponto 2;
```



Note-se ainda que, sendo estes objectos referenciados através de variáveis, haverá que ter em atenção que atribuições entre estas variáveis assumem uma *semântica de apontadores*, ou seja, o que é atribuído ou manipulado são referências.
Assim, se fizermos ponto1 = ponto2, ambas as variáveis para a valer null. Por outro lado, se fizermos ponto2 = ponto1, ambas passam a referenciar o mesmo objecto.



Note-se ainda que o resultado do teste ponto1 == ponto2 daria neste caso true, já que ambas as variáveis referenciam neste momento exactamente o mesmo objecto. Porém, se tivéssemos a situação,



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

ARRAYS

Os *Arrays* são em Java entidades *referenciadas* mas *não são objectos* dado que não são, ao invés de todos os *objectos*, criados a partir de uma classe. Porém, por serem manipulados por referência não são associados a tipos-valor mas a tipos referênciados. Em certas situações, a sintaxe das operações de manipulação de arrays assemelha-se à manipulação de objectos.

Arrays são criados em Java dinâmicamente, ou seja, em tempo de execução, e o seu espaço au-tomaticamente reaproveitado quando deixam de estar referenciados. A criação dinâmica de um array faz-se usando igualmente new, indicando ainda o tipo dos seus componentes, e, opcio-nalmente, as suas diferentes dimensões - são multidimensionais. Note-se, no entanto, que esta multidimensionalidade é implementada em Java como *aninhamento*, isto é, arrays multidimen-sionais são, de facto, *arrays de arrays*.

Vejamos alguns exemplos de declarações, com e sem inicializações.

int lista[]; // declaração **"à la C"** de que a variável lista é um array de inteiros;
int[] lista; // declaração equivalente à anterior em que [] se junta ao tipo dos elementos

int[] lista = { 10, 12, 33, 23, 56, 67, 89, 12 }; // declaração com inicialização
int lista[] = { 10, 12, 33, 23, 56, 67, 89, 12 }; // declaração com inicialização

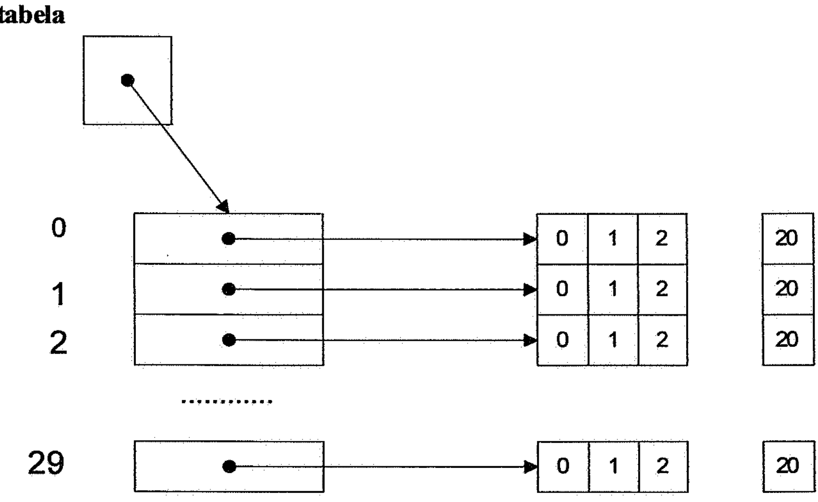
int lista[] = new int[20]; // array de inteiros com 20 componentes e nome *lista*
int[] lista = new int[20]; // sintaxe alternativa para a mesma declaração

byte[] pixels = new byte[600*800]; // pixels é um array de 480000 bytes
byte pixels[] = new byte[600*800]; // pixels é um array de 480000 bytes

String[] texto = new String[200]; // texto é um array de 200 strings
String texto[] = new String[200]; // texto é um array de 200 strings

int[][] tabela = new int[30][20]; // tabela é array bidimensional de 30 linhas x 20 colunas
int tabela[][] = new int[30][20]; // tabela é array bidimensional de 30 linhas x 20 colunas

Esta última declaração define a variável tabela como sendo do tipo int[][]. Dinamicamente aloca um array de 30 elementos, sendo cada um destes elementos do tipo int[20], ou seja, um array de 20 inteiros. Aloca depois os 30 arrays de 20 inteiros, guardando a referência para cada um destes no array de 30 posições. Cada um dos 20 inteiros de cada um dos arrays alocados é inicializado com o valor 0.



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

int[][] matriz = new int[30][];
Neste caso, seria alocado um array de 30 posições, tal como o anterior, mas com referências de valor null, de momento, para cada um dos elementos a alocar nestas posições, sabendo-se

no entanto que tais elementos devem ser do tipo int[], ou seja, arrays de inteiros de dimensão não especificada.

Ao não obrigar à especificação da segunda dimensão, Java flexibiliza e permite que os elemen-tos do tipo int[] deste array possam ter dimensões distintas. Esta flexibilidade não implica para o programador qualquer tipo de descontrolo, cf. saber de facto qual a verdadeira dimensão de cada um dos arrays alocados, dado ser possível em Java saber a dimensão efectiva de um array através da construção sintáctica *array.length*, que dá como resultado a dimensão actual de tal array. Assim, mesmo que tenhamos elementos de tipo array de dimensão diferente, será sempre possível determinar o número efectivo de elementos armazenados, e, portanto, não aceder a elementos inexistentes - o que geraria o erro ArrayIndexOutOfBoundsException. Note-se ainda que em Java os arrays não necessitam de ser "rectangulares", ou seja, de dimensões fixas de n x n.

Finalmente, e quanto às inicializações, elas podem ser feitas na altura da declaração do array, usando a forma sintáctica que se exemplifica a seguir através de diversos exemplos:

```
int[] potencias2 = {2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096,
8192 };

String[] moedas = { "Escudo", "Peseta", "Franco", "Coroa", "Lira", "Libra" };

int[][] matriz = { {1,2}, {10, 20, 30}, {19, 2, 34, 21, 16} } ; // inicializador
aninhado
```

ACESSO AOS ELEMENTOS DE UM ARRAY

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

```
int[] a = new int[20];
byte[] pixels = new byte[600*800];
int[][] tabela = new int[20][];
int x, y;
byte b;


x = a[0]; y = a[a.length-1]; a[0] = x + y;
for(int i=0; i < a.length ; i++) x = x + a[i] ;
......................................// várias operações entretanto realizadas
b = pixels[12*24];
x = tabela[10][2];
..............................................

System.out.println(tabela[1][5]*(int) b + x + a[4]*a[7]);
```

Operadores Disponíveis Sobre Tipos Primitivos

Na tabela seguinte, designam-se por *tipos integrais*, os tipos byte, short, char, int e long. São todos eles, excepto, char, tipos com bit de sinal. Na tabela indica-se a precedência de cada operador, a sua sintaxe, o tipo ou tipos dos operandos, o sentido de associatividade (direita ou esquerda) e o significado.

<table>
<tr><th>Precedência</th><th>Operador</th><th>Tipo Operando(s)</th><th>Assoc</th><th>Operação</th></tr>
<tr><td>15</td><td>.</td><td>objecto, membro</td><td>E</td><td>acesso a membro</td></tr>
<tr><td></td><td>[]</td><td>array, int</td><td>E</td><td>acesso a elemento</td></tr>
<tr><td></td><td>(args)</td><td>método, lista args</td><td>E</td><td>invocação de método</td></tr>
<tr><td></td><td>++, --</td><td>variável</td><td>E</td><td>pós increm/decrem.</td></tr>
<tr><td>14</td><td>++, --</td><td>variável</td><td>D</td><td>pré increm/decrem.</td></tr>
<tr><td></td><td>+, -</td><td>número</td><td>D</td><td>sinal; unário</td></tr>
<tr><td></td><td>~</td><td>inteiro</td><td>D</td><td>complemento de bits</td></tr>
<tr><td></td><td>!</td><td>booleano</td><td>D</td><td>negação</td></tr>
<tr><td>13</td><td>new</td><td>class, lista args</td><td>D</td><td>criação de objectos</td></tr>
<tr><td></td><td>(tipo)</td><td>tipo, qualquer</td><td>D</td><td>conversão ("casting")</td></tr>
<tr><td>12</td><td>*, /, %</td><td>número, número</td><td>E</td><td>mult, divisão e resto</td></tr>
<tr><td>11</td><td>+, -</td><td>número, número</td><td>E</td><td>soma e subtracção</td></tr>
<tr><td></td><td>+</td><td>string, qualquer</td><td>E</td><td>concatenação</td></tr>
<tr><td>10</td><td><<</td><td>inteiro</td><td>E</td><td>deslocam. esq. de bits</td></tr>
<tr><td></td><td>>></td><td>inteiro</td><td>E</td><td>deslocam. dir. de bits</td></tr>
<tr><td></td><td>>>></td><td>inteiro</td><td>E</td><td>deslocam. dir. com 0</td></tr>
<tr><td>9</td><td><, <=</td><td>números</td><td>E</td><td>comparação</td></tr>
<tr><td></td><td>>, >=</td><td>aritméticos</td><td>E</td><td>comparação</td></tr>
<tr><td></td><td>instanceof</td><td>referência, tipo</td><td>E</td><td>teste de tipo</td></tr>
<tr><td>8</td><td>==</td><td>primitivos</td><td>E</td><td>igual valor</td></tr>
<tr><td></td><td>!=</td><td>primitivos</td><td>E</td><td>valor diferente</td></tr>
<tr><td></td><td>==</td><td>objectos</td><td>E</td><td>mesmo objecto</td></tr>
<tr><td></td><td>!=</td><td>objectos</td><td>E</td><td>objectos diferentes</td></tr>
<tr><td>7</td><td>&</td><td>inteiros</td><td>E</td><td>E de bits</td></tr>
<tr><td></td><td>&</td><td>booleanos</td><td>E</td><td>E lógico</td></tr>
<tr><td>6</td><td>^</td><td>inteiros</td><td>E</td><td>OUEXC de bits</td></tr>
<tr><td></td><td>^</td><td>booleanos</td><td>E</td><td>OUEXC lógico</td></tr>
<tr><td>5</td><td>|</td><td>inteiros</td><td>E</td><td>OU de bits</td></tr>
<tr><td></td><td>|</td><td>booleanos</td><td>E</td><td>OU lógico</td></tr>
<tr><td>4</td><td>&&</td><td>booleano</td><td>E</td><td>E condicional</td></tr>
<tr><td>3</td><td>||</td><td>booleano</td><td>E</td><td>OU condicional</td></tr>
<tr><td>2</td><td>? :</td><td>bool, qualq, qualq</td><td>D</td><td>condicional ternário</td></tr>
<tr><td>1</td><td>=</td><td>variável, qualquer</td><td>D</td><td>atribuição</td></tr>
<tr><td></td><td>*= /= %=</td><td>variável, qualquer</td><td>D</td><td>atribuição após oper.</td></tr>
<tr><td></td><td>+= -=</td><td>variável, qualquer</td><td>D</td><td>atribuição após oper.</td></tr>
<tr><td></td><td><<= >>=</td><td>variável, qualquer</td><td>D</td><td>atribuição após oper.</td></tr>
<tr><td></td><td>>>>= &=</td><td>variável, qualquer</td><td>D</td><td>atribuição após oper.</td></tr>
<tr><td></td><td>^= |=</td><td>variável, qualquer</td><td>D</td><td>atribuição após oper.</td></tr>
</table>

INSTRUÇÕES - ESTRUTURAS DE CONTROLO

O controlo mais básico da execução dos programas, ou seus componentes, é em Java, como não podia deixar de ser, expresso sob a forma de *estruturas de controlo* tradicionais. Assim, poderemos expressar controlo de execução à custa das tradicionais estruturas de controlo tais como:

- **Condicionais**: *if/else, switch*
- **Repetitivas**: *for, while, do/while*
- **De sincronização**: *synchronized*
- **De tratamento de erros**: *try/catch/finally*
- **De contexto**: *import, package*

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

if/else

A forma sintáctica genérica é da forma,

if (expressão booleana **)** instruções1; [**else** instruções2;]

sendo, como se indica, [instruções;] opcional. Caso esta não exista, tratar-se-á então de uma simples estrutura condicional do tipo if-then. Se instruções1 corresponder a uma única instrução, então esta apenas necessitará de ser terminada com ;. Caso seja uma sequência de instruções - ou seja, um *bloco* -, então deverá ser tal sequência colocada dentro de { } - ou seja, entre estes delimitadores de blocos de instruções de Java - de facto, herdados do C..

Vejamos alguns exemplos.

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

if (a.length == 0)        // a é um array
   x = -9999;
else
   x = a[a.length-1];   // o último elemento
```

```
if (tipo == 1) {            // este estilo de colocação dos { } é um dos possíveis
   livros = livros + 1;  req = req + 1;
} else {
   naoLivro = naoLivro + 1; consulta = consulta + 1;
}
```

ou até, seguindo um estilo diferente, codificando da forma,

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

ou até, na forma, talvez mais legível, mas nem sempre utilizável de,

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

switch

A instrução switch - existente em C e C++ - é a estrutura de controlo que possibilita a execução de uma dentre várias porções alternativas de código, em função do valor - *simples* - tomado por uma dada expressão ou variável. Trata-se portanto de uma estrutura *case*, que assume a seguinte forma sintáctica genérica :

```
switch (expressão) {
     case valor_1 : instruções ; break;
     case valor_2 : instruções ; break;
     case valor_3 : instruções ; break;

     case valor_n : instruções ; break;
     default : instruções ;
};
```

Assim, em função do valor concreto da expressão indicada - do tipo char, int, short, long e byte - e ainda em função dos diferentes casos que se pretende distinguir, são executadas a ou as instruções indicadas para cada valor constante. A instrução break é utilizada para se garantir que a execução do código da instrução switch termina aí. Caso não seja colocada a instrução break, as instruções associadas ao valor seguinte serão executadas também.

Caso o valor da expressão não seja igual a nenhuma das constantes associadas a cada caso, ao conjunto de instruções associado à palavra chave default - caso exista - é então executado. Vejamos alguns exemplos.

```
switch (caracter) {
  case 'a' :
  case 'A' : { i = (int) caracter; conta[i]++ ; x-- ; break; }
  case 'b' :
  case 'B' : x-- ; break;
  default: c++;
}
```

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

for

A estrutura repetitiva for assume igualmente uma forma sintáctica herdada da linguagem C, e que é, de certa forma complexa dado o conjunto de expressões aceites. A estrutura genérica assume a forma seguinte:

for (inicialização ; condição_ de_ saída ; iteração) *instruções_a_iterar*

O bloco designado inicialização contém em geral a declaração e inicialização da variável que irá servir como variável de controlo do ciclo. A expressão designada como condição_ de_ saída, tem exactamente a missão de declarar uma expressão *booleana* que será avaliada a cada iteração, e que conduzirá ao fim da iteração logo que o seu valor seja falso. Note-se que caso esta expressão tenha o valor falso logo no início, ou seja, antes da primeira execução, nada será executado. O bloco designado por iteração contém as declarações que indicam de que forma os valores das variáveis evoluem de uma iteração para a outra.

Note-se que todas as expressões indicadas são opcionais, pelo que poderíamos ter, no limite, a seguinte estrutura for

```
for ( ; ; ) { ... }
```

que conduziria a um ciclo infinito, já que a condição de saída, se omitida, é equivalente a true. A única forma de terminarmos um ciclo infinito destes seria através da utilização de uma instrução break algures dentro do bloco de código a iterar. Esta é no entanto uma programação que se desaconselha vivamente dada a falta de clareza implícita.

Exemplos:

```
int soma = 0;
for (int i = 0; i < a.length; i++) {    // somatório dos elementos de um array
   soma = soma + a[i];
};
```

```
int temp;
for (i = 0, j = a.length - 1 ; i < j ; i++, j--) {        // inversão de um array
   temp = a[i] ;
   a[i] = a[j];
   a[j] = temp;
};
```

while

A estrutura repetitiva while tem a forma sintáctica,

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

e a sua semântica é muito simples: enquanto a condição_de_iteração for verdadeira o corpo do ciclo é executado. A condição_de_iteração pode ser ou não verdadeira no início da ite-ração. Se for falsa, nada será executado, e o programa continua na instrução seguinte. Se for verdadeira, a primeira iteração é realizada, ou seja, todo o corpo definido é executado, sendo de seguida a condição_de_iteração reavaliada. Naturalmente que se nenhum valor das vari-áveis usadas para exprimir esta condição_de_iteração for modificado no corpo, então esta-mos numa situação de ciclo infinito.

Note-se que sendo a condição_de_iteração testada antes da primeira execução do corpo, pode acontecer que, sendo a primeira imediatamente falsa, este não seja executado. Assim, a principal característica determinante na utilização de uma estrutura repetitiva do tipo while, é que a mesma conduz a *0 ou mais iterações* do corpo de instruções a iterar. O número de itera-ções que serão efectivamente realizadas não pode nestes casos ser expresso em função dos su-cessivos valores tomados por uma variável de controlo. Quando tal pode ser expresso de tal forma, a estrutura aconselhada é uma estrutura repetitiva do tipo for. É no entanto curioso ve-rificar que os programadores de C, por erro conceptual que não se aconselha que seja reprodu-zido em Java, implementam estruturas while usando estruturas for.

Aconselha-se vivamente que, sempre que pretendermos expressar uma iteração que pode ocorrer 0 ou mais vezes, se expresse tal facto usando uma estrutura while. Sempre que pretender-mos expressar uma iteração que ocorre tantas vezes quantos os valores que uma dada variável deve tomar, quaisquer que sejam os incrementos, para, partindo de um dado valor, atingir outro, então tal deve ser expresso usando uma estrutura for.

A adopção destas regras simples melhorará, sem dúvida, a clareza do código final.
Apesar de tudo, e tendo em atenção a semântica de ambas as construções, poder-se-á dizer que a construção

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

é equivalente à construção,

```
expressão1;
while (expressão2) {
   instruções;
   expressão3;
}
```

Vejamos agora alguns exemplos de aplicação da estrutura repetitiva while.

```
x = a.length;
while (x >= 0) {
   s = s + a[x];
   x--;
}

found = false;
x = 0;
while (!found & x < a.length-1) {  // procura de um elemento num array
   if ( a[x] == chave )
          found = true;
   else
          x++;
}
```

do/while

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

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

Exemplo:

```
do {
      s = s + a[i]; i++ ; }
while( i < a.length ) ;
```

CLASSES e HERANÇA.

(PPO III)

- ☞ **DEFINIÇÃO de CLASSE.**
- ☞ **RELAÇÃO CLASSE-INSTÂNCIAS.**
- ☞ **MECANISMO DE INSTANCIAÇÃO.**
- ☞ **CLASSES E SUPERCLASSES.**
- ☞ **HIERARQUIA e HERANÇA.**

☞ DEFINIÇÃO de CLASSE.

- CLASSES são objectos-padrão que servem para definir a **ESTRUTURA COMUM** (variáveis de instância ou estado) e o **COMPORTAMENTO COMUM** (métodos ou rotinas) de todos os objectos (instâncias) que a partir destas se podem criar.

- CLASSES podem ser vistas como descrevendo UMA particular implementação de um TAD (Tipo Abstracto de Dados).

- CLASSES representam um poderoso mecanismo de "partilha de código" já que os métodos a que respondem todas as instâncias de uma CLASSE possuem o seu código associado à CLASSE, logo sem duplicados.

- CLASSES são igualmente um mecanismo de CLASSIFICAÇÃO, muitas das vezes associado ao mecanismo bem conhecido de TIPOS das linguagens convencionais.

- Cada INSTÂNCIA de uma CLASSE é um objecto auto-identificado dado que transporta consigo, o seu TIPO ie. a CLASSE a que pertence.

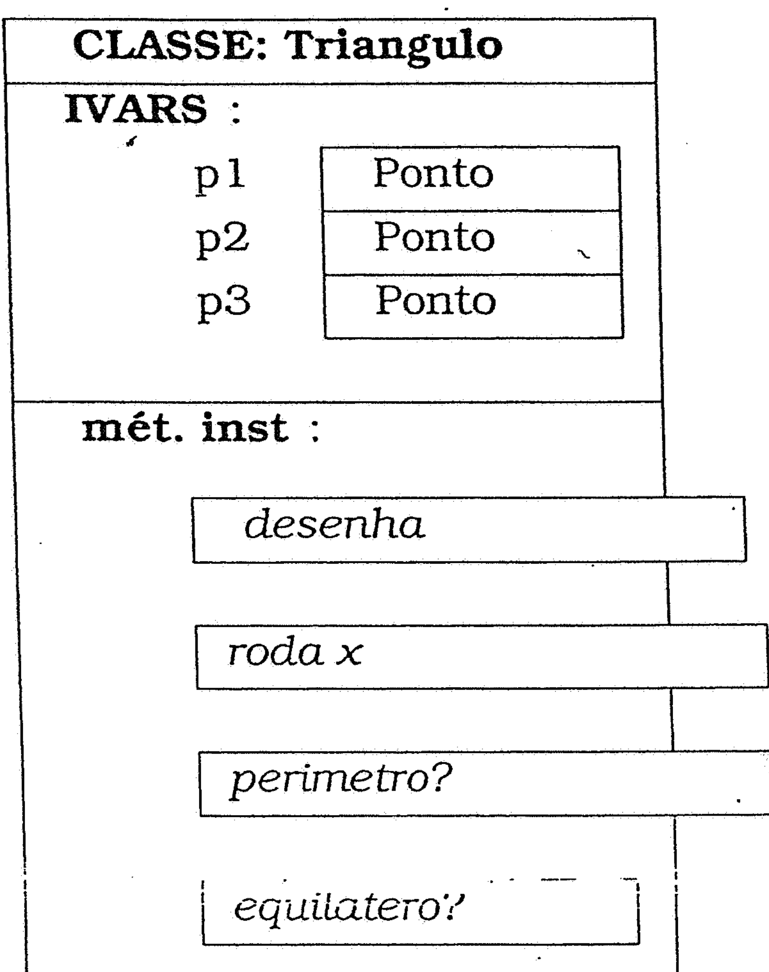
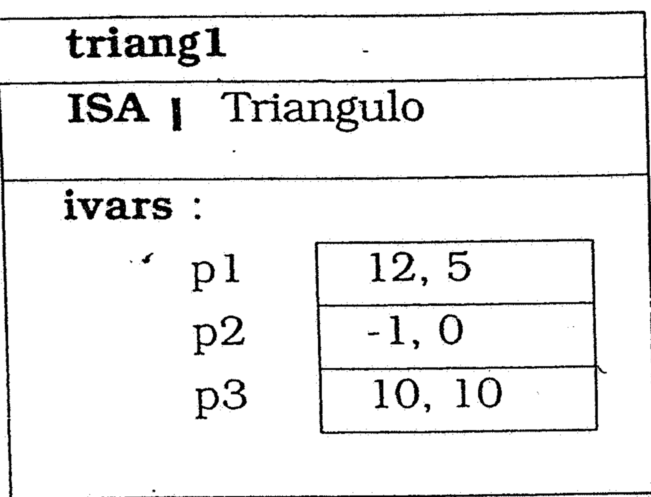


Fig. 17 - UMA CLASSE.



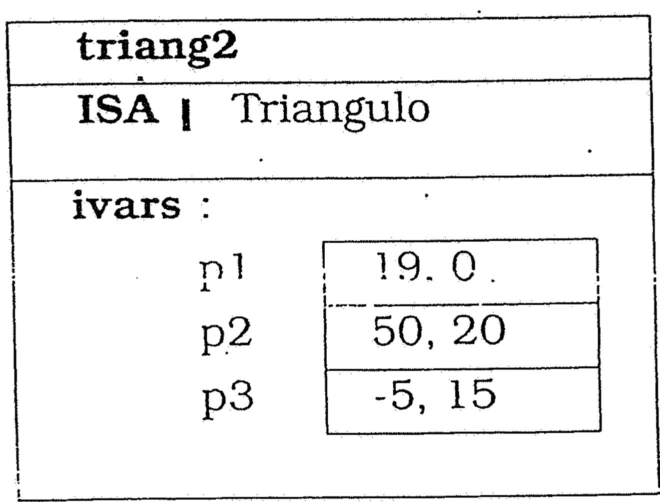


Fig. 18 : DUAS INSTÂNCIAS DA CLASSE.

✏ Uma estrutura de CLASSE como a anterior permite definir completamente a ESTRUTURA e o COMPORTAMENTO de qualquer instância que venha a ser criada.

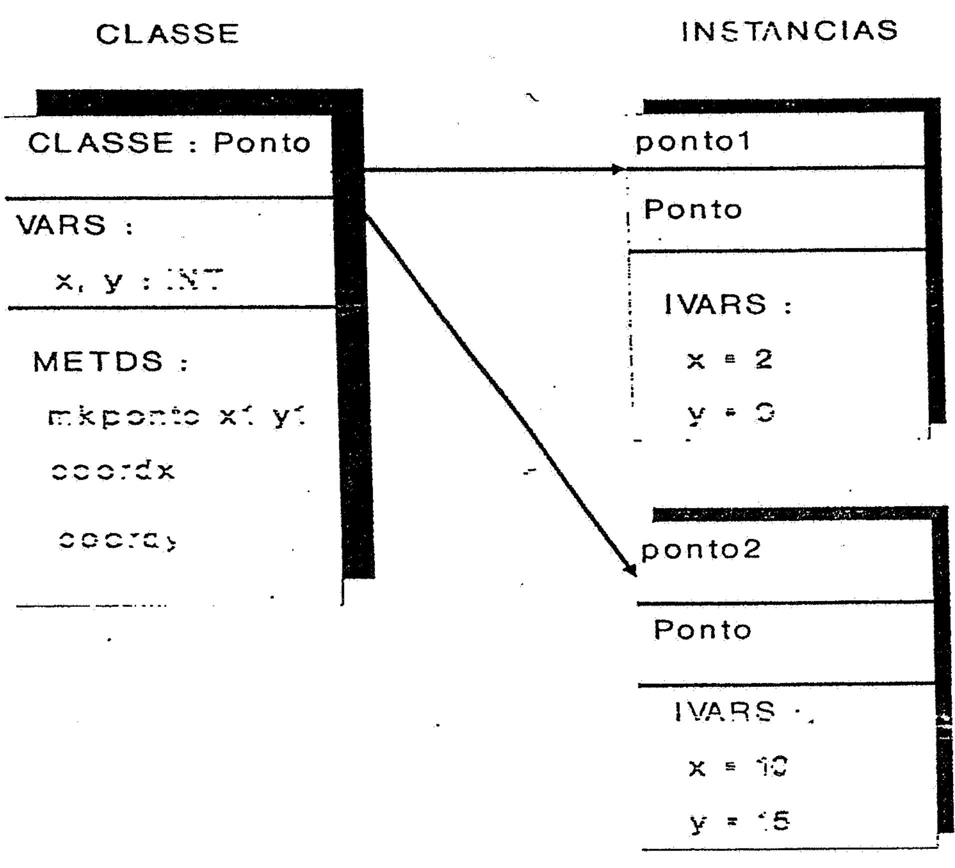


Fig. 19 : Relação Classe-Instâncias.

3.2 Criação de Classes em JAVA: Introdução

A noção de CLASSE, tal como foi definida anteriormente, é suficientemente clara para que, de imediato, possam ser criadas algumas classes simples em JAVA, 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 alguns outros conceitos muito importantes em PPO, vendo como os mesmos estão representados na linguagem JAVA.

Requisitos Iniciais

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

- *Os contadores deverão ser contadores de tipo inteiro;*
- *Deverá ser possível criar contadores com valor inicial igual a 0;*
- *Deverá ser possível criar contadores com valor inicial igual ao valor dado como parâmetro;*
- *Deverá ser possível saber qual o valor actual de um dado contador;*
- *Deverá ser possível incrementar o contador de 1 unidade ou de um valor dado como parâmetro;*
- *Deverá ser possível decrementar o contador de 1 unidade ou de um valor dado como parâmetro;*
- *Deverá ser possível obter uma representação textual de um contador;*

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 contadores inteiros que pretendemos, ou seja, pensarmos quais deverão ser as suas *variáveis de instância* (os seus nomes e os seus tipos).

Neste caso, cada *contador* deverá ter apenas que ser capaz de conter um valor de tipo *inteiro* correspondente à contagem que tal contador representa. Assim sendo, os contadores que pretendemos criar necessitam apenas de ter uma variável de instância a que vamos dar o nome de `conta` e que vai ser do tipo `int`.

Vamos, então, ver como gradualmente vai evoluindo a classe `Contador` 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. Tais definições consistem, por agora, na declaração das diversas variáveis de instância e na codificação de cada um dos métodos de instância. Assim, partindo-se de um esqueleto básico contendo apenas a identificação da classe:

```
class Contador {
// definições
}
```

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

```
class Contador {
// variáveis de instância
  int conta;
}
```

Definição do Comportamento. Os Construtores de Instâncias

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*, e que são 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 dada classe tenha já sido definida, por exemplo uma classe designada `Triangulo`, é de imediato possível criar instâncias dessa classe se usarmos uma das seguintes construções sintácticas:

```
Triangulo t1 = new Triangulo();
```

ou

```
Triangulo t1;
t1 = new Triangulo();
```

Em ambos os casos começamos por declarar uma variável `t1` como sendo do tipo `Triangulo`, 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 Triangulo()`. Esta expressão, que traduzida se torna clara, corresponde à criação de *uma nova instância* da classe `Triangulo` pela execução do método `Triangulo()`.

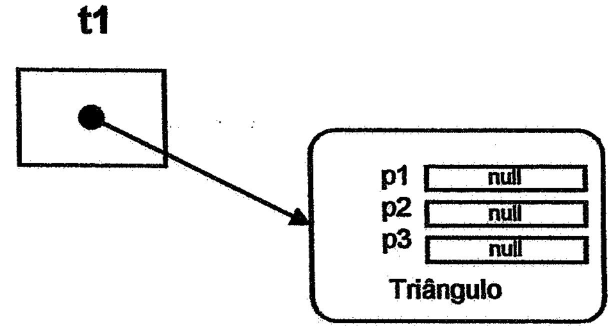


Fig. 3.2 – Instância de Triângulo referenciada por t1

A nova instância é referenciada através da variável `t1`. O método especial `Triangulo()` que em conjunto com `new` é capaz de construir instâncias da classe `Triangulo` designa-se por *construtor*.

Em JAVA, para cada classe definida de nome **X** é automaticamente disponibilizado um construtor **`X()`**, capaz de criar instâncias de tal classe usando a expressão predefinida **`new X()`**. Porém, este construtor implementado pela linguagem JAVA por omissão, ou seja, por ausência de outros mais específicos, tem a característica particular de atribuir o valor `null` a todas as variáveis de instância do objecto que foi por si criado que não sejam de um tipo simples, tal como ilustrado na Figura 3.2.

Não sendo o valor `null` um valor manipulável ou tratável por não ter qualquer tipo a si associado, seja simples seja classe, este construtor de instâncias oferecido por omissão em JAVA tem, como se comprova, uma utilidade prática muito reduzida. Por tal razão, JAVA permite que tal construtor possa ser redefinido pelo programador na definição de uma dada classe, desde que o programador associe um comportamento diferente a outro construtor do mesmo nome.

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

```
Contador conta1 = new Contador();
```

associaria à variável `conta1` uma instância de *contador* que teria o valor inicial 0 na sua variável de instância `conta`, dado esta ser do tipo `int`. Como facilmente se compreende, este é um valor inicial adequado para um objecto que pretende ser um contador de inteiros. Porém, gostaríamos que o valor inicial atribuído a um objecto que é um contador de inteiros positivos fosse indicado de forma explícita no seu construtor, para que não houvesse qualquer ambiguidade.

Portanto, e em função da análise realizada, vamos dar uma definição nossa para o construtor `Contador()`, 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` à variável de instância do contador. O código do mesmo será simplesmente:

```
Contador() {
     conta = 0;
}
```

Os *construtores* de uma classe são todos os 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, dado criarem instâncias de uma dada classe, não têm, obviamente, que especificar qual o seu resultado, que será sempre uma instância da respectiva classe.

Por outro lado, é sempre possível, e muitas vezes bastante útil, definir mais do que um construtor de instâncias de uma dada classe, construtores que, em geral, apenas diferem nas inicializações realizadas.

Se pretendermos no exemplo em questão criar instâncias da classe `Contador` tendo na sua variável de instância valores iniciais dados como parâmetro, então poderemos criar um construtor adicional que receba um número inteiro como parâmetro e atribua tal valor à variável de instância da instância criada.

Teríamos, neste caso, o seguinte construtor:

```
Contador(int val) {
      conta = val;
}
```

A nossa classe `Contador` tem neste momento a seguinte estrutura:

```
class Contador {
   // construtores

   Contador() {
         conta = 0;
   }
   Contador(int val) {
         conta = val;
   }
   // variáveis de instância
   int conta;

}
```

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 `Contador` quando recebem as mensagens a que são 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 particular, as mesmas tenham que ser introduzidas.

Vamos, desde já, introduzir uma regra crucial para uma correcta programação em PPO, regra já 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 que para tal devem ser programados em cada uma das 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.

Note-se ainda que estas regras correspondem a princípios de programação que não são verificados pelos compiladores. De facto, mesmo as mais evoluídas linguagens de PPO possibilitam, ou seja, não detectam em tempo de compilação, situações de completa infracção desta regra tão importante.

Por exemplo, em JAVA, e usando o que apenas foi até agora definido para a classe `Contador`, qualquer classe "cliente" da classe `Contador` poderia ter um método que, necessitando de ter o valor actual da variável de instância `conta` da instância da classe `Contador` referenciada pela variável `ct1`, conteria o código seguinte:

```
int c = ct1.conta;
```

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

```
ct1.conta = 22;
```

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 `Contador` caso a classe `Contador` seja redefinida, por exemplo, alterando simplesmente o nome da variável de instância ou, pior ainda, a representação do valor inteiro, é que se torna inutilizável dado usar uma representação que é agora errada. Este tipo de código, que não obedece aos princípios básicos do encapsulamento, é *dependente do contexto*, pelo se torna caduco logo que o contexto seja modificado.

No limite, se todas as classes forem dependentes da representação interna de uma outra classe qualquer, e sê-lo-ão caso a prática de programação seja esta, à menor alteração de uma delas todas as outras se tornam inutilizáveis.

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 a frase sintácticamente correcta para o respectivo compilador, *c.x*.

Obviamente que, dentro dos princípios de programação que vimos defendendo para que, de facto, possamos atingir certos objectivos no desenvolvimento de aplicações, 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 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 nome genérico X devem ser designados por **getX** (p.ex. `getNome()`, `getConta()`, etc.). Por outro lado, todos os métodos que alteram o valor de uma variável genérica X devem ser designados por **setX** (p.ex. `setNome()`, `setConta()`, etc.).

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 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, designam-se por *modificadores* e, em geral, têm parâmetros de entrada mas não devolvem qualquer resultado.

Em JAVA, a definição de qualquer método consiste na definição do seu *cabeçalho* ("header") e do seu *corpo* ("body"). A estrutura mais básica de declaração em JAVA do cabeçalho de um método é a seguinte:

<tipo de resultado> <identificador> (<pares tipo-nome: parâmetros>)

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

O *<identificador>* de um método é uma sequência de letras e dígitos, devendo o primeiro caracter ser uma letra, em geral minúscula.

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

Apresentam-se, em seguida, alguns exemplos deste tipo de cabeçalhos:

```
int getConta()
boolean maiorQue(int valor)
void setValor(int valor)
boolean valorEntre(real min, real max)
```

A **assinatura** (*signature*) de um método é constituída pelo nome do método e pelo número, tipo e ordem dos seus parâmetros. 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.

O **corpo de um método** é definido como sendo um bloco de instruções, tendo por delimitadores as usuais { }. Caso o método devolva um resultado, este será o valor resultante do cálculo do valor da expressão que está associada à instrução `return`, instrução esta que, em boa programação, aparece uma só vez no código do método, em geral no fim do mesmo, sendo sempre a última a ser executada.

Retomando o exemplo da classe `Contador`, podemos agora escrever o tal método de instância que deve poder ser invocado do exterior, logo sendo público, e que deve devolver como resultado o valor actual da variável de instância `conta`.

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

```
int getConta() {
     return conta;          // interrogador - selector
}
```

Como se pode verificar, o método limita-se a devolver o valor da variável de instância `conta`. Assim, e mais uma vez admitindo que a variável `ct1` referencia uma instância de `Contador`, qualquer objecto externo passa a poder ter acesso ao valor da variável de instância de `ct1`, escrevendo o código que corresponde ao envio da mensagem que activa o método `getConta()`, por exemplo:

```
contagem = ct1.getConta();
```

Vamos agora desenvolver outros métodos que, conforme os requisitos originais, se pretendem ver implementados na classe `Contador`. Assim, pretende-se programar um método modificador que incremente de uma unidade o valor da variável onde é guardada a contagem actual de tal contador. Tal método terá como resultado `void`, será designado por `incConta` e não terá parâmetros de entrada.

```
void incConta() {
     conta = conta + 1;     // modificador do estado
}
```

Numa versão mais familiar aos programadores de C, teríamos:

```
void incConta() {
     conta++;                // modificador do estado
}
```

Não devolvendo este método qualquer resultado, a sua invocação no código de um objecto exterior teria, por tal razão, a seguinte forma:

```
ct1.incConta();
```

ou seja, o objecto exterior altera o estado interno de `ct1`, mas em resultado de tal reacção do receptor não necessita de receber qualquer resultado. Note-se mais uma vez, pelo exemplo, a similaridade entre a *forma sintáctica da mensagem* e a *assinatura* do método.

Vamos em seguida desenvolver o código do método modificador que permite que a variável de contagem seja incrementada de um número de unidades que é dado como parâmetro. O código do método será, neste caso:

```
void incConta(int x) {
     conta = conta + x;    // modificador do estado
}
```

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 antropomorfismo aos objectos que devemos programar, ou assumindo nós a sua posição de serviço, em qualquer dos casos procuramdo 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, em função dos requisitos apresentados, à mensagem m que me foi enviada ?*

Esta noção de ter que programar à *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.

MÉTODOS DE INSTÂNCIA: O *OVERLOADING* OU SOBRECARGA DE NOMES

Note-se que na definição da classe `Contador` acabamos por definir dois métodos que usam o mesmo identificador `incConta`. Permitirá o compilador de JAVA que dois diferentes métodos possam ter o mesmo nome dentro da mesma definição de uma dada classe, ou seja, na definição do comportamento visível do exterior das suas respectivas instâncias?

A resposta é sim, permite, e ainda bem que permite. Tal como muitos operadores conhecidos de outras linguagens (cf. o típico + para adição e concatenação), em JAVA, tal como na maioria das linguagens de PPO, é possível definir dentro da mesma classe métodos tendo o mesmo nome mas diferentes parâmetros formais de entrada, técnica que se designa por **sobrecarga de métodos** (*method overloading*) e que é extremamente útil na identificação de funcionalidade semelhante, ainda que a partir de dados diferentes.

Como vimos anteriormente com os construtores da classe `Contador`, os métodos construtores são métodos particulares que estão sempre em sobrecarga, dado que são mesmo obrigados a ter o mesmo nome, que é até o identificador da classe. No entanto, esta possibilidade técnica torna-se extremamente útil, dado que, em geral, pretendemos ter múltiplas e diferentes possibilidades de construir instâncias de uma dada classe.

Com os métodos de instância passa-se exactamente o mesmo e apesar de o nome de um método dever estar sempre associado a, e ser identificativo de, uma dada computação, este não é univocamente caracterizador de tal computação, dado que, em geral, tal computação pode ser realizada de diferentes formas em função dos diferentes tipos de parâmetros. Assim, *overloading* de nomes de métodos em PPO é uma possibilidade de grande importância, porque, sem se perder na relação entre nome e função, ganha-se em flexibilidade e em normalização.

Tal é possível em PPO, em primeiro lugar porque em PPO métodos e mensagens são entidades distintas. As mensagens são as entidades que são responsáveis pela activação da computação programada num dado método. Assim, quando se envia uma mensagem com um certo identificador e uma certa lista de argumentos a um objecto, a determinação de qual o método que deve ser executado pelo objecto receptor depende da compatibilidade entre a estrutura da mensagem recebida e as assinaturas dos métodos pelo mesmo tornados acessíveis. É pois escolhido para execução o método cuja assinatura corresponda à estrutura da mensagem recebida quanto ao nome, número, tipo e ordem dos parâmetros actuais de tal mensagem.

Por outro lado, dois métodos definidos na mesma classe são considerados distintos desde que tenham assinaturas distintas. Sendo a definição de assinatura mais rica que a definição do identificador do método, tal significa que em JAVA podemos ter métodos distintos, ainda que possuam o mesmo identificador. A distinção, caso seja necessária, será portanto feita pelo número, tipo e ordem dos seus parâmetros formais.

Adicionalmente, e dado que o tipo de resultado de um método não entra na sua assinatura, JAVA permite ainda que métodos sobrecarregados, ou seja de igual nome, possam ter até tipos de resultado distintos, desde que possuam assinaturas distintas. Assim, por exemplo, na classe `Contador` poderíamos definir, sem quaisquer conflitos, métodos de incremento do valor interno da variável `conta` tão díspares e "estranhos" quanto os que a seguir se apresentam:

```
void incConta(real x) {
    conta = conta + (int) x;
}
```

```
String incConta(float x) {
     conta = conta + (int) x;
     return (new String("conta = " + conta));
}
```

Tal só é possível porque, muito embora os quatro métodos definidos tenham por identificador `incConta` e todos, de facto, realizem uma alteração de estado, as suas assinaturas são todas diferentes, daí que os tipos de resultado também possam ser, dado não existirem colisões de definições.

Vamos agora completar a definição da classe `Contador` definindo os restantes métodos de instância, usando já o conjunto de conhecimentos que foram sendo adquiridos ao longo do exemplo.

Definiremos os métodos que decrementam o valor de um contador, usando já sem qualquer problema a técnica de *sobrecarga*, como sendo:

```
void decConta() {
     conta = conta - 1;          // ou conta--
}
```

ou ainda,

```
void decConta() {
     conta -= 1;
}
```

e também,

```
void decConta(int x) {
     conta = conta - x;
}
```

ou

```
void decConta(int x) {
     conta -= x;
}
```

Vamos agora analisar como poderemos satisfazer o requisito de apresentar sob a forma de uma *string*, ou sequência de caracteres, representável em JAVA como uma instância particular da classe `String` predefinida, uma dada instância desta classe `Contador` sob a forma de, por exemplo, `Contador = 27`.

```
// devolve uma String em representação do contador
     String toString() {
       return (new String("Contador = " + this.getConta()));
     }
```

Analisemos com atenção a expressão associada à instrução `return`, dado conter algumas construções interessantes de JAVA. Em primeiro lugar, note-se que tudo se

passa à volta da construção para criação de uma instância da classe `String` tal como a frase `new String(...)` indica. Este construtor particular permite que se passe como parâmetro o valor inicial da *string* como, por exemplo, em:

```
String nome = new String("Rita Isabel");
```

que, de facto, é equivalente à expressão:

```
String nome = "Rita Isabel";
```

O operador + é o operador que realiza a concatenação de *strings*. No entanto, o operador é *overloaded*, permitindo operandos de tipos simples, que converte numa *string* para depois realizar a concatenação, tal como, por exemplo, nas expressões:

```
String nome = "Pedro Nuno tem " + 13 + "anos";
```

No entanto, na expressão que estamos a analisar, surge a subexpressão,

```
"Contador = " + this.getConta()
```

onde o operador + tem por operandos uma *string* e o resultado da expressão `this.getConta()`. Sabemos que pretendemos ter como segundo operando o valor inteiro que é o valor actual da variável de instância `conta`. Qual então o real significado da expressão `this.getConta()`?

A REFERÊNCIA `this`

No exemplo anterior, no código do método `toString()` da classe `Contador` que estamos a desenvolver, surge a expressão `this.getConta()` que temos que analisar. Esta expressão surge da necessidade de termos que invocar o código do método `getConta()` dentro do código de um método do mesmo objecto. Ora uma mensagem deve ser sempre enviada a um receptor bem definido, pelo que se coloca o problema de como referenciar o próprio objecto no seu próprio código, sendo certo que a variável que posteriormente o vai referenciar é desconhecida.

Em JAVA, este problema é resolvido criando uma *referência especial* para cada objecto de nome **`this`**, que é de facto uma forma de *auto-referência* utilizável não do seu exterior, mas no seu interior.

Através desta referência especial **`this`**, qualquer objecto pode aceder aos seus métodos e até às suas variáveis de instância (cf. `this.conta`, `this.x`, etc.), sendo evidente quando se trata de referência a um método ou a uma variável de instância dada a inexistência de `()` neste segundo caso.

Assim, a expressão `this.getConta()` corresponde ao envio da mensagem que activa o método `getConta()`, ao mesmo objecto que foi o receptor da mensagem que activou o método `toString()` onde este `this` aparece.

Por exemplo, no código seguinte as expressões que se sublinham a negrito

```
String strVal;
Contador ct1, ct2;
ct1 = new Contador(10);
ct2 = new Contador(20);
strVal = ct1.toString();
strVal = ct2.toString();
```

correspondem à execução do mesmo método sobre dois contadores referenciados por `ct1` e `ct2`. A primeira expressão tem como receptor da mensagem o objecto referenciado por `ct1`. Então, ao ser executado o código de tal método, a expressão `this.getConta()` referencia de forma inequívoca o receptor original, pelo que, tal como se pretende, acaba por ser equivalente a `ct1.getConta()`. Na segunda, sendo o receptor `ct2`, `this.getConta()` torna-se equivalente a `ct2.getConta()`. Em ambos os casos, `this` garante que o método a ser procurado para execução é um método do receptor da mensagem original.

De facto, JAVA acrescenta de forma automática a referência `this` ao conjunto de parâmetros de um qualquer método de instância, como referência à instância que foi a receptora da mensagem. Qualquer que seja o código executado em resposta a tal mensagem, seja de um método da classe da instância ou de uma superclasse, se tal código usar a referência `this`, tratar-se-á sempre da instância que recebeu a mensagem que despoletou a execução de tal código.

No exemplo, os resultados de tal execução seriam respectivamente os seguintes:

```
strVal  = "Contador = 10"
strVal  = "Contador = 20"
```

Finalmente, podendo as variáveis de instância ser directamente referenciáveis pelo código dos métodos de instância da mesma classe, dado serem membros da mesma definição de classe e não existir neste caso qualquer vantagem em que o acesso não seja directo, o código do método `toString()` poderia ser de facto escrito como:

```
    // devolve uma String em representação do contador

    String toString() {
     return (new String("Contador = " + conta));
  }
```

DEFINIÇÃO DA CLASSE Contador

```
class Contador {

    // construtores

    Contador() {
        conta = 0;
    }

    Contador(int val) {
        conta = val;
    }

    // variáveis de instância

    int conta;

    // métodos de instância

    int getConta() {
        return conta;           // interrogador - selector
    }

    void incConta(int x) {
        conta = conta + x;      // modificador do estado
    }

    void incConta() {
        conta = conta + 1;      // modificador do estado
    }

    void decConta() {
        conta = conta - 1;      // modificador do estado
    }

    void decConta(int x) {
        conta = conta -x;       // modificador do estado
    }

    String toString() {
   return (new String("Contador = " + conta));
  }
}
```

PACKAGES: AGRUPAMENTO 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 `Contador` teria pois a designação *Contador.java*. Após a compilação de uma classe, é gerado um ficheiro de *bytecodes*, designado por ficheiro de classe, que terá o mesmo nome da classe compilada mas extensão *.class*. A compilação do ficheiro *Contador.java* daria, pois, origem ao ficheiro *Contador.class*, pronto a ser executado pelo interpretador de JAVA.

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. Tal não apenas permite que o mesmo identificador de classe possa ser usado em classes pertencentes a *packages* diferentes, como também 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 *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* reune todas as classes fundamentais à execução de programas JAVA. Em JAVA 2 existem definidos 59 diferentes *packages*. Ao longo deste livro necessitaremos, no máximo, de utilizar classes de 6 destes *packages*.

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

```
java.lang.String.size()
java.io.System.out.println("abc");
```

Porém, sempre que na definição de uma classe necessitarmos de usar classes pertencentes a um dado *package*, poderemos sempre usar uma **cláusula de importação** existente em JAVA, que torna os elementos importados disponíveis na definição da classe que os importa, podendo a partir daí as classes e respectivos membros importados serem designados pelos seus nomes abreviados (sem haver necessidade de usar prefixos qualificadores).
A cláusula de importação permite realizar importações qualificadas (específicas), em qualquer número, tal como por exemplo em:

```
import java.util.Vector;
import java.lang.String;
```

bem como importações globais, tal como em:

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

estando sempre implícita em qualquer programa JAVA a cláusula de importação que permite importar todas as classes fundamentais à execução, pelo que se torna redundante escrever a cláusula de importação:

```
import java.lang.*;
```

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 a 2.6 sobre instalação do JDK) as directorias onde tais classes devem ser posteriormente 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 à directoria actual de trabalho, desde que na variável CLASSPATH tal caminho seja referenciado através do símbolo ".".

REGRAS DE ACESSIBILIDADE A UMA CLASSE

JAVA fornece mecanismos de **controlo de acesso** 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** possíveis de serem especificados em JAVA para estas entidades. Três destes especificam-se usando palavras reservadas que se designam por *modificadores de acesso* designadamente: `public`, `protected` e `private`. O quarto tipo de acesso é definido por omissão, ou seja, caso nenhum modificador seja declarado, tal como no exemplo da classe `Contador`.

São as seguintes as principais regras de acessibilidade para as classes:

- *Qualquer que seja o modificador usado na definição de uma classe, uma classe é sempre acessível a todas as outras classes que pertençam ao mesmo package;*

- *Quando nenhum modificador é usado na definição da acessibilidade de uma classe, então a classe apenas pode ser acedida dentro do package a que pertence;*

- *Quando uma classe é declarada como* `public`, *a ela terá acesso qualquer código JAVA que tenha acesso ao package a que tal classe pertence;*

- *Quando uma classe é não-pública, então apenas é acessível dentro do seu package. Por esta razão,* `private` *e* `protected` *apenas são usados em classes especiais (ver secção 3.8).*

A classe `Contador` 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 da directoria 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 pretendessemos declarar que a classe `Contador` depois de compilada deverá fazer parte do *package* `java.minhasclasses`, cuja directoria faz parte da CLASSPATH, então escreveríamos no ínicio do ficheiro:

```
package java.minhasclasses
class Contador {
```

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

```
package java.minhasclasses
public class Contador {
```

REGRAS DE ACESSIBILIDADE A VARIÁVEIS E MÉTODOS DE INSTÂNCIA

Os membros de uma dada classe, variáveis e métodos de instância, podem de igual forma especificar a sua acessibilidade usando os modificadores de acesso já antes apresentados.

- *Por razões de preservação do encapsulamento, é boa prática não declarar variáveis de instância nem por omissão de acesso nem como sendo públicas, e criar métodos de acesso específicos que são declarados como* `public` *e disponibilizados como tal na API;*

- *Um método declarado como* `public` *é acessível de qualquer ponto de um programa JAVA;*

- *A API de uma classe JAVA é de facto o conjunto dos métodos de instância que não forem declarados como* `private`*;*

- *Um método sem modificador de acesso é acessível a qualquer classe do mesmo package;*

- *Um método ou variável declarados como* `private` *são apenas acessíveis dentro da própria classe;*

- *Um método ou variável declarados como* `protected` *são acessíveis na própria classe, de outra qualquer classe do mesmo package e ainda nas subclasses da classe (ver noção de subclasse no capítulo 4).*

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

DEFINIÇÃO DA CLASSE Contador COM MODIFICADORES DE ACESSO

```
public class Contador {

   // construtores

   public Contador() {
         conta = 0;
   }

   public Contador(int val) {
         conta = val;
   }

   // variáveis de instância

   private int conta;

   // métodos de instância

   public int getConta() {
         return conta;                  // interrogador - selector
   }

   public void incConta() {
         conta = conta + 1;             // modificador do estado
   }

   public void incConta(int x) {
         conta = conta + x;             // modificador do estado
   }

   public void decConta() {
         conta = conta - 1;             // modificador do estado
   }

   public void decConta(int x) {
         conta = conta -x;              // modificador do estado
   }


   public String toString() {
     return (new String("Contador = " + conta));
   }
}
```

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.

TESTE DA CLASSE Contador

Vamos agora estudar qual a forma usual de em JAVA se desenvolver código para teste de uma classe acabada de desenvolver, neste caso, a classe `Contador`.

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 razões de complexidade, o código da classe de teste não apresentará qualquer interacção com o utilizador mas apenas entre a classe de teste e a classe a testar.

Ou seja, pretendemos desenvolver uma classe de teste da classe `Contador` que nos permita criar instâncias de `Contador`, enviar a estas instâncias mensagens que correspondem à invocação e execução dos métodos da sua API, assim consultando e modificando o estado interno de tais instâncias e podendo observar no monitor os diversos resultados produzidos pelo envio de tais mensagens.

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 à directoria corrente. Sendo, em geral, de acesso público, esta classe tem a seguinte estrutura de definição genérica:

```
public class TesteContador {
// Classe de teste da Classe Contador

// método principal com todo o código de teste

      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(String args[])`.

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 APIs 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 `System.out.print()` ou `println()`.

```
public class TesteContador {

// Classe de teste da Classe Contador
// método único com todo o código de teste

    public static void main(String args[]) {

    // Criação de Instâncias

    Contador ct1, ct2, ct3;
    ct1 = new Contador();
    ct2 = new Contador(20);
    ct3 = new Contador(10);

    // Utilização das Instâncias

    int c1, c2, c3; // variáveis auxiliares

    c1 = ct1.getConta();
    c2 = ct2.getConta();

    // primeira saída de resultados para verificação

    System.out.println("c1 = " + c1);
    System.out.println("c2 = " + c2);

    // alterações às instâncias e novos resultados

    ct1.incConta(); ct2.incConta(12);
    c1 = ct1.getConta(); c2 = ct2.getConta();
    c3 = c1 + c2; System.out.println("c1 + c2 = " + c3);

    ct3.decConta(); ct2.decConta(5);
    c1 = ct2.getConta(); c2 = ct3.getConta();
    c3 = c1 + c2; System.out.println("c1 + c2 = " + c3);

    // conversão para string e apresentação

    System.out.println(ct1.toString());
    System.out.println(ct2.toString());

    }

}
```

MÉTODOS COMPLEMENTARES USUAIS

Vamos agora complementar a definição da classe `Contador` com dois métodos que, não tendo sido pedidos na lista de requisitos, são métodos de implementação quase obrigatória, juntamente com o método `toString`, e que são os métodos `equals` e `clone`.

O método `equals` deverá permitir comparar o objecto parâmetro com o receptor e devolver um valor booleano `true` ou `false` conforme estes sejam ou não iguais em valor. O método `equals`, por razões que serão explicadas no próximo capítulo, deve obedecer à seguinte assinatura e resultado,

```
boolean equals(Object obj)
```

ou seja, receber como parâmetro uma instância da classe mais genérica existente em JAVA, a classe `Object`, verificar se o parâmetro é de facto compatível com a classe do receptor e, caso seja, implementar a igualdade de objectos dessa classe, 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 `Contador`:

```
public boolean equals(Object obj) {
  if ((obj != null) && (obj intanceof Contador))
         return this.getConta() == obj.getConta() ;
  }
  return false;
}
```

Em primeiro lugar, a condição garante que apenas iremos comparar os valores da instância parâmetro com o receptor se duas condições forem garantidas. Primeiro, que o objecto parâmetro não tem o valor `null`, ou seja, se `obj != null`. Em seguida, se o objecto parâmetro for de facto uma instância de `Contador`, teste realizado usando o operador `instanceof` em (`obj intanceof Contador`), e que deve devolver o valor `true`.

Satisfeitas estas duas condições, então poderemos comparar os *valores* das duas instâncias de `Contador`, parâmetro e receptor, comparando neste caso os valores das suas respectivas variáveis de contagem, acedidas pelo método `getConta()`, o que neste exemplo corresponde à comparação de dois inteiros usando o respectivo operador de igualdade ==.

Note-se que, se pretendêssemos escrever o mesmo método para uma qualquer classe de nome X, apenas teríamos que, relativamente ao anterior, modificar a parte correspondente ao algoritmo de comparação, cf. em :

```
public boolean equals(Object obj) {
  if ((obj != null) && (obj intanceof X)) {
        return  comparação específica da classe X ;
  }
  return false;
}
```

O método clone deverá permitir que se crie e devolva uma cópia do objecto que é o receptor da mensagem `clone()`, devendo ser original e cópia garantidamente objectos diferentes, ou seja, devendo-se garantir que, qualquer que seja `x`, então são verificáveis as seguintes propriedades escritas em JAVA:

`x.clone() != x`
`x.clone().getClass() == x.getClass()` dá resultado `true`
`x.clone().equals(x)` dá como resultado `true`

Assim, a clonagem de uma dada instância de uma dada classe deve garantir que o original e a cópia **não são o mesmo objecto, são instâncias da mesma classe** e **têm o mesmo valor**.

Pelas mesmas razões apontadas na definição do método anterior, o método `clone` deve obedecer à seguinte assinatura e resultado:

```
Object clone()
```

Assim, e no caso de pretendermos realizar a clonagem de instâncias de `Contador`, deveríamos definir o código seguinte:

```
public Object clone () {
     return new Contador(this.getConta());
}
```

garantindo assim que, qualquer que fosse o receptor da mensagem `clone`, uma nova instância de `Contador` seria criada contendo o mesmo valor do receptor.

Dado que o método `clone` devolve uma instância de `Object`, na sua utilização há que ter em atenção este facto e, usando `casting`, redefinir a classe do objecto, tal como se exemplifica no código seguinte:

```
Contador ct1, ct2;
ct1 = new Contador(10);
ct2 = (Contador) ct1.clone();    // casting
```

COMPLEMENTO DA DEFINIÇÃO 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**.

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 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 valores, 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 fariam sentido colocar em qualquer outro local.

Por exemplo, imaginemos que no conjunto de requisitos para a definição da classe `ContaBanc` (conta bancária) nos era solicitado o seguinte:

- *Deverá ser possível possuir a cada momento o número total de contas já 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 que o âmbito de uma instância, dado que diz respeito a *todas as instâncias*, então tal variável de instância, por exemplo, `tcontas`, apareceria em todas as instâncias de `ContaBanc` 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 passam a ser 121.

O mesmo raciocínio se aplica à variável que deverá a cada momento conter o total dos saldos das mais diversas contas.

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 `ContaBanc`. 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 foi nunca instanciada.

As variáveis de classe servem também por vezes para conter valores constantes, ou seja imutáveis a partir da sua inicialização, dado não existirem métodos de classe de modificação, 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 usando os mesmos valores de referência. Por exemplo, se definirmos uma classe `Circulo` com um método de instância `area()` e pretendermos que o valor de π 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 π em cada instância, correndo 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 e, portanto, não podem invocar qualquer método de instância.

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 seguinte mostra a estrutura gráfica usada em geral 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.

✏ As mensagens a que uma CLASSE responde implementam-se em métodos neste caso designados por **MÉTODOS de CLASSE**.

✏ Dos MÉTODOS de CLASSE mais importantes incluem-se os que permitem criar INSTÂNCIAS da CLASSE (cf. **new** , **create**, etc.)

✏ No entanto outros podem existir, por exemplo, para aceder às VARIÁVEIS da CLASSE.

✏ Assim, a completa definição de uma CLASSE, em linguagens em que estas existem em "run-time", consiste, fundamentalmente, na definição das seguintes entidades :

- VARIÁVEIS de CLASSE.
- MÉTODOS de CLASSE.
- VARIÁVEIS de INSTÂNCIA.
- MÉTODOS de INSTÂNCIA.

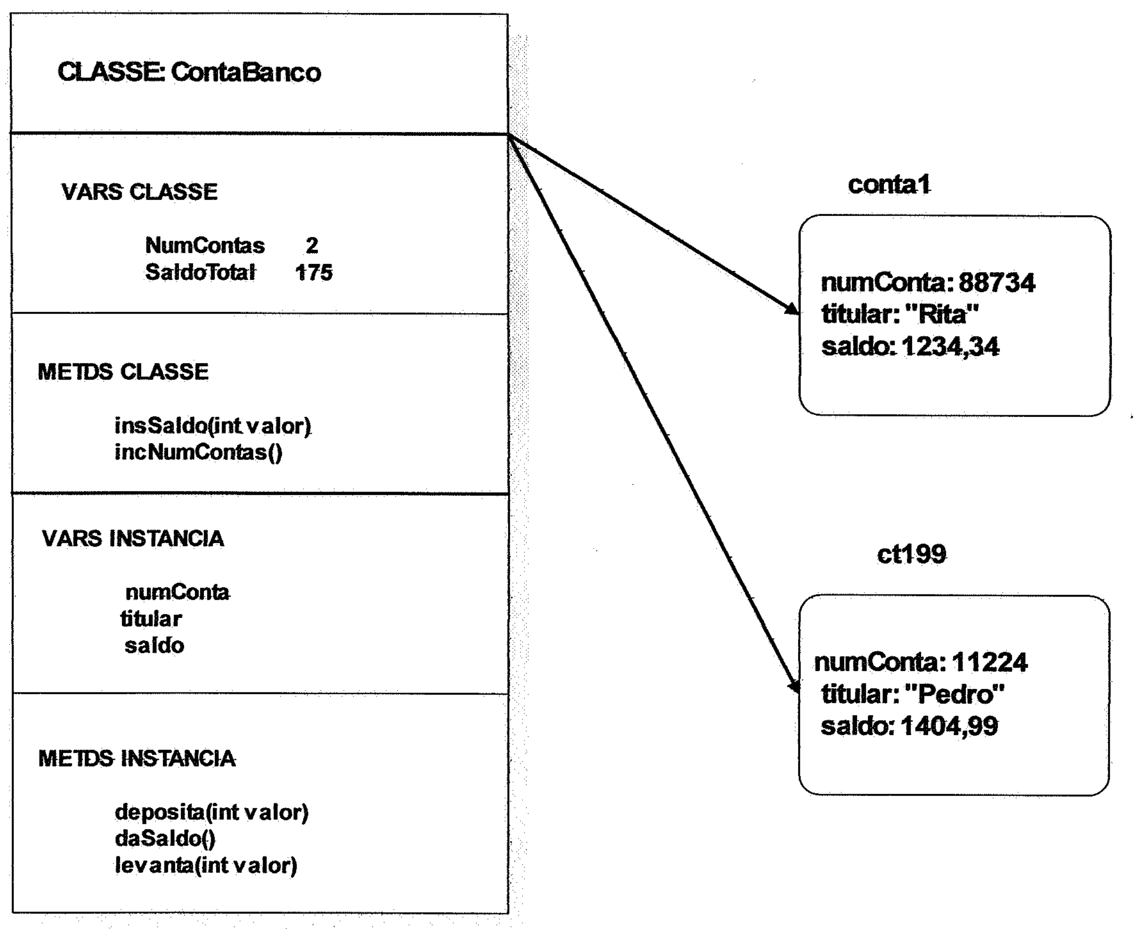
✏ **VARIÁVEIS de CLASSE** irão conter os valores que representam ***informação partilhada por todas as instâncias*** da CLASSE. São portanto similares à variáveis globais no contexto de uma CLASSE e das suas instâncias.

- **MÉTODOS de CLASSE** definem o COMPORTAMENTO da CLASSE.

- VARIÁVEIS e MÉTODOS de INSTÂNCIA definem a ESTRUTURA e o COMPORTAMENTO comum a todas as INSTÂNCIAS da CLASSE.

- Nas linguagens como SMALLTALK em que variáveis são polimórficas (ie. assumem qualquer tipo), a definição de VARIÁVEIS de CLASSE ou de INSTÂNCIA é muito simples: **declaração do seu identificador !**

- Identificadores de CLASSE são em geral iniciados por uma letra MAIÚSCULA. Em Smalltalk tal significa que um identificador de uma CLASSE é uma variável global:



CLASSE E DUAS INSTÂNCIAS DA CLASSE

Apresenta-se em seguida o código Java correspondente à criação de uma classe `ContaBanco` muito semelhante a esta apresentada na figura anterior, e da sua utilização numa classe de teste onde são criadas contas, consultados os respectivos saldos, etc.

De notar que a declaração de métodos e de variáveis de classe se distingue das declarações de variáveis e de métodos de instância apenas pelo facto de serem precedidas pela palavra reservada `static`.

De notar finalmente que frases JAVA em que o receptor de uma mensagem tem um identificador iniciado por letra maiúscula se referem a classes.

X. m(); // mensagem m() enviada à classe X
x.m(); // mensagem m() enviada a uma qq. instância de nome x

```
// CLASSE QUE DEFINE CONTAS BANCÁRIAS SIMPLES
// E QUE USA VARIÁVEIS DE CLASSE PARA GUARDAR DADOS COMUNS
// A TODAS AS SUAS INSTÂNCIAS


public class ContaBanco {

   // variaveis de Classe

   static int numContas = 0;

   // metodos de Classe

   static void novaConta() {
       numContas++;
   }

   static int daNovoNumConta() {
       return numContas;
   }

   static int totalContas() {
       return numContas;
   }

   // Construtor

   public ContaBanco(String nome, int saldoInic) {
        this.novaConta();
        titular = nome;
        numConta = this.daNovoNumConta();
        saldo = 0;
        this.deposito(saldoInic);
        totalMovimentos = 0;
   }

   // variaveis de instancia

   private String titular;
   private int numConta;
   private int saldo;
   private int totalMovimentos;

   // metodos de instancia

   public int daSaldo() { return saldo; }
   public int movimentos() { return totalMovimentos; }
   public String titular() { return titular; }
   public int numConta() { return numConta; }

   public void deposito(int valor) {
                  saldo = saldo + valor;
                  totalMovimentos++;
```

```
  public boolean temSaldo(int valor) { return saldo >= valor; }

  public void levantamento(int valor) {
                 saldo = saldo - valor;
                 totalMovimentos++;
  }

  public String toString() {
   return (new String("Conta:" + numConta + "/Saldo =" + saldo));
  }
}
```

```
import java.lang.*;

public class TstContaBanco {

// teste

public static void main(String args[]) {

   String nome = new String("Abel");
   int saldo = 10000;

   ContaBanco ct1 = new ContaBanco(nome, saldo);

   nome = new String("Luisa");
   saldo = 20000;

   ContaBanco ct2 = new ContaBanco(nome, saldo);

   System.out.println(ct1.toString());
   System.out.println(ct2.toString());

   System.out.println("Total de Contas = " + ContaBanco.totalContas());


   System.out.println(ct1.getClass());
   System.out.println(ct1 instanceof ContaBanco);


    if (ct1.temSaldo(500)) ct1.levantamento(500);
    ct1.deposito(200);
    ct2.deposito(10000);
    if (ct2.temSaldo(12000)) ct2.levantamento(12000);

   System.out.println(ct1.toString());
   System.out.println(ct2.toString());

   System.out.println(ct1.titular());
   System.out.println(ct2.titular());

   System.out.println(ct1.movimentos());
   System.out.println(ct2.movimentos());


  }
  }
```

```
C:\jdk1.1>java TstContaBanco
Conta:1/Saldo =10000
Conta:2/Saldo =20000
Total de Contas = 2
class ContaBanco
true
Conta:1/Saldo =9700
Conta:2/Saldo =18000
Abel
Luisa
2
2
```

3.4 DEFINIÇÃO DE CLASSES USANDO COMPOSIÇÃO

Um dos exemplos típicos da PPO é a criação de uma classe que represente contas bancárias. Uma conta bancária é, simultaneamente, uma entidade bem conhecida e interessante. Vamos, então, criar uma classe que defina contas bancárias, a classe `ContaBanc`, sendo as seguintes as características de estrutrura e comportamento de tais contas:

- *Qualquer conta bancária tem um identificador único;*
- *Uma conta bancária pode ter 1 ou mais titulares, sendo um deles o titular principal, do qual se conhece a morada;*
- *Cada conta possui um saldo actual e um "plafond" de crédito que pode variar de conta para conta, mas que é definido quando esta é criada;*
- *A qualquer momento é possível realizar um depósito;*
- *Um levantamento apenas pode ser realizado se a importância pedida não ultrapassar o "plafond" de crédito definido;*
- *A qualquer momento deverá ser possível saber o saldo da conta;*
- *Deverá ser possível eliminar titulares e acrescentar titulares novos;*
- *Deverá registar-se o número total de movimentos activos realizados sobre a conta, ou seja, depósitos e levantamentos;*

Assim, a estrutura interna de uma conta bancária deverá possuir, pelo menos, os seguintes campos:

ContaBanc
numConta **morada** **titulares** **saldo** **numMov** **plafond**

Vamos agora definir o tipo de cada uma das variáveis de instância. O número de conta deve ser um identificador único, sendo constituído por uma sequência de mais ou menos dígitos. Com este número não iremos realizar quaisquer operações

aritméticas, pelo que será representado como uma instância de `String`. A morada do titular principal será igualmente representada como uma `String`. O saldo, o número total de movimentos e o *plafond* serão definidos como `int`. Finalmente, o conjunto de titulares, de cada um dos quais se pretende guardar o nome, deverá ser guardado numa estrutura linear ilimitada. Tal exclui a possibilidade de usarmos um *array* contendo as *strings* que representam os nomes dos titulares, dado que os *arrays* possuem dimensões fixas e não podem crescer de modo dinâmico.

Este problema levar-nos-ia, de imediato, a procurar uma classe JAVA predefinida que pudesse ser reutilizada e satisfizesse os requisitos. Daí, como se referiu antes, a grande importância de ter um profundo conhecimento das classes preexistentes em JAVA.

Poupando, de momento, tal esforço, vamos apresentar uma classe JAVA que nos oferece a funcionalidade desejada, e que é a classe **`Vector`**. A classe `Vector` de JAVA implementa uma abstracção de dados que é uma estrutura linear indexada a partir do índice 0, estrutura e indexação equivalentes às de um *array*, no entanto sem qualquer limite de dimensão. Para além disso, ao contrário do que sucede com a manipulação dos *arrays*, neste caso estão já implementados vários métodos:

- `Vector(int capInicial);` *// construtor com dimensão inicial*
- `Vector();` *// construtor; dimensão inicial 0*
- `void addElement(Object obj);` *// junta obj no fim do vector*
- `void insertElementAt(Object obj, int index);`
- `Object clone();` *// devolve uma cópia do vector*
- `boolean contains(Object obj);` *// verifica se pertence*
- `Object firstElement();` *// dá o 1º elemento do vector*
- `Object lastElement();` *// devolve o último elemento*
- `Object elementAt(int index);` *// dá o elemento em tal índice*
- `int indexOf(Object elem);` *// índice da 1ª ocorrência de elem*
- `boolean isEmpty();` *// testa se o vector está vazio*
- `boolean remove(int index);` *// remove elemento em index*
- `Object[] toArray();` *// converte em array de Object*
- `int size();` *// determina a dimensão actual*

Estes são apenas alguns dos métodos disponibilizados pela classe `Vector`. Como vemos agora, se os titulares da conta forem associados a um *vector de strings*, a sua manipulação consistirá no envio a tal instância de `Vector` das mensagens que são adequadas à activação dos métodos que implementam a funcionalidade que se pretende, e ao tratamento dos respectivos resultados. Não temos, pois, necessidade de desenvolver código, mas apenas de utilizar correctamente o já existente.

Neste caso, teremos apenas que especificar, desde já, que o titular principal da conta, que deve sempre existir, será sempre o primeiro elemento do vector, ou seja, ocupará sempre o índice 0.

Assim, representaremos os *titulares da conta* usando um `Vector` que irá conter as *strings* que representam, por sua vez, os seus respectivos nomes. Para tal, apenas teremos que declarar:

```
Vector titulares;
```

dado que sendo a classe `Vector` genérica, ou seja, tendo sido desenvolvida para aceitar como elemento qualquer `Object`, poderá ter nos seus elementos instâncias de qualquer classe. No nosso caso, apenas iremos inserir no vector instâncias de `String`. Porém, tal não necessita de ser especificado na declaração.

Teremos, assim, a seguinte estrutura para a classe `ContaBanc`:

ContaBanc
String numConta **String morada** **Vector titulares** **int saldo** **int numMov** **int plafond**

A classe `ContaBanc` é composta por instâncias das classes `String` e `Vector`, o que em muito vai facilitar a sua construção final, ou seja o desenvolvimento do seu código, dado que que a maior parte deste se irá basear na invocação de

métodos já definidos e tornados públicos nas APIs destas classes predefinidas de JAVA.

Vamos, em seguida definir a sintaxe dos métodos que vão ser tornados públicos pela classe `ContaBanc`. Em primeiro lugar, deveremos programar um conjunto de métodos de acesso às variáveis de instância de `ContaBanc`, métodos que terão as assinaturas seguintes:

```
String getNumConta()
String getTitular()
int getSaldo()
int getNumMov()
int getPlafond()
Object[] getTitulares()
```

Quanto aos métodos que são modificadores do estado interno de cada instância de `ContaBanc`, teremos, de momento, os seguintes:

```
void deposita(int valor)
void levanta(int valor)
void insereTit(String titular)
void alteraMorada(String mora)
void alteraPlafond(int nplaf)
```

Comecemos por criar os construtores de `ContaBanc`. Não sendo, naturalmente, aceitável criar uma conta bancária sem pelo menos o número de conta, o titular principal, a sua morada, um saldo inicial e um *plafond* de crédito, estes deverão ser os parâmetros de entrada obrigatórios a tal construtor, que seria então:

```
public ContaBanc(String titp, String mora,
                                int sld, int plf) {
      titulares = new Vector(5);
      titulares.insertElementAt(titp, 0);
      morada = mora;
      if (sld >=0)     // valida saldo inicial
            saldo = sld;
      else
            saldo = 0;
      plafond = (plf >=0 ? plf : 0); // valida plafond
      numMov = 0;      // inicializa nº de movimentos
}
```

O código de cada um dos métodos interrogadores do estado é bastante simples:

```
public String getNumConta() {
      return numConta;
}

public String getTitular() {
      return titulares.firstElement();
}
```

ou ainda:

```
public String getTitular() {
      return titulares.elementAt(0);
}

public int getSaldo() {
      return saldo;
}

public int getNumMov();
      return numMov;
}

public int getPlafond() {
      return plafond;
}

public Object[] getTitulares() {
      return titulares.toArray();
}
```

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

```
public void deposita(int valor) {
      saldo = saldo + valor;
      numMov = numMov + 1;
}
```

```
public void insereTit(String titular) {
      titulares.addElement(titular);
}

public void alteraMorada(String mora) {
      morada = mora;
}

public void alteraPlafond(int nplaf) {
      plafond = nplaf;
}
```

MÉTODOS PARCIAIS E PRÉ-CONDIÇÕES

Deixámos propositadamente para o fim o método `levanta`, 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 levantamento 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 a conta, não deverá nunca ser possível realizar um levantamento de uma importância maior que a soma do saldo com o seu *plafond*, e a soma destes dois nunca será inferior a zero. Assim, apenas deveremos invocar a operação de levantamento numa situação em que a operação pode 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 identificar que a operação não pôde ser executada,

No nosso exemplo, uma má codificação deste método poderia conduzir ao código seguinte, que é bastante típico:

```
public boolean levanta(int valor) {
      if (plafond + 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 deve devolver um booleano. 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 booleano, teremos que invocá-lo não usando a que seria a expressão normal caso o resultado fosse `void`:

```
conta.levanta(10000);
```

mas antes uma série de expressões, tais como:

```
boolean res = conta.levanta(10000);
if (!res)
  System.out.println("Levantam. impossível !");
```

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 `levanta`. 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, o serviço que lhe foi solicitado.

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 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 público e que devolve como resultado um `boolean` após realizar o teste ao estado interno do objecto.

```
public boolean preLevanta(int valor) {
     return (saldo + plafond) >= 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`).

```
public void levanta(int valor) {
     saldo = saldo - valor;
     numMov = numMov + 1;
}
```

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 (conta.preLevanta(val)) {
  conta.levanta(val); // realiza a operação
else
   System.out.println("Levantam. impossível !");
}
```

Se à pré-condição tivéssemos dado o nome sugestivo de `podeLevantar(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 tratamento de Excepções de JAVA, que será realizado no capítulo 6, 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 `ContaBanc`.

```
public class ContaBanc {

// construtor

public ContaBanc(String numct, String titp,
                 String mora, int sld, int plf) {
      numConta = numct;
      morada = mora;
      saldo = sld >=0 ? sld : 0;
      plafond = plf >=0 ? plf : 0;
      numMov = 0;
      titulares = new Vector(5);
      this.insereTit(titp);
}

// variáveis de instância

private String numConta;
private String morada;
private Vector titulares;
private int saldo;
private int plafond;
private int numMov;

// métodos de instância

public String getNumConta() {
      return numConta;
}

public String getTitular() { // casting !!
      return titulares.firstElement();
}
```

```java
    public int getSaldo() {
        return saldo;
    }

    public int getNumMov() {
        return numMov;
    }

    public int getPlafond() {
        return plafond;
    }

    public Object[] getTitulares() {
        return titulares.toArray();
    }

    public boolean preLevanta(int valor) {// pré-cond
        return (saldo + plafond) >= valor ;
    }

    void levanta(int valor) {         // ver pré-cond
        saldo = saldo - valor;
        numMov = numMov +1;
    }

    public void deposita(int valor) {
        saldo = saldo + valor;
        numMov = numMov + 1;
    }

    public void insereTit(String titular) {
        titulares.addElement(titular);
    }

    public void alteraMorada(String mora) {
        morada = mora;
    }

    public void alteraPlafond(int nplaf) {
        plafond = nplaf;
    }

}
```

3.5 Complemento da Definição 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**.

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 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 valores, 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 fariam sentido colocar em qualquer outro local.

Por exemplo, imaginemos que no conjunto de requisitos para a definição da classe `ContaBanc` nos era solicitado o seguinte:

- *Deverá ser possível possuir a cada momento o número total de contas já 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 que o âmbito de uma instância, dado que diz respeito a *todas as instâncias*, então tal variável de instância, por exemplo, `tcontas`, apareceria em todas as instâncias de `ContaBanc` 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 passam a ser 121.

O mesmo raciocínio se aplica à variável que deverá a cada momento conter o total dos saldos das mais diversas contas.

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 `ContaBanc`. 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 foi nunca instanciada.

As variáveis de classe servem também por vezes para conter valores constantes, ou seja imutáveis a partir da sua inicialização, dado não existirem métodos de classe de modificação, 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 usando os mesmos valores de referência. Por exemplo, se definirmos uma classe `Circulo` com um método de instância `area()` e pretendermos que o valor de π 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 π em cada instância, correndo 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 e, portanto, não podem invocar qualquer método de instância.

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 seguinte mostra a estrutura gráfica usada em geral 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.

ContaBanc
CVARS
CMETDS
CONSTRUCT
IVARS
IMETDS

Fig. 3. 3 – Estrutura completa de definição de uma Classe

Vamos, em seguida, apresentar a forma de em JAVA declararmos as variáveis e os métodos de classe, voltando ao exemplo concreto da classe `ContaBanc`.

Assim, imediatamente a seguir ao cabeçalho da declaração de classe, devemos declarar as variáveis e métodos da classe. 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`.

```
public class ContaBanc {

// variáveis de classe

public static int numContas = 0; // c/ inicialização
public static int saldoTotal = 0;// c/ inicialização


// métodos de classe

public static int getNumContas() {
      return numContas;
}

public static int getSaldoTotal() {
      return saldoTotal;
}

public static void incNumContas() {
      numContas++;
}

public static void actSaldoTotal(int saldo) {
      saldoTotal += saldo;
}
           ...............................................
```

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 tais como:

```
int total = ContaBanc.getNumContas();
int saldot = ContaBanc.getSaldoTotal();
```

sendo claro, em função do identificador do receptor das mensagens, `ContaBanc`, que é um nome de classe, que tais mensagens dizem respeito a métodos de classe e não a métodos de instância.

Os outros dois métodos são modificadores do estado das variáveis de classe, sendo pois usados em expressões da forma:

```
ContaBanc.incNumContas();
ContaBanc.actSaldoTotal(novoSaldo);
```

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 "pseudo-variável", deste modo dando a indicação de que se trata de uma constante e, em geral, por uma questão de estilo, usando apenas letras maiúsculas no seu identificador, tal como em:

```
public static final double PI = 3.1415926535897932;
public static final int MAX_INT = 555;
```

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 constantes realiza a sua substituição imediata pelos respectivos valores, há neste caso toda a vantagem em explicitamente se designar tais constantes. Assim, e ao contrário do que dissemos relativamente a variáveis de classe e de instância, constantes de classe podem ser usadas directamente sob a forma de expressões, tais como:

```
double area = Circulo.PI * raio * raio;
int x = ClasseX.MAX_INT * 10;
```

A palavra reservada `static` indica que existe apenas uma cópia de tal variável que é associada à classe, ao contrário das variáveis de instância que possuem tantas cópias quantas as instâncias efectivamente criadas.

Procurando-se apenas completar a nova definição da classe `ContaBanc`, que passou a ter variáveis de classe na sua definição, vejamos como deveria ser definido o novo construtor, dado que deve agora actualizar tais variáveis de classe invocando os métodos de classe para tal definidos.

```
public ContaBanc(String numct, String titp,
                 String mora, int sld, int plf) {
  this.incNumContas();     // incrementa nº de contas
  numConta = numct;
  morada = mora;
  if (sld >=0)  saldo = sld; else saldo = 0;
  this.actSaldoTotal(saldo); // actualiza saldo total
     plafond = (plf >=0 ? plf : 0);
     numMov = 0;
     titulares = new Vector(5);
     this.insereTit(titp);
}
```

Note-se que, tal como se mostra no exemplo, a referência `this` no contexto de um construtor tanto pode referenciar a classe propriamente dita como a instância de tal classe que está a ser criada, pelo que, deste modo, quer os métodos de classe quer os de instância podem ser invocados no código dos seus próprios construtores.

3.6 PROJECTO: MÁQUINA AUTOMÁTICA DE VENDAS

Procurando aplicar o conjunto de conhecimentos adquiridos neste capítulo sobre as classes de JAVA, vamos desenvolver uma classe que implemente a estrutura e a funcionalidade de uma Máquina Automática de Vendas com os seguintes requisitos de projecto:

Pretende-se construir uma classe cuja estrutura e comportamento programado simule uma Máquina Genérica de Venda de produtos de dado tipo (cf. chocolates, bebidas, tabaco, jornais, etc).

*A Classe **MaqVenda** deverá definir a seguinte informação fundamental,*

- *tipo de produto genérico que vende (chocolates, bebidas, tabaco, etc.);*
- *tabela de triplos produto-preço-quantidade existentes na máquina;*
- *total em dinheiro existente na máquina;*
- *número total de compras realizadas;*
- *estado da máquina (avariada, sem dinheiro, sem produtos, ok);*

bem como o seguinte comportamento,

- *criar uma nova máquina de vendas de dado produto;*
- *determinar que tipo de produto vende a máquina;*
- *determinar para uma dada máquina o preço de dado produto;*
- *determinar o estado de uma máquina;*
- *determinar o número total de máquinas criadas;*
- *realizar uma compra válida de dado produto;*
- *alterar o estado de uma dada máquina;*
- *carregar uma máquina com dinheiro ou com dado produto.*

Analisando os requisitos apresentados, torna-se claro que a classe a construir, que vamos designar por `MaqVenda`, necessita de ter duas variáveis de classe que irão conter quer o número total de máquinas criadas até um dado momento, quer uma constante que indique o número máximo de produtos que uma máquina pode ter.

Teremos então duas variáveis de classe declaradas como:

```
static int numMaquinas; // total de máquinas criadas
static final int MAXPRODS = 100; // máximo de produtos
```

Do ponto de vista da estrutura interna de uma máquina de vendas, os requisitos apresentados mostram que é necessário possuir uma estrutura capaz de guardar as informações fundamentais sobre cada produto, designadamente, o seu nome, o seu preço e a quantidade deste existente na máquina. Assim, cada produto vendável vai ser caracterizado por um triplo contendo as suas informações relevantes. Ainda que fosse aceitável representar a informação referente a cada produto como sendo um *array* ou um `Vector` com três campos, decidiu-se criar uma classe própria, cujas instâncias são triplos que irão fazer parte da informação de cada máquina.

A definição da classe `TriploMaq` será então a seguinte:

```
// CLASSE QUE IMPLEMENTA OS TRIPLOS REPRESENTATIVOS DA
// INFORMAÇÃO DE UM PRODUTO DE UMA MÁQUINA DE VENDA

    public class TriploMaq {

        // construtores

        public TriploMaq() {
            produto = new String("");
            preco = 0; quant = 0;
        }

        public TriploMaq(String prod, int pr, int qt) {
            produto = prod;
            preco = pr; quant = qt;
        }

        // variáveis de instância

        private String produto;
        private int preco;
        private int quant;

        // métodos de instância

        public String prod() {
            return produto;
        }

        public int preco() {
            return preco;
        }

        public int quant() {
            return quant;
        }

        public Object clone() {
         return new TriploMaq(this.produto, this.quant,
                              this.preco);
        }
```

```
        public String toString() {
         return new String("Triplo = " + produto + ","
                      +preco + ", " + quant);
        }
    }
```

Admitindo que estes triplos que contêm a informação dos produtos de uma dada máquina vão ser armazenados num *array*, então a estrutura interna das instâncias da classe `MaqVenda` será a seguinte:

```
private String tipoprod; // tipo de produto da maquina
private int totalDinheiro;// saldo actual
private int numCompras;   // total de compras
private String estado;    // flexivel => validacoes !
private TriploMaq[ ] tab; // array de inf. de produtos
private int numprods;     // nº de produtos a escolher
```

Apresenta-se a seguir a definição completa da classe `MaqVenda`, com comentários suficientes para que seja fácil compreender a funcionalidade associada a cada um dos métodos de instância, programados conforme os requisitos do projecto.

```
// CLASSE QUE DEFINE A ESTRUTURA E COMPORTAMENTO DE UMA
// MÁQUINA AUTOMÁTICA DE VENDA DE PRODUTOS DE DADO TIPO

  public class MaqVenda {

      // variáveis de classe

      static int numMaquinas; // total de máquinas criadas
      static final int MAXPRODS = 100; // máximo de produtos

      // métodos de classe

      // incrementa o número de máquinas criadas

      public static void incNumMaquinas() {
            numMaquinas += 1 ;
      }

      // acede à constante de número máximo de produtos

      public static int maxProds() {
            return MAXPRODS;
      }

      // consulta o número de máquinas criadas

      public static int qtMaquinas() {
            return numMaquinas;
      }

      // variáveis de instância

      private String tipoprod; // tipo de produto da maquina
      private int totalDinheiro;
      private int numCompras; // total de compras efectuadas
      private String estado;  // flexivel => validacoes !
      private TriploMaq[ ] tab; // array de triplos
      private int numprods;     // nº de produtos a escolher
```

```
// construtores

public MaqVenda(String tprod, TriploMaq tabinic[],
                            int cash, int nprods) {
      this.incNumMaquinas();
      tipoprod = tprod;
      totalDinheiro = cash;
      tab = tabinic;
      numprods = (nprods <= this.maxProds() ? nprods :
                            this.maxProds());
      numCompras = 0;
      estado = new String("OK"); // atencao ao valor
                                     inicial !
}

// métodos de instância privados (ie. auxiliares)

// devolve o triplo na posição dada por index

private TriploMaq daTriplo(int index) {
  TriploMaq triplo = (TriploMaq) tab[index].clone();
  return triplo;
}

// procura no array de triplos o índice do produto
// de nome dado; devolve 0 se tal nome não existir.

private int procuraProduto(String prod) {
  boolean enc = false;
  int ind = -1;
  while (! enc && ind < numprods) {
    ind++;
    enc = ((this.daTriplo(ind).prod()).equals(prod) ?
                    true : false) ;
  };
  return (enc ? ind : 0);
}

// métodos de instância

public String queProduto() {
      return tipoprod;
}
```

```java
    public int cash() {
        return totalDinheiro;
    }

    public int ncompras() {
        return numCompras;
    }
    public int numProds() {
        return numprods ;
    }

// devolve um array que é uma cópia da tabela de
   triplos

    public TriploMaq[] daTab() {
      TriploMaq t[] = new TriploMaq[MaqVenda.maxProds()];
      System.arraycopy(tab,0, t, 0, numprods);
      return t;
    }

    // determina o preço de um dado produto;
    // devolve 0 se tal produto não existir.

    public int precoDe(String prod) {
      int ind = this.procuraProduto(prod);
      return (ind != 0 ? this.daTriplo(ind).preco() : 0);
    }

    // devolve o estado actual da máquina

    public String daEstado() {
        return estado;
    }

    // realiza uma compra previamente validada (pré-cond)

    public void compra(String prod, int quantia) {
     int ind = this.procuraProduto(prod);
     totalDinheiro += tab[ind].preco();
     numCompras += 1;
     tab[ind] = new TriploMaq(prod, daTriplo(ind).preco(),
                        daTriplo(ind).quant() - 1);
    }
```

```
// pré-cond de compra = produto existe, quantidade > 0
// e máquina OK

public boolean preCompra(String prod) {
      int i = this.procuraProdutó(prod);
      return ( i != 0 && tab[i].quant() > 0 &&
                        this.daEstado().equals("OK") );
}

// altera o estado actual da máquina (ou fica igual!)

public void mudaEstado(String nvest) {
      // não testa valores correctos; apenas atribui !
      // admite-se um teste antes do envio da mensagem
      // usando estadoOk() (= teste de pre-condição)
      estado = nvest;
}

// pré-condição para poder alterar o estado da máquina

public boolean estadoOk(String est) {
      return (est.equals("OK") || est.equals("SP") ||
              est.equals("SD") || est.equals("AV"));
}

// carrega a maquina com mais dinheiro

public void maisCash(int dinheiro) {
      totalDinheiro = totalDinheiro + dinheiro;
}

// insere um novo produto na máquina - 2 métodos

public void insNovoProd(String prod, int preco,
                        int qt)  {
// a pre-condicao foi testada, ie. o produto é novo e
// a máquina tem menos de 20 produtos diferentes.
tab[this.numProds()] = new TriploMaq(prod, preco, qt);
this.numprods += 1;
}
```

```
    // métodos de teste de pré-condições de inserção

    public boolean preInsNovoProd(String prod, int preco,
                                           int quant) {
     return (this.numProds() < 20 &&
              this.procuraProduto(prod) == 0 && preco > 0
              && quant > 0 ) ;
    }


    public boolean preInsNovoProd(TriploMaq t) {
      return (this.numProds() < 20 &&
               this.procuraProduto(t.prod()) == 0 &&
               t.preco() > 0 && t.quant() > 0 ) ;
    }

    public void insNovoProd(TriploMaq triplo) {
    // pre-condicao testada
      tab[this.numProds()] = triplo;
      numprods++;
    }

}
```

A classe `MaqVenda` que acaba de ser desenvolvida, apresenta já uma estrutura e um comportamento que lhe conferem alguma complexidade. De facto, a classe foi definida, em termos da sua estrutura interna, usando composição, em particular por inclusão de variáveis de instância da classe `String` e por um *array* de instâncias da classe `TriploMaq`, classe esta que foi desenvolvida especificamente para ser uma classe auxiliar à definição da classe `MaqVenda`.

No entanto, e apesar de satisfazer todos os requisitos, a concepção desta classe `MaqVenda` não está completamente correcta, dado que, em termos de reutilização de classes predefinidas, algumas decisões de projecto que foram tomadas podem considerar-se pouco eficazes. Por exemplo, considerou-se que a melhor forma de representar os triplos de informação sobre os produtos contidos numa máquina seria através da utilização de um *array* contendo instâncias da classe `TriploMaq`. Esta decisão, à luz das classes predefinidas de JAVA, foi pouco eficaz já que, como podemos ver pelo código apresentado, tivemos que desenvolver o código de

métodos de procura e de inserção no *array*, dado que os *arrays* de JAVA, só por si, pouca funcionalidade oferecem.

Se tivéssemos optado por uma instância de `Vector` para armazenar as instâncias de `TriploMaq`, como vimos já anteriormente, métodos de inserção, consulta e modificação estariam de imediato à nossa disposição, pelo que apenas teríamos que os invocar de forma correcta, conforme a sua API, e não ter que desenvolver código, tal como tivémos que fazer por termos decidido utilizar um *array*.

Apenas o efectivo conhecimento da funcionalidade disponibilizada noutras classes predefinidas, em particular dos *packages* `java.util` e `java.lang`, tais como, para este projecto, as classes `Hashtable` ou `HashMap`, poderia permitir a quem tem que desenvolver a nova classe, um máximo de eficiência, por reutilização, na definição desta.

Assim, e em conclusão, pretende-se com estas observações chamar a atenção para o facto de que, para além do conhecimento necessário de um conjunto de regras e métodos que são muito importantes para a construção, usando linguagens de PPO, de aplicações modulares, reutilizáveis e escaláveis, um bom conhecimento de tudo o que a tecnologia subjacente oferece, seja C++, JAVA, ou outra, é fundamental a uma eficaz utilização da mesma.

Como é que tal eficácia pode ser conseguida? Claramente estudando tudo o que já existe disponível para reutilização, e reutilizando segundo o princípio do "menor esforço", ou seja, seguindo as grandes regras de programação em PPO que neste livro se advogam para a construção de aplicações possuindo propriedades que são inequivocamente consideradas cruciais, reutilizando ao máximo o que o ambiente oferece e procurando programar o mínimo.

3.7 CLASSES NÃO INSTANCIÁVEIS

A criação de classes é, em PPO, como vimos, o processo usual que permite que a partir de tal classe possam ser criadas tantas instâncias idênticas em estrutura e comportamento quantas as necessárias. Este é, portanto, o desiderato fundamental da criação e da utilização de classes.

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 dispôr 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`, invocáveis, portanto, a partir de qualquer outra classe. 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 ao identificador da classe.

Apresentam-se, em seguida, alguns exemplos de utilização desta classe `Math` que não pode criar instâncias (que sentido teria criar as instâncias `mat1` e `mat2` ?) mas que presta serviços de grande valor.

```
double x = Math.sqrt(y);
double x = Math.cos(y);
long i   = Math.round(y);
double x = Math.max(y, z);
```

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. É, no entanto, 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 de tal classe, 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.

3.8 CLASSES: SÍNTESE

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

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 e, portanto, 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.

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()`.

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 `package`, `protected`, `public` e `private`. 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.

Para além das classes de alto nível que, ao serem criadas, vão passar a fazer parte da hierarquia de classes do ambiente JAVA, outras classes podem ser definidas, as chamadas *inner classes*, que, apesar das diversas restrições que apresentam, são auxiliares por vezes importantes para uma mais fácil definição das classes que as vão conter. Serão estudadas posteriormente dada a sua relativa complexidade.

No capítulo 4 iremos estudar uma outra forma de relacionamento entre classes e um outro mecanismo particular de reutilização, que irão trazer ainda mais poder expressivo, e de desenvolvimento, ao paradigma da PPO.

3.9 EXERCÍCIOS PROPOSTOS

Para todos os exercícios propostos desenvolva em JAVA as classes especificadas, realize a compilação do código e, em seguida, crie e compile os programas de teste das mesmas.

1.- Defina e desenvolva uma classe `PortaMoedasMBanco`, *que apresente uma funcionalidade semelhante ao conhecido cartão. Cada instância desta classe ao ser criada deve possuir um titular e um saldo positivo. A partir da sua criação, as instâncias de tal classe deverão ser capazes de responder adequadamente a um conjunto de mensagens que correspondem às operações que podem ser realizadas com o cartão, designadamente:*

- *Realização de um pagamento, caso seja possível;*
- *Apresentação do saldo actual;*
- *Realização de um carregamento do cartão com dada importância;*
- *Indicação do número total de pagamentos realizados;*
- *Apresentação, como string, de todos os dados do cartão.*

Que alterações deveria introduzir na classe `PortaMoedasMBanco` *caso fosse também necessário implementar uma operação que determinasse o valor médio de todos os pagamentos realizados?*

2.- Desenvolva uma classe `FilaDeEspera`, *tendo por base um array que irá conter os nomes das pessoas em tal fila de espera, definindo uma dimensão máxima para a fila. Cada instância desta classe deverá ser capaz de responder às seguintes mensagens:*

- *Qual o actual comprimento da fila?*
- *Qual o comprimento máximo da fila?*
- *Quem é o primeiro da fila?*
- *Remover o primeiro da fila;*
- *Inserir um novo nome na cauda da fila;*
- *Número de vagas na fila actual?*

3.- Desenvolva uma nova classe `FilaDeEspera` *tendo agora por base um Vector de nomes das pessoas em tal fila de espera, ou seja, reutilizando os métodos que JAVA predefine para a classe Vector.*

4.- Defina uma classe `ListadeInteiros`*, cujas instâncias são capazes de guardar sequências de números inteiros. Defina igualmente métodos que sejam capazes de realizar as seguintes operações sobre tais listas:*

- *Determinar a posição do número X, caso exista;*
- *Determinar quantas vezes ocorre o número X na lista;*
- *Dado um número X, determinar quantos elementos da lista são menores;*
- *Devolver uma lista igual à do receptor mas ordenada;*
- *Inserir um novo elemento na lista;*
- *Remover todas as ocorrências do número X;*
- *Determinar o tamanho actual da lista;*
- *Implementar* `equals()` *e* `toString()`*.*

5.- Defina uma classe `CatalogoWeb` *que implemente um arquivo sobre websites, no qual a cada identificador de website se associa uma lista de temas que em tal website são tratados. Implemente ainda a seguinte funcionalidade:*

- *Determinação do número total de websites do catálogo;*
- *Quais os temas que se podem encontrar no website W?*
- *O tema T faz parte dos temas do website W?*
- *Quantos temas podem ser encontrados no website W?*
- *Lista de websites que tratam o tema T;*
- *Acrescentar um novo website ao catálogo e respectivos temas;*
- *Remover o website W dado como parâmetro;*

6.- Estude as classes de JAVA `StringBuffer` *e* `StringTokenizer`*. Defina em seguida uma classe de nome* `Texto`*, tendo uma variável de instância* `texto`*, que é uma instância da classe StringBuffer, e que irá conter um dado texto. Codifique os métodos que irão corresponder às mensagens a ser enviadas a cada instância da classe* `Texto` *para que se obtenham os seguintes resultados:*

* *O número total de palavras contidas no texto;*
* *Um Vector com todas as palavras contidas no texto;*
* *Um array com todas as palavras do texto mas sem repetições;*
* *O número total de linhas do texto;*
* *Juntar o texto dado como parâmetro;*
* *Substituir todas as ocorrências da palavra P pela palavra P1.*

7.- Um Bag é um conjunto que admite duplicados. É, em geral, usado para contar o número de ocorrências de valores de um dado domínio, por exemplo, o número de ocorrências das letras do alfabeto num dado texto, o número de ocorrências de votos no partido X numa eleição, etc. Crie uma representação de "bags" com base em classes predefinidas de JAVA. Sugestão: estude e use as classes Hashtable ou HashMap. Implemente, usando a representação escolhida, os usuais métodos da classe `Bag`*, designadamente:*

* *insere (junta a 1ª, ou mais uma, ocorrência de um dado valor ao Bag);*
* *elementos (devolve um array ou um Vector contendo os elementos do Bag, sem repetições);*
* *ocorrencias (determina o número de vezes que um dado valor ocorre no Bag, devolvendo 0 caso tal valor não exista);*
* *soma (calcula o somatório de todos os valores registados no receptor);*
* *toString() (usando uma StringBuffer).*

8.- Até à versão 1.2, JAVA não possuia uma classe Set com o comportamento de um conjunto matemático, ou seja, com as usuais operações da teoria de conjuntos. Apresente a definição e implementação de uma classe `Set`*, incluindo métodos de instância que implementem as mais importantes operações sobre conjuntos, nomeadamente:*

* *reuniao (faz a reunião matemática do receptor com o argumento);*
* *diferença (dá o conjunto dos elementos do receptor que não pertencem ao conjunto parâmetro)*
* *intersecção (intersecção do receptor com o argumento);*
* *membro (testa se um elemento pertence ao conjunto receptor);*

* *equals (testa se dois conjuntos são iguais);*
* *insere (adiciona um novo elemento ao conjunto, se possível).*

9.- Pretende-se implementar em JAVA uma classe `Polinómio` *cujas instâncias sejam polinómios inteiros a uma variável, P(x), e estejam representados na sua forma canónica (sem termos repetidos). Defina a estrutura das classes* `Termo` *e* `Polinómio`*, implementando os métodos insereTermo (insere um termo novo), alteraCoef (muda o coeficiente de dado termo), removeTermo (remove o termo de grau dado) e calcula (que dado um polinómio segundo a representação proposta, calcula o valor de P(X) para um inteiro X dado).*

10.- Desenvolva uma classe ArrayBiDim correspondente a uma representação de array bidimensionais. As instâncias desta classe deverão rèponder a mensagens de manipulação dos seus valores através de referências aos números de linha e de coluna, para inserção, alteração, consulta e remoção de tais valores. Métodos de varrimento por linha e por coluna deverão ser igualmente implementados, dando como resultado arrays ou Vectores contendo tais valores.

CONTROLO DOS ACESSOS EM JAVA 1.1
PROTECÇÃO E ENCAPSULAMENTO EM JAVA 1.1

O CONTROLO DO ACESSO AOS MEMBROS DE UMA CLASSE DEFINIDA EM JAVA, VARIÁVEIS E MÉTODOS, E ATÉ À PRÓPRIA CLASSE, É DEFINIDO ATRAVÉS DE 4 ESPECIFICADORES DE ACESSO, DESIGNADAMENTE,

private
protected
public
package

QUE DEFINEM AS REGRAS DE ACESSO AOS MEMBROS DE UMA CLASSE EM 4 CONTEXTOS POSSÍVEIS EM JAVA, DESIGNADAMENTE,

classe
subclasse
package
mundo

Temos pois que estudar e perceber como poderemos preencher a tabela 4x4 que representa todas as combinações:

DECLAR.	classe	subclasse	package	mundo
private	*			
protected	*	(*)	*	
public	*	*	*	*
package	*		*	

NOTA: ESTA É A TABELA DO JAVA 1.1!

DI

NÍVEL DE ACESSO: private

- Este é o nível de acesso mais restritivo de JAVA.
  Um membro de uma classe com atributo private, como se vê pela tabela, pode apenas ser acedido no contexto da classe em que é definido (seja variável de instância ou método de instância).

- MÉTODOS private podem apenas ser "invocados" (via mensagem) em métodos da mesma definição de classe. Funcionam pois como métodos auxiliares.

- VARIÁVEIS private <u>devem</u> apenas ser acedidas por métodos de instância da classe a que pertencem e <u>podem</u> apenas ser referidas no código destes directamente (o que não se aconselha); porém, mesmo assim, só o podem ser em contextos tão fechados quanto as próprias instâncias da classe.

EXEMPLO:

```
class A {
  // var. instância
  private int a_priv;
  -----------
  // met. instância
  private void met_priv();
  ----------- }
}
```

UTILIZAÇÕES:

A) ERRADA

```
class B {
  // met. instância
  void acesso() {
      A aux = new A();
      aux.a_priv = 99;   // ilegal
      aux.m_priv();      // ilegal
  }
  -------
}
```

ERRO: Um objecto de tipo B não pode aceder às componentes privadas de um objecto de tipo A.

B) CORRECTA (mas apenas em termos de JAVA, não em PPO!)

```
class A {
  ---- ----
  ---- ----

  boolean igual (A outro) {
    return (this.a_priv == outro.a_priv);
  }
```

OK: No contexto da classe (ie. para todas as suas instâncias), sendo todas as instâncias do mesmo tipo, não há restrições!

NÍVEL DE ACESSO: protected

- Têm acesso a membros protected de uma classe A:

  1) A classe (ie. todas as instâncias);

  2) Todas as classes do mesmo "package", incluindo portanto naturalmente todas as subclasses de A;

  3) Subclasses de A existentes noutros "packages", embora não podendo aceder aos membros protected de instâncias da superclasse A mas apenas das suas instâncias ou de outras subclasses de A.

EXEMPLO:

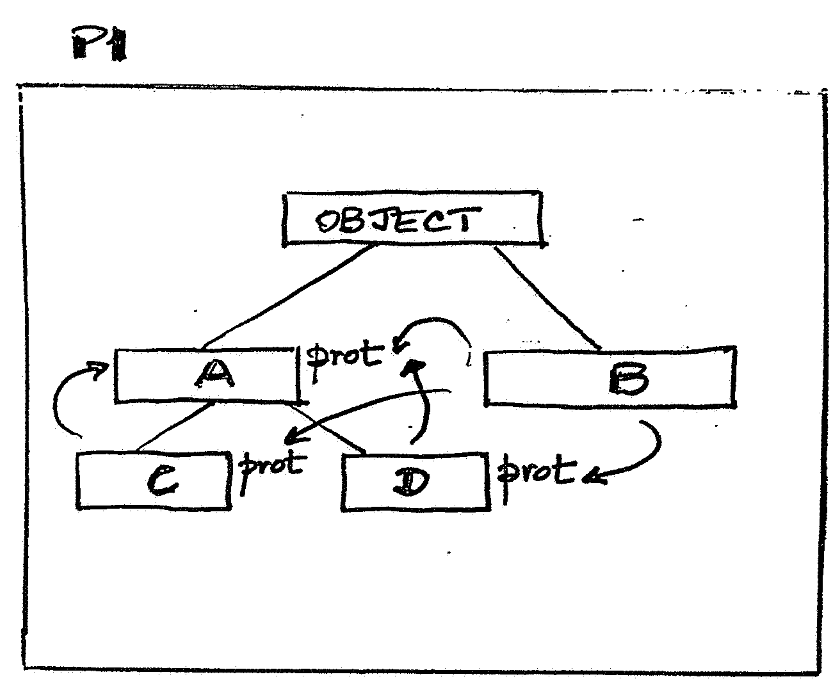
```
package P1;

class A {

    protected int var_prot;
    protected void met_prot( ) {
        System.out.println("sou o met-prot!");
    }
    ------
}
```

NOTA: Vamos criar uma classe B, dentro do package P1, não relacionada com A.

```
package P1;
class B {
   void acesso() {
      A aux = new A();
      aux.var_prot = 10;      // legal
      aux.met_prot();         // legal
   }
   -------
}
```



NOTA: DENTRO DO MESMO "PACKAGE", MEMBROS PROTECTED SÃO ACESSÍVEIS GLOBALMENTE.

Como VEREMOS:

PROTECTED = PUBLIC NO "PACKAGE" + CASOS ESPECIAIS.

CASO ESPECIAL : UMA CLASSE A1 QUE VAI SER SUBCLASSE DE A MAS QUE VAI RESIDIR NO "PACKAGE" P2.

```
import P1.* ;
package P2 ;

class A1 extends A { // herda "protected"
    void acesso(A a, A1 a1) {
        a.var_prot = 99 ;      // ilegal
        a1.var_prot = 99 ;     // OK!
        a.met_prot() ;         // ilegal
        a1.met_prot() ;        // OK!
    }
    ---------
}
```

CONCLUSÃO :

SUBCLASSES DE UMA DADA CLASSE A QUE PERTENÇAM A UM PACKAGE DIFERENTE DO DA DEFINIÇÃO DE A, NÃO PODEM ACEDER A MEMBROS PROTECTED DE INSTÂNCIAS DA SUA SUPERCLASSE,

```
a.var_prot = 99 ;
a.met_prot() ;
```

MAS APENAS AOS MEMBROS PROTECTED DAS SUAS PRÓPRIAS INSTÂNCIAS (OU DE OUTRAS SUBCLASSES DE A DO MESMO "PACKAGE").

NÍVEL DE ACESSO: public

- QUALQUER INSTÂNCIA DE QUALQUER CLASSE DE UM QUALQUER "PACKAGE" TEM ACESSO A UM MEMBRO DE QUALQUER CLASSE DECLARADO COMO PUBLIC;

PUBLIC ⇒ GLOBAL (!!)

NÍVEL DE ACESSO POR OMISSÃO : package
(ie. SE NÃO ESPECIFICADO)

- O NÍVEL DE ACESSO É CONFINADO AO "PACKAGE", MAS, DENTRO DESTE, E CURIOSAMENTE, NÃO ÀS SUBCLASSES DA CLASSE MAS A TODAS AS OUTRAS CLASSES NÃO RELACIONADAS.

METÁFORA: CONFIAM-SE OS SEGREDOS AOS AMIGOS MAS NADA SE DIZ À FAMÍLIA, EM PARTICULAR OS DESCENDENTES!

EXERCÍCIO:

COMO ARTICULAR AS DECLARAÇÕES DE NÍVEL DE ACESSO COM AS PREOCUPAÇÕES DE PPO, EM PARTICULAR:

a) ENCAPSULAMENTO;
b) REUTILIZAÇÃO;
c) HERANÇA E PROGRAMAÇÃO DIFERENCIAL;
d) MODULARIDADE;
e) SEGURANÇA;